XSLT

Programmer's Reference

Second Edition

Michael Kay

XSLT

Справочник программиста

Второе издание

Майкл Кэй

Санкт-Петербург
2002

Майкл Кэй

XSLT. Справочник программиста

Перевод М. Зислиса, Е. Морозова,
Г. Нифонтовой, М. Смирнова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Е. Морозов*
Редактор	*А. Лосев*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко*

Кэй М.
XSLT. Справочник программиста. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 1016 с., ил.
ISBN 5-93286-039-1

С ростом популярности стандарта XML появилась необходимость в адекватном средстве обработки XML-данных. Консорциумом W3C был разработан XSLT – гибкий и переносимый язык преобразований, используемый и на стороне сервера, и на стороне клиента. Он быстро завоевал популярность: свои реализации XSLT-процессоров выпустили Microsoft, Oracle и Apache Software Foundation. Эта книга, написанная автором XSLT-процессора Saxon, поможет освоить XSLT, научит программированию на нем и послужит отличным справочником в дальнейшей работе.

Издание состоит из четырех частей: подробного введения в историю возникновения XSLT, его роль, основные понятия и внутреннюю структуру; справочника по элементам и функциям XSLT с детальным описанием всех языковых конструкций; руководства по разработке программ с советами по дизайну и примерами; приложений, содержащих сведения о последних версиях XSLT-процессоров. Издание проиллюстрировано большим количеством работающих примеров и написано для программистов, знакомых с XML и HTML и желающих воспользоваться мощными возможностями и совместимостью языка XSLT для создания новых веб-приложений.

ISBN 5-93286-039-1
ISBN 1-861005-06-7 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 28.03.2002. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Объем 63,5 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Об авторе

Майкл Кэй работает в Software AG, где занимается стандартами и интерфейсами для создаваемой ее специалистами линейки продуктов XML, в основном для баз данных Tamino. Он также представляет Software AG в рабочей группе W3C по XSL. До этого большую часть своей карьеры Майкл занимался разработкой программных продуктов и систем в ICL. Его специализация (и степень Ph.D., полученная в Кембридже) имеет прямое отношение к базам данных – он проектировал программные продукты для сетевых, реляционных и объектно-ориентированных баз данных, а также текстовые поисковые машины. В сообществе XML Майкл Кэй известен как разработчик XSLT-процессора Saxon – продукта с открытым исходным кодом – первой полностью совместимой реализации стандарта XSLT.

Майкл живет в Рединге, графство Беркшир, с женой и дочерью. Увлекается, как можно понять из примеров в этой книге, генеалогией и хоровым пением.

Благодарности

Прежде всего, я хотел бы отметить титанический труд рабочей группы W3C по XSL, создавшей язык XSLT. Без их усилий не было бы ни языка, ни этой книги. Первое издание, написанное еще до моего вступления в группу, было «взглядом со стороны», и я постарался сохранить этот же стиль во втором издании, несмотря на то, что теперь надо мной тяготеет доля ответственности за текущее состояние спецификации.

Особая моя благодарность Джеймсу Кларку, редактору спецификаций XSLT и XPath, который вежливо и быстро отвечал на великое множество вопросов.

Многое из того, что я знаю об XSLT, почерпнуто от участников списка рассылки по XSL, притом не только от экспертов, отвечающих на вопросы, но и от огромного количества новичков, их задающих. Масса технических приемов и пояснений появилась во втором издании благодаря идеям, витавшим в этой конференции.

Еще я хочу поблагодарить ICL и Software AG, моих работодателей, которые поддерживали и поощряли меня во время работы над этим проектом.

Редакторы Wrox Press и технические рецензенты внесли неоценимый вклад в подготовку моей книги, указав на многочисленные места, которые требовали улучшения.

И наконец, я еще раз хочу поблагодарить за поддержку Пенни и Пиппу, которые восприняли новость о том, что я планирую начать работу над вторым изданием, со вздохом смирения.

Введение

Не будет преувеличением сказать, что язык XML стал самым большим потрясением, всколыхнувшим Сеть, за все время ее существования – чуть меньше десятилетия. Он также – одно из самых многообещающих средств мира управления информацией. Сам по себе XML фактически ничего **не делает**, вся его заслуга в том, что он обеспечивает способ структурирования информации и передачи ее из одной программы в другую. До сих пор для создания приложения, которое обладало бы достоинствами XML, пришлось бы писать код низкого уровня, используя интерфейсы (такие как DOM и SAX), требующие очень серьезного знания способа организации данных.

Это напоминает мне методы, с помощью которых раньше, до появления SQL, приходилось обращаться к базам данных (вы видели седину в волосах на фотографии на обложке книги); и когда я впервые увидел язык преобразований XSL (XSL Transformation), XSLT, я понял, что он сыграет роль SQL для Сети: высокоуровневый язык оперирования данными, который превратит XML из просто формата для сохранения и передачи данных в активный источник информации, к которому можно обращаться с запросами и которым можно управлять с помощью гибких, декларативных методов.

В начале 1998 года я искал способы включения XML в средства управления содержимым для веб-публикаций и начал разрабатывать библиотеку Java под названием Saxon, чтобы предоставить высокоуровневые интерфейсы для управления XML. Я очень критично относился к первым проектам XSL, но одновременно с ускорением процесса стандартизации я начал осознавать факт возникновения полезного и мощного языка и решил изменить направленность программного пакета Saxon, чтобы превратить его в XSLT-процессор. Когда 16 ноября 1999 года вышла, наконец, рекомендация по XSLT версии 1.0, я уже через 17 дней смог объявить о готовой, полностью совместимой реализации с открытыми исходными текстами.

Хотя я осознавал мощь и возможности языка XSLT, я видел также, что он включает в себя некоторые новые и трудные концепции, с которыми сражаются первые пользователи. Это напомнило мне о начале работы с SQL: сейчас это трудно представить, но когда SQL только появился, его радикально новые концепции, в частности внутренние и внешние соединения и трехзначная логика, были весьма трудны для понимания.

Именно поэтому я написал эту книгу. XSLT – захватывающий и мощный язык, но не настолько простой, чтобы постичь его сразу. До сих пор имелось очень немного доступных ресурсов по XSLT, кроме официальной спецификации W3C непосредственно, которая примерно так же читабельна для большинства программистов, как страницы налогового законодательства. Я надеюсь, что эта книга заполнит имеющийся пробел и поможет освоить огромный потенциал этого нового языка.

Для кого эта книга?

Я написал эту книгу для профессиональных разработчиков программ, которые уже использовали ряд языков в прошлом и теперь хотят научиться работать с XSLT для создания новых приложений. Я предполагаю, что читатели имеют основное представление об XML, хотя понимаю, что, если они и знали когда-то, что такое внешняя общая анализируемая сущность, то, возможно, уже забыли это. Я рассчитываю также на понимание общей архитектуры Сети и HTML.

Я не предполагаю обязательного знания любого конкретного языка программирования, хотя писал книгу в расчете на то, что читатели имеют некоторый опыт программирования. Если весь этот опыт заключается лишь в написании страниц HTML с небольшими включениями Javascript, часть материала книги может показаться нелегкой.

Зачем второе издание?

Основная причина выхода второго издания всего лишь через год после первого в том, что язык XSLT теперь гораздо лучше поддерживается программными продуктами, а книга, которая описывает только язык, ничего не сообщая о доступных программах, делает только половину дела. Год назад MSXML3 от Microsoft был ранней реализацией, в которой обеспечивалась только половина стандарта; а сегодня он полностью поддерживает стандарт и полностью соответствует ему. И не только Microsoft работает в этом направлении; в приложениях к книге я описал дюжину других XSLT-процессоров, из которых можно выбирать.

Произошли изменения и в самих стандартах – вышел рабочий проект XSLT 1.1. В книге ему уделено много внимания, потому что многие особенности проекта уже реализуются в программных продуктах.

Наконец, сообщество XSLT изобрело много оригинальных способов использования языка XSLT 1.0 – путей обхода его ограничений, которые не устраивают кого-то. В первом издании я утверждал категорически (и необдуманно), что не существует возможности построения пересечения двух наборов узлов; но прежде чем высохли чернила, я уже нашел решение проблемы. И блестящее применение Стивом Мюнчем (Steve Muench) функции `key()` для решения проблем группировки, безусловно, должно найти свое место в книге.

Что охватывает книга?

Основная часть этой книги (главы от 1 до 10) посвящена XSLT (и вместе с ним XPath) как языку, безотносительно к конкретным программам. В приложениях описано несколько наиболее известных (и несколько почти неизвестных), связанных с XSLT продуктов. Здесь больше внимания уделено изделиям, которые вероятнее всего будут использоваться читателями: MSXML3 от Microsoft, XDK от Oracle, Saxon и Xalan, – но также рассказывается и о других, менее известных, процессорах, в частности о 4XSLT и Unicorn. Ассортимент XSLT-продуктов очень подвижен, новые процессоры появляются каждый месяц (о некоторых потом больше не слышно), поэтому автор не стремится включить все; эта часть книги неизбежно устареет довольно быстро, хотя, по счастью, крупные программные пакеты типа MSXML3 уже достигли некоторого уровня стабильности.

Книга **не затрагивает** ранний диалект XSL, который выпустила компания Microsoft при выходе Internet Explorer 5 в 1998 году (здесь за неимением лучшего названия этот диалект назван WD-xsl[1]). XSLT прошел длинный путь со времени выхода раннего рабочего проекта, и описание двух таких разных версий языка в одной книге будет только вносить путаницу. Хотя WD-xsl широко используется и установлен на многих рабочих местах, компания Microsoft переместила свои усилия на продвижение W3C-спецификации XSLT 1.0, поэтому WD-xsl имеет явно ограниченный срок службы.

Как устроена книга?

Основное содержание книги естественным образом делится на три части.

В первую часть входят главы 1–3. Их задача – объяснить концепции языка XSLT. Глава 1 посвящена роли и цели языка, истории его возникновения и развития и факторам, которые повлияли на его проектирование, – ее можно было бы назвать «Почему XSLT?». Глава 2 раскрывает значение концепции **преобразования** и обсуждает модель обработки, которая основана на описании взаимосвязи конечного дерева с исходным деревом в таблице стилей XSLT. Далее в главе 3 показана внутренняя структура таблицы стилей, взаимосвязь ее модулей и основные сведения о компонентах, которые могут в ней присутствовать.

Вторая часть, главы 4–8, содержит справочную информацию. Цель этих глав – дать полное описание всех особенностей языка, подробно объяснить их синтаксис и производимые ими действия, дать советы по их использованию и на примерах показать, как все это работает. Материал в этих главах

[1] WD – сокращение от Working Draft – можно примерно перевести как «рабочий проект» или «рабочий черновик». Автор дал такое название языку, видимо, потому, что Microsoft выпустила его, основываясь на Рабочем Проекте спецификации языка XSL, которая впоследствии сильно изменилась. Подробнее об этом будет рассказано в главе 1. – *Примеч. ред.*

упорядочен скорее по принципу легкости поиска нужной информации, чем по последовательности изложения: элементы XSLT – в главе 4, выражения XPath – в главе 5, синтаксис образцов – в главе 6, а библиотеки стандартных функций – в главе 7. Наконец, в главе 8 описаны введенные в рабочий проект XSLT 1.1 новые возможности (которые еще могут претерпеть изменения) для вызова из таблицы стилей XSLT внешних функций, написанных на Java или JavaScript. Многие из этих глав организованы в алфавитном порядке, что позволяет быстро найти сведения о конкретной особенности языка.

Третья часть книги, главы 9 и 10, предназначена помочь применению языка XSLT для разработки реальных промышленных приложений. В главе 9 обсуждается ряд образцов проектирования, а в главе 10 приведены три подробных примера разработок.

Наконец, в большинстве приложений даны сведения о конкретных программных продуктах. Информация об MSXML3 от Microsoft, Oracle, Saxon и Xalan выделена в отдельные приложения, где описываются API, используемые для вызова этих продуктов, взаимосвязь с другими программными продуктами того же поставщика и подробно обсуждаются зависящие от производителя дополнения и ограничения. В приложении E дан менее детальный обзор возможностей широкого круга других изделий, включая и XSLT-процессоры (4XSLT, Infoteria, Sablotron, Transformiix и Unicorn), и инструментальные средства разработки XSLT, в том числе Stylus Studio, Whitehill's XSL Composer и XML Spy. Автор не пытался привести здесь всю необходимую информацию по каждому продукту, а скорее дал сводку характеристик каждого из них, чтобы читатели имели возможность сделать обоснованный выбор и исследовать его далее.

Приложение F несколько выпадает из общего стиля. В нем описывается TrAX API, который представляет собой Java API для управления XSLT-преобразованиями. Он занимает положение среднего звена между спецификациями, не зависящими от поставщика, и данными по конкретным изделиям (этот API выпущен компанией Sun и в настоящее время реализован в процессорах Saxon и Xalan-Java).

В конце книги приведены полезный глоссарий терминов и индекс.

Другие ресурсы по XSLT

Если читатель не сможет найти необходимые сведения в книге, у него есть возможность поискать ответы на свои вопросы в Сети, где есть несколько хороших ресурсов по данной тематике. Чтобы не приводить длинный список сайтов, имеющих отношение к XSLT, перечислим только те из них, которые содержат хорошие подборки ссылок[1]:

[1] Русскоязычным читателям можно предложить сайт *xmlhack.ru*, на котором находятся также ссылки на другие русскоязычные ресурсы по XML и XSLT. – *Примеч. науч. ред.*

- *http://www.w3.org/Style/XSL/*

  Это официальный сайт Консорциума World Wide Web. Там содержатся все изданные спецификации и рабочие проекты, а также большое количество новостей, статей и официальных документов.

- *http://www.xslinfo.com/*

  Это хорошо организованный сайт, созданный Джеймсом Таубером (James Tauber) и состоящий почти полностью из классифицированного перечня программного обеспечения XSL, статей, руководств и других ресурсов.

- *http://www.oasis-open.org/cover/xsl.html*

  Страницы ссылок Робина Ковера (Robin Cover) обеспечивают всесторонний охват всего, что случается или когда-либо случалось в мире SGML и XML. Здесь приведена ссылка на его раздел по XSL.

- *http://www.xml.com/*

  Хороший сайт с критическими обзорами и сводками новостей, касающимися мира XML вообще.

- *http://msdn.microsoft.com/xml/*

  Этот URL является хорошим местом для знакомства со взглядом Microsoft на мир XML и XSL. Множество полезных статей и официальных документов, не всегда ограничивающихся только их собственными продуктами.

- *http://www.mulberrytech.com/xsl/xsl-list*

  Домашняя страница почтовой рассылки XSL, очень объемного, но чрезвычайно полезного обсуждения всех проблем XSLT, от вопросов новичков до глубоких теоретических дебатов. Даже если нет времени для участия в дискуссиях, можно воспользоваться архивом и ссылками с домашней страницы на различные FAQ и обучающие сайты.

Что необходимо для работы с этой книгой?

Чтение книги о новом языке не сделает читателя экспертом по языку; для этого требуется применять язык и учиться на своем опыте. В этой книге есть много примеров, которые стоит запустить, но более важно то, что они могут стимулировать читателей опробовать собственные идеи.

Все коды работающих примеров можно загрузить со страниц веб-сайта *http://www.wrox.com*, посвященных этой книге.

В приложениях перечислены некоторые доступные XSLT-продукты, и, что немаловажно, там есть хороший выбор. В основном это свободные программы (freeware), хотя всегда следует читать условия лицензирования.

Можно начать с продукта Microsoft – MSXML3. Его можно загрузить с вебстраницы *http://msdn.microsoft.com/xml*, а в приложении А даны инструкции по установке. MSXML3 позволяет запускать таблицы стилей XSLT прямо в броузере Internet Explorer. Этот программный пакет хорошо соответствует спецификации языка XSLT 1.0. Однако он имеет один большой недостаток: отсутствует понятная диагностика ошибок.

По этой причине для платформы Windows можно предложить начинать с:

- **Instant Saxon**, доступного по адресу:

  http://users.iclway.co.uk/mhkay/saxon/instant.html

- **Programmer's File Editor (PFE)**, доступного на различных архивных сайтах, включая:

  http://www.simtel.net/pub/simtelnet/win95/editor/pfe101i.zip

Рекомендации относительно Instant Saxon можно считать предвзятыми, поскольку он создан автором книги. Достоинство Saxon в том, что он полностью отвечает стандарту, является открытым программным продуктом и его легко устанавливать и использовать. Все примеры, приведенные в книге, проверены с использованием Saxon. Они должны также работать с процессорами MSXML3, Xalan и Oracle XSL, – все это замечательные изделия, которые обещают 100-процентное соответствие стандарту, но это не проверялось.

Подобно большинству других Java-основанных процессоров, Saxon требует создания и редактирования файлов XML и таблиц стилей в текстовом редакторе и запуска их из командной строки операционной системы. Для тех, кто вырос на Windows, это может показаться довольно примитивным. PFE рекомендован, потому что он содержит интуитивный текстовый редактор, который позволяет открывать сразу большое количество файлов, и потому что он дает доступ к командной строке операционной системы привычным для Windows способом. Вместо этого редактора можно использовать и другие. Автор не настаивает только на PFE.

В приложениях описывается несколько программных продуктов для не-Windows платформ, написанных на Java и выполняющихся почти в любой среде. Нужно только установить виртуальную машину Java на своем компьютере и ознакомиться с механизмом установки и выполнения приложений Java в конкретной среде.

Условные обозначения

Для удобства пользования книгой принят ряд условных обозначений.

Работающие примеры – те, которые можно загрузить и опробовать непосредственно – оформлены следующим образом:

Образец примера

Исходный документ

В этом разделе даются исходные данные XML, входной документ для преобразования. Если имя файла выглядит как `example.xml`, этот файл можно найти в архиве и загрузить с веб-сайта Wrox[1] *http://www.wrox.com/*, как правило, из подкаталога, содержащего все примеры для данной главы.

```
<source data="xml"/>
```

Таблица стилей

В этом разделе описывается таблица стилей XSLT, используемая для данного преобразования. Будет приведено еще одно имя файла типа `style.xsl`, по которому можно найти эту таблицу стилей в архиве загрузки веб-сайта Wrox.

```
<xsl:stylesheet...
```

Конечный документ

В этом разделе приводится результат применения таблицы стилей к исходным данным в формате XML, в форме листинга HTML или в виде изображения на экране.

```
<html>...</html>
```

Фрагменты кода обычно оформляются примерно так:

```
<data>
XML-данные или XSLT-код
</data>
```

Что касается стилей в тексте:

- Важные термины при первом употреблении выделяются полужирным шрифтом: **важные слова**.
- Имена файлов и коды внутри текста выделяются моноширинным шрифтом: `dummy.xml`.
- Текст интерфейса пользователя пишется наподобие: File/Save as.
- Двойные угловые кавычки применяются для явного отделения фрагмента кода от окружающего текста, например: «a=3;». Они использовались вместо обычных одинарных или двойных кавычек отчасти потому, что они лучше выделяются, а также во избежание любой неоднозначности с кавычками, являющимися частью образца кода; это означает, что можно

[1] Многие примеры переведены на русский язык, поэтому не надо удивляться, если вдруг в скачанном файле будет только английский текст. – *Примеч. науч. ред.*

записывать примеры по типу «select="'Madrid'"», где кавычки " и ' – часть образца кода, а двойные угловые кавычки « и » – нет.

- Имена элементов XML, имена функций и атрибутов выделяются жирным моноширинным шрифтом: <xsl:value-of>, concat(), href.

Кроме того:

> ***Подобное выделение содержит важную информацию по обсуждаемому вопросу, которую следует помнить.***
>
> *В то время как подобный стиль используется для вопросов, находящихся в стороне от текущего обсуждения.*

Синтаксические правила (которые встречаются главным образом в главах 5 и 6) в значительной степени соответствуют соглашениям, принятым в стандартах W3C, за исключением использования двойных угловых кавычек для заключения в них литералов. Основными условными обозначениями, используемыми в этих синтаксических правилах, являются:

Выражение	Значение
clause|paragraph	Либо clause, либо paragraph. Значения clause и paragraph будут определены отдельными правилами: это **нетерминальные конструкции**. Вертикальная черта «|» используется для разделения вариантов.
«!»|«?»	Либо восклицательный, либо вопросительный знак. Слова или символы, заключенные в двойные угловые кавычки, должны использоваться, как написано: это **терминальные конструкции**.
adjective noun «.»	adjective, сопровождаемое noun, за которым следует точка.
adjective? noun	Только noun или adjective, сопровождаемое noun. «?» указывает, что предшествующая конструкция необязательна.
adjective* noun	Ни одного или более adjective, сопровождаемых noun.
adjective+ noun	Одно или более adjective, сопровождаемых noun.
sentence («?»|«!»)	sentence, сопровождаемое вопросительным или восклицательным знаком. Круглые скобки обозначают группировку.

Поддержка читателей

Издательство Wrox осуществляет поддержку книг тремя способами. Читатели могут:

- Ознакомиться с обнаруженными опечатками на веб-сайте *www.wrox.com*.
- Принять участие в форумах на веб-сайте *p2p.wrox.com*.
- Послать по электронной почте запрос на техническую поддержку или просто свое мнение о книгах Wrox.

Опечатки

На веб-сайте *www.wrox.com* можно ознакомиться с опечатками в книге, просто перейдя на страницу, посвященную этой книге, где будет ссылка на перечень опечаток.

Почтовые рассылки

Можно принять участие в дискуссионных форумах пользователей на веб-сайте *p2p.wrox.com*, где можно задавать вопросы автору книги, рецензентам и другим специалистам в этой области. Почтовая рассылка по XSLT находится в разделе 'XML'. Можно присоединиться к списку подписчиков или заказать эту рассылку в форме еженедельного дайджеста. Если нет времени или возможности получать рассылку, то можно просматривать ее архив, доступный в Интернете. Там обеспечена возможность поиска по конкретной тематике или по ключевым словам. Поскольку эти рассылки администрируются, это гарантирует быстрый поиск полезной, точной информации. Почтовые сообщения могут редактироваться модераторами или перемещаться в соответствующие разделы, что делает этот ресурс очень эффективным. Сообщения, не относящиеся к теме обсуждения, и спам удаляются, а адреса электронной почты подписчиков защищены уникальной системой Lyris от веб-роботов, которые могут автоматически собирать адреса участников почтовых рассылок и телеконференций.

Поддержка по электронной почте

При обнаружении новой опечатки в книге можно сообщить о ней на веб-сайт или прямо обсудить это с экспертом, который детально знает книгу, послав сообщение по адресу *support@wrox.com*. В этом сообщении желательно указать следующие детали:

- В поле Subject – название книги, последние четыре цифры ISBN книги и номер страницы с предполагаемой опечаткой.
- В теле сообщения – свое имя, контактную информацию и описание обнаруженной опечатки.

Сообщите нам свое мнение

Автор книги и коллектив издательства Wrox упорно трудились, чтобы сделать чтение этой книги приятным и полезным удовольствием для читателей, поэтому мнение читателей очень ценно. Мы всегда готовы прислушаться к тому, что читателям понравилось больше всего и какие улучшения, по их мнению, возможны. Автор книги и коллектив издательства Wrox очень ценят обратную связь и учитывают в последующей редакционной работе как критику, так и похвалу. При необходимости комментарии и вопросы пересылаются автору. Все, кто желает высказать что-то по поводу книги, могут прислать сообщение по адресу: *feedback@wrox.com*.

1

Общее представление об XSLT

Эта глава призвана дать общее представление о языке XSLT. В ней говорится о сути XSLT и задачах, для выполнения которых он предназначен. О том, к какому типу языков принадлежит XSLT, как он зарождался и как соотносится со всеми другими технологиями, которые используются в типичных веб-приложениях. Здесь мало внимания уделяется тому, как выглядит таблица стилей XSLT и как она работает: это обсуждается позже, в главах 2 и 3.

Сначала речь пойдет о задаче, для выполнения которой разработан XSLT, – **преобразовании** – и о том, зачем нужно преобразовывать XML-документы. На тривиальном примере будет показано, что это означает практически.

Далее обсуждается соотношение XSLT с другими стандартами растущего семейства языков XML, чтобы в контексте показать его назначение и объяснить, как он дополняет другие стандарты.

Также здесь будет описано, что из себя представляет язык XSLT, и хронологически показано, как он развивался. Нетерпеливые читатели могут пропустить хронологию и сконцентрироваться на использовании языка, но рано или поздно они задумаются, почему же он создан именно таким. Автор надеется, что в этот момент они вернутся, чтобы прочесть о том, как XSLT появился на свет.

В конце будут затронуты различные пути использования XSLT в рамках общей архитектуры приложения, в котором неминуемо будут задействованы многие другие технологии и компоненты, играющие свои собственные роли.

Что такое XSLT?

XSLT, что означает **eXtensible Stylesheet Language: Transformations (расширяемый язык таблиц стилей: Преобразования)**, – это язык, который, со-

гласно самой первой фразе в спецификации (ее можно найти по адресу *http://www.w3.org/TR/xslt*), предназначен, в первую очередь, для преобразования одного XML-документа в другой. Однако XSLT способен не только преобразовывать XML в HTML и многие другие основанные на тексте форматы, поэтому более общим определением будет следующее:

XSLT – это язык для преобразования структуры XML-документа.

Зачем это нужно? Чтобы правильно ответить на этот вопрос, выясним сначала, почему XML завоевал такой успех и вызвал такой ажиотаж.

Зачем преобразовывать XML?

Технология XML – простой, стандартный способ взаимообмена структурированными текстовыми данными между компьютерными программами. Рост ее популярности связан отчасти с тем, что для прочтения XML-документов или для их создания достаточно простого текстового редактора, но это никак не умаляет основного назначения XML – служить средством связи между программными системами. В этом плане XML удовлетворяет двум насущным потребностям:

- **Отделение данных от их представления**. Потребность отделять информацию (например, прогноз погоды) от деталей способа ее отображения на конкретном устройстве. Необходимость в этом все возрастает по мере расширения круга устройств, имеющих доступ к Интернету. Организации, вложившие свои средства в создание полезных источников информации, должны иметь возможность поставлять их не только для традиционных компьютерных веб-броузеров (которые сейчас тоже очень разнообразны), но также и для телевизоров, и сотовых телефонов, не говоря уже о сохраняющейся потребности в печатной информации.
- **Передача данных между приложениями**. Потребность передавать информацию (заказы и счета) из одной организации в другую, не тратя при этом средства на интеграцию специально заказываемого программного обеспечения. С увеличением темпов роста электронной коммерции объем данных, которыми предприятия обмениваются между собой, ежедневно растет, и эта потребность также становится все более насущной.

Безусловно, эти два пути использования XML не являются взаимоисключающими. Счет можно представить на экране или ввести в финансовое приложение, а метеорологические данные вместо прямого отображения могут быть также проанализированы, проиндексированы и сгруппированы самим получателем. Еще одним ключевым преимуществом XML является его способность объединять миры документов и данных, обеспечивая единый способ представления структуры, независимо от того, предназначена ли информация для человека или для машины. Идея в том, что XML-данные, получаемые в итоге людьми или приложением, очень редко будут использоваться прямо в том виде, в каком они поступают: сначала они должны быть преобразованы в нечто другое.

Если данные предназначены для человека, это «нечто» может быть документом, который можно отобразить или распечатать: например HTML-файлом, PDF-файлом или даже звуковым файлом. Конвертирование данных XML в HTML для отображения – вероятно, наиболее частое применение XSLT сегодня, и именно оно будет использоваться в большинстве примеров в этой книге. Получив данные в формате HTML, их легко отобразить с помощью любого броузера.

Для передачи данных между разными приложениями необходимо преобразовать их из модели, используемой одним приложением, в модель, используемую другим. Для загрузки данных в приложение подходят различные форматы: файлы с разделителями-запятыми, сценарии SQL, HTTP-сообщения или последовательность вызовов какого-либо программного интерфейса. Как вариант это может быть и XML-файл, использующий словарь, отличный от словаря исходного документа. С распространением основанной на XML электронной коммерции роль XSLT в преобразовании данных между приложениями также становится все более важной. То, что XML используют все, не может отменить потребность в преобразовании данных. Всегда будет одновременно существовать множество стандартов. Например, газетная индустрия, вероятно, будет использовать для обмена статьями не такие форматы, как радиовещание. Кроме того, никогда не отпадет нужда в таких вещах, как извлечение адреса клиента из заказа на товары и внесение его в бланк счета. Так что электронная коммерция будет все больше становиться процессом определения способа извлечения и комбинирования данных из одного набора XML-документов для создания другого набора XML-документов, и XSLT – идеальный инструмент для этой работы.

В конце этой главы будут приведены конкретные примеры использования XSLT для преобразования XML. Здесь же главным было – дать читателю почувствовать важность и полезность преобразования XML. Прежде чем перейти к более подробному обсуждению языка XSLT и увидеть, как он работает, рассмотрим пример, хорошо показывающий многообразие форматов, в которые с помощью языка XSLT можно преобразовать XML-документы.

Пример: Преобразование музыки

Веб-страница *http://www.xml.org/xmlorg_registry/index.shtml* – превосходный реестр словарей и схем XML.

Там можно найти с полдюжины различных схем XML для описания музыки. Все они были созданы для разных задач: требования к языку разметки, который используется издателем печатных нотных альбомов, отличаются от тех, что предъявляются к языку, позволяющему слушать музыку с помощью броузера. Например, язык MusicML ориентирован на графическое отображение музыкальных нот; ChordML предназначен для кодирования музыкального сопровождения (гармонического аккомпанемента) песен; MusicXML разработан для представления музыкальных партитур, в особенности западных нотных записей XVII века, в то время как гораздо более академичный язык разметки музыки MML (Music Markup Language), созданный в

университете Претории (University of Pretoria), предназначен для серьезного музыковедческого анализа и охватывает восточные, африканские, а также западные музыкальные стили (рис. 1.1).

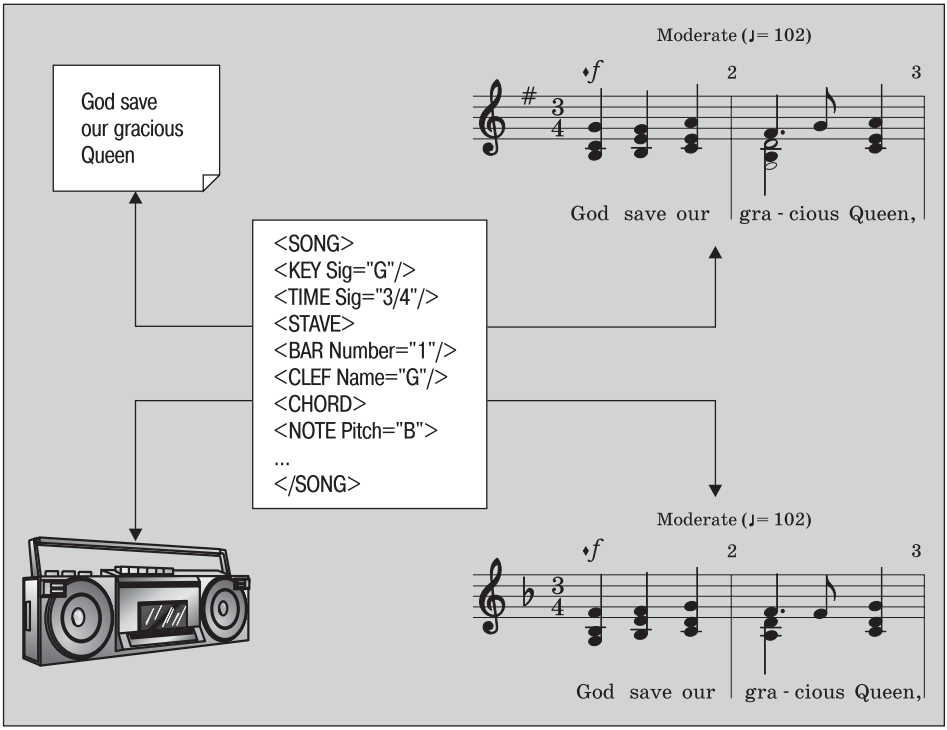


Рис. 1.1. *Разметка музыки*

Таким образом, с помощью XSLT можно произвести разметку музыки многими различными способами:

- Можно использовать XSLT для конвертирования музыки из одного из этих представлений в другое, например из MusicXML в MML.
- Можно использовать XSLT для преобразования музыки из любого из этих представлений в визуальные нотные записи, генерируя основанный на XML векторный графический формат SVG.
- Можно использовать XSLT для проигрывания музыки на синтезаторе, генерируя файлы MIDI (Musical Instrument Digital Interface, Цифровой интерфейс музыкальных инструментов).
- Можно использовать XSLT для изменения музыки, например, для транспонирования ее в другую тональность.
- Можно использовать XSLT для извлечения текста песни в документ HTML или текстовый XML.

Как видите, XSLT служит для преобразования XML-документов не только в HTML!

Как XSLT преобразовывает XML?

Возможно, читатели книги уже задумывались, как же работает XSLT с XML-документом при конвертировании его в требуемый формат. Эта обработка включает обычно два аспекта:

- Первая стадия – структурное преобразование, при котором структура данных исходного XML-документа конвертируется в структуру, отражающую желаемый вывод.
- Вторая стадия – форматирование, при котором новая структура выводится в заданном формате, например HTML или PDF.

Вторая стадия охватывает вопросы, обсуждавшиеся в предыдущем разделе. Структуру данных, к которой приводит первая стадия, можно вывести как файл формата HTML, текстовый или XML-документ. Вывод HTML позволяет людям просматривать полученные данные прямо в броузере или в любом современном текстовом процессоре. Вывод в текстовой форме дает возможность отформатировать данные таким способом, который будет приемлемым для конкретного приложения, например, разделяя значения запятыми или используя еще какой-то из множества текстовых форматов обмена данными, которые были разработаны еще до появления XML. И наконец, вывод в формате XML позволяет поставлять данные в приложения нового поколения, которые принимают непосредственно XML. Как правило, выходной файл будет использовать словарь тегов XML, отличающийся от словаря исходного документа: например XSLT-преобразование может представить исходные данные ежемесячных продаж в виде гистограммы, используя для этого основанный на XML стандарт для векторной графики SVG. Также можно использовать XSLT-преобразование с целью генерировать формат VoxML для звукового исполнения полученных данных.

> *Ссылки на информацию о языке разметки голоса VoxML (Voice Markup Language) фирмы Motorola можно найти по адресу http://www.oasis-open.org/cover/voxML.html.*

Давайте углубимся теперь в первую стадию, преобразование, – стадию, с которой, прежде всего, имеет дело XSLT и которая обеспечивает возможность получения на выходе разнообразных форматов данных. Эта стадия может включать выборку данных, их объединение и группирование, сортировку или выполнение арифметических действий, например перевод сантиметров в дюймы.

Итак, как же это происходит? До появления языка XSLT исходные XML-документы можно было обрабатывать только с помощью специальных приложений собственной разработки. От такого приложения не требовалось проведения структурного анализа необработанного XML-документа, но оно должно было через определенный интерфейс прикладного программирования (API, Application Programming Interface) вызвать синтаксический анализатор XML, чтобы получить из документа информацию и как-то обработать ее. Для этих целей существует два основных API: SAX (Simple API for

XML, простой API для XML) и DOM (Document Object Model, объектная модель документа).

SAX – это интерфейс, основанный на событиях, когда по мере анализа документа синтаксический анализатор уведомляет приложение о каждом отдельном фрагменте данных. Если же используется DOM, синтаксический анализатор считывает документ и формирует в памяти его древовидную объектную структуру. Можно затем написать собственное приложение (на одном из процедурных языков: C++, Visual Basic или Java), которое будет производить анализ полученной древовидной структуры. Это возможно, когда определена специальная **последовательность шагов**, которые нужно выполнить для получения требуемого формата выходного документа. Таким образом, независимо от типа используемого синтаксического анализатора, этот процесс всегда имеет один и тот же существенный недостаток: для каждого нового типа XML-документа приходится создавать новую специальную программу, описывая последовательность действий при обработке данного XML.

Оба API – DOM и SAX – подробно описаны в книге «Professional XML», Wrox Press, ISBN 1-861003-11-0.[1]

Почему же использование XSLT для выполнения преобразований XML лучше, чем написание вручную специальных приложений? Идея XSLT основана на том, что все эти приложения похожи друг на друга, и поэтому возможно запрограммировать то, что они делают, используя **описательный** язык высокого уровня, вместо того, чтобы создавать каждую программу от начала и до конца на языках C++, Visual Basic или Java. Нужное преобразование можно выразить через набор правил. Эти правила определяют, что должно генерироваться на выходе, когда в исходном документе встречается тот или иной специфический образец. Язык XSLT является описательным в том смысле, что он описывает требуемое преобразование, а не выдает последовательность инструкций для его выполнения. XSLT лишь описывает необходимое преобразование, а выбор наиболее эффективного способа его осуществления предоставляется XSLT-процессору.

XSLT все еще использует синтаксический анализатор XML – неважно, с каким интерфейсом, DOM или SAX, – чтобы привести XML-документ к древовидной структуре. XSLT манипулирует именно древовидным представлением XML-документа, а не самим документом. Для читателя, знакомого с интерфейсом DOM, представление структурных единиц XML-документа (элементов, атрибутов, инструкций обработки и т. д.) как узлов дерева будет уже привычным. XSLT – это высокоуровневый язык, который может выполнять перемещения по дереву, выбирать заданные узлы и производить с ними сложные манипуляции.

Концепции древовидных моделей XSLT и DOM схожи, но не идентичны. Модель, используемая XSLT, детально обсуждается в главе 2.

[1] Д. Мартин и др. «XML для профессионалов». – М.: Лори, 2001.

Из приведенной характеристики XSLT (описательный язык, который может отыскивать и выбирать конкретные данные, а затем производить с ними различные действия) напрашивается его подобие стандартному языку запросов к базам данных, SQL. Давайте остановимся на их сравнении.

XSLT и SQL: сходство

Лучше проанализировать сходство XSLT с реляционными базами данных. В них данные представлены в виде набора таблиц. Сами по себе таблицы не имеют большого значения – данные могли бы также сохраняться в обычных файлах с разделителями-запятыми. Главное достоинство реляционной базы данных не в структуре данных, а в языке, который обрабатывает эти данные, – SQL. Точно так же, XML лишь определяет структуру данных. Эта структура несколько богаче таблиц реляционной модели, но сама по себе фактически не несет ничего особо полезного. Только при наличии высокоуровневого языка, специально предназначенного для работы со структурированными данными, начинаешь понимать, что получено нечто интересное, а для данных XML таким языком является XSLT.

Внешне SQL и XSLT – очень разные языки. Но если взглянуть глубже, в них можно обнаружить много общего. Для начала возьмем хотя бы то, что для запуска обработки конкретных данных – будь они в реляционной базе данных или в XML-документе – язык обработки должен воспользоваться декларативным синтаксисом запросов для отбора данных, подлежащих этой обработке. В SQL – это оператор отбора `SELECT`. В XSLT эквивалентом являются **XPath-выражения**.

Язык выражений XPath является компонентом XSLT, несмотря на то, что он определен в отдельной рекомендации консорциума W3C (*http://www.w3.org/TR/XPath*), так как может использоваться и независимо от XSLT (взаимосвязь между XPath и XSLT обсуждается позже в этой главе в разделе «XSLT и XPath»).

Синтаксис запросов на языке XPath предназначен для отыскания узлов в XML-документе и основан на пути по XML-документу или контексте, в котором находится заданный узел. Это позволяет получить доступ к конкретным узлам, сохраняя исходную иерархию и структуру документа. Затем XSLT обрабатывает результаты этих запросов (перегруппировывая выбранные узлы, создавая новые узлы и т. д.).

XSLT и SQL имеют и другие общие черты:

- Оба языка усиливают стандартные возможности поиска полезными добавлениями для выполнения основных арифметических действий, обработки строк и операций сравнения.
- В обоих языках декларативный синтаксис запросов дополнен полупроцедурными средствами для описания последовательности обработки, которую требуется произвести. Кроме того, в них предусмотрена возможность обращения к традиционным языкам программирования, когда алгоритмы становятся слишком сложными.

- Оба языка обладают важным свойством, которое называется **замкнутостью**, и означает, что выходные данные имеют ту же самую структуру, что и входные. Для SQL эта структура – таблицы, для XSLT это – деревья, древовидное представление XML-документов. Свойство замкнутости чрезвычайно ценно, потому что оно означает, что действия, выполняемые с использованием языка, можно последовательно объединять, чтобы производить гораздо более сложные действия: просто выходные данные одной операции становятся входными для следующей операции. В SQL это можно делать, определяя представления или подзапросы; в XSLT – пропуская данные через ряд таблиц стилей или (по новой спецификации XSLT 1.1, разрабатываемой в настоящее время) фиксируя результат одной фазы преобразования как временное дерево и используя это временное дерево в качестве исходных данных для следующей фазы преобразования.

XSLT и SQL, безусловно, должны сосуществовать в реальном мире. Существует много возможных способов их взаимодействия, но самым типичным является хранение данных в реляционных базах и передача их между системами в формате XML. Эти два языка не сочетаются так удобно, как хотелось бы, так как они используют совершенно разные модели данных. Но преобразования, осуществляемые при помощи XSLT, могут сыграть важную роль соединительного мостика между ними. Ряд производителей СУБД выпустил программы, объединяющие XML и SQL, хотя пока еще в этой области нет никаких стандартов. Поинтересуйтесь на веб-сайтах поставщиков самыми последними версиями Microsoft SQL Server 2000 и Oracle 9i.

Прежде чем перейти к рассмотрению простого рабочего примера XSLT-преобразования, следует кратко обсудить несколько процессоров XSLT, которые способны произвести эти преобразования.

XSLT–процессоры

Основная роль XSLT-процессора – применить таблицу стилей XSLT к исходному документу XML и сформировать конечный документ. Важно отметить, что все компоненты этого процесса являются приложениями XML, так что базовая структура каждого – дерево. Следовательно, XSLT-процессор фактически обрабатывает три дерева.

Есть несколько XSLT-процессоров, из которых можно выбирать. Здесь будут упомянуты три из них: Saxon, Xalan и Microsoft MSXML3. Все они могут быть бесплатно загружены с соответствующих веб-сайтов (но следует ознакомиться с условиями лицензирования).

> *Эти три процессора, а также некоторые другие подробно описаны в приложениях к этой книге.*

Saxon – XSLT-процессор с открытыми исходными текстами, разработанный автором этой книги. Он является приложением Java и может запускаться прямо из командной строки, не требуя веб-сервера или броузера. Программа

Saxon преобразует XML-документ, скажем, в документ HTML, который затем может быть помещен на веб-сервере. В этом случае и броузер, и веб-сервер имеют дело только с преобразованным документом.

Для использования процессора Saxon в среде Windows (95/98/NT/2000) проще всего загрузить Instant Saxon, который выполнен как исполняемая программа Windows. Потребуется также установить Java, но это уже сделано, если в системе есть любая недавняя версия Internet Explorer. На платформах, отличных от Windows, нужно будет установить полный пакет Saxon и следовать сопроводительным инструкциям. Instant Saxon можно загрузить бесплатно с веб-страницы *http://users.iclway.co.uk/mhkay/saxon/index.html*. Saxon будет работать с любым синтаксическим анализатором XML, поддерживающим интерфейс SAX2 (в варианте Java), а кроме того, в комплект Saxon входит копия синтаксического анализатора Ælfred, так что его не придется устанавливать отдельно.

Xalan – другой открытый XSLT-процессор, доступный на веб-сайте организации Apache *http://xml.apache.org/*. Xalan произошел от программного продукта IBM под названием LotusXSL, но с тех пор он развивался как открытое программное средство и живет своей собственной жизнью. Xalan доступен в нескольких вариантах – C++ и Java. Подобно Saxon, процессор Xalan-Java является приложением Java, которое может выполняться прямо из командной строки. Xalan тоже может работать с любым синтаксическим анализатором, поддерживающим интерфейс SAX2; кроме того, он поставляется с копией собственного анализатора – Xerces.

Процессоры Saxon и Xalan-Java поддерживают один и тот же Java-интерфейс, называемый TrAX, который описан в приложении F. Это значит, что можно создавать программные продукты, которые будут работать с обоими процессорами. Оба они соответствуют спецификации XSLT 1.0, поэтому таблицы стилей будут полностью совместимы; но возможности вне требований стандарта у этих процессоров разные. Оба продукта более подробно описаны в приложениях этой книги.

В действительности с таблицами стилей XSLT может работать и Internet Explorer. Серийные варианты IE5 и IE5.5 снабжены встроенным процессором, который поддерживает диалект XSLT компании Microsoft, в некоторой степени соответствующий спецификации раннего рабочего проекта W3C 1998 года: этот язык (который здесь будет называться *рабочим проектом XSL* или *WD-xsl*) весьма отличается от окончательной версии, поэтому настоятельно рекомендуется избегать его. Сейчас Microsoft обеспечивает новую реализацию процессора – MSXML3, – полностью отвечающую утвержденной спецификации XSLT 1.0. Вероятно, она будет поставляться как стандарт с Internet Explorer 6[1], но уже сейчас ее можно загрузить отдельно с веб-сайта *http://msdn.microsoft.com* и использовать с IE5 или IE5.5.

[1] В настоящее время Internet Explorer 6.0 уже вышел. MSXML3 действительно поставляется вместе с ним. – *Примеч. ред.*

Нужно загрузить и установить оба пакета – SDK и пакет для конечных пользователей. Кроме того, необходимо загрузить и установить программу `xmlinst.exe`. Выполните эту программу, чтобы установить MSXML3 в качестве XML-процессора по умолчанию для Internet Explorer (если не сделать этого, IE5 будет использовать старый процессор WD-xsl). Возможность выполнения XSLT-преобразований непосредственно в броузере является большой удачей технологии Microsoft.[1]

В тексте книги автор постарался избегать обсуждения конкретных программ, так как подобная информация довольно быстро устаревает. Программные продукты, описанные в приложениях, в настоящее время вполне стабильны, но их круг постоянно пополняется новыми программами. Свежую информацию о состоянии дел в этой области лучше черпать из Интернета. Хорошими источниками для этого могут служить следующие сайты:

- *http://www.w3.org/Style/XSL*
- *http://www.xslinfo.com/*
- *http://www.xml.com/*
- *http://www.oasis-open.org/cover*

Пример таблицы стилей

Теперь можно рассмотреть пример использования XSLT для преобразования очень простого XML-документа.

Пример: Таблица стилей XSLT "Hello, world!"

Керниган и Ричи (Kernighan и Ritchie) в своей классической книге «The C Programming Language» («Язык программирования C») первыми продемонстрировали в самом начале книги простую, но полноценную программу, и с тех пор использование в качестве примера программы "Hello world" стало уважаемой традицией. Конечно, пока не определены еще многие концепции, невозможно детально описать, как работает этот пример, но не стоит волноваться – все непонятное будет объяснено позже.

Исходный документ

Какое же преобразование выполнить? Давайте попробуем преобразовать следующий XML-документ:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet type="text/xsl" href="hello.xsl"?>
<greeting>Hello, world!</greeting>
```

[1] Кроме Internet Explorer, XSLT также поддерживается броузером с открытыми исходными текстами Mozilla. Кроме того, у Mozilla есть несколько преимуществ перед Internet Explorer: он лучше поддерживает многие современные веб-стандарты, работает на большом количестве платформ, имеет открытые исходные тексты. – *Примеч. науч. ред.*

Простое представление этого документа в виде узлов дерева выглядит следующим образом:

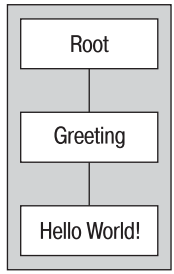


Рис. 1.2. *Представление структуры исходного документа в виде дерева*

В документе может существовать только один корневой узел. В модели XSLT корневой узел выполняет ту же функцию, что и узел структуры документа в DOM-модели. Объявление XML невидимо синтаксическому анализатору[1] и поэтому не включено в дерево.

Чтобы было понятнее, здесь в исходный файл XML специально включена инструкция обработки `<?xml-stylesheet?>`. Многие XSLT-процессоры используют ее в качестве таблицы стилей, если дополнительно не определена другая. Атрибут `href` указывает на относительный URI заданной по умолчанию таблицы стилей для этого документа.

Результат

В результате будет получен следующий HTML-документ, который просто изменит заголовок в окне броузера на строку "Today's greeting" и будет отображать любое приветствие, находящееся в исходном XML-файле:

```
<html>
<head>
   <title>Today's greeting</title>
</head>
<body>
   <p>Hello, world!</p>
</body>
</html>
```

Таблица стилей

Без лишних предисловий приведем таблицу стилей XSLT под названием `hello.xsl`, которая определяет характер преобразований:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

[1] Это утверждение не совсем верно. Если бы анализатор не видел xml-объявление, то он не мог бы правильно обрабатывать документ. А вот приложениям видеть его ни к чему, поэтому через интерфейс любого анализатора, будь то SAX или DOM, оно недоступно. – *Примеч. науч. ред.*

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <head>
      <title>Today's greeting</title>
   </head>
   <body>
      <p><xsl:value-of select="greeting"/></p>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

Запуск таблицы стилей

Можно запустить эту таблицу стилей с помощью любого из трех процессоров, описанных в предыдущем разделе.

Saxon

Ниже приведена последовательность действий при работе с процессором Saxon (для платформы Windows):

- Загрузите пакет Instant Saxon
- Поместите исполняемую программу `saxon.exe` в соответствующий каталог и сделайте его текущим каталогом
- Используя Notepad, создайте в этом каталоге два файла с приведенными выше текстами: `hello.xml` и `hello.xsl` соответственно (или загрузите их с веб-сайта Wrox *http://www.wrox.com*)
- Откройте сеанс MS DOS (Пуск | Программы | Сеанс MS DOS – в Windows 98, или смотрите в меню Стандартные в Windows NT или 2000)
- Наберите в командной строке:

```
saxon hello.xml hello.xsl
```

- Полюбуйтесь на полученный HTML, выведенный на экран.

Для просмотра выходных данных с помощью броузера просто сохраните вывод командной строки как HTML-файл:

```
saxon hello.xml hello.xsl > hello.html
```

Поскольку для этого исходного XML-документа использована заданная по умолчанию таблица стилей, можно также ввести такую команду:

```
saxon -a hello.xml > hello.html
```

Xalan-Java

При использовании процессора Xalan-Java нужно выполнить очень похожую процедуру. Однако в отличие от Saxon, нет специальной версии Xa-

lan для платформы Windows, поэтому придется выполнять приложение непосредственно в среде Java (то же самое относится и к процессору Saxon при использовании его полной версии).

В этом случае следует установить виртуальную машину Java. Xalan не поддерживает виртуальную машину от Microsoft, которая встроена в Internet Explorer (так как ему требуются возможности более свежих версий Java), поэтому придется сделать это отдельно. Можно порекомендовать для этих целей пакет JDK 1.3, разработанный Sun, который доступен по адресу *http://java.sun.com/j2se/1.3/*. Установка его достаточно проста, хотя файл для загрузки довольно объемен.

Получив Xalan-Java 2 в виде `.zip`-файла, распакуйте его в соответствующий каталог. В этом пакете есть два важных файла, которые должны быть доступны виртуальной машине Java. Это файлы `xalan.jar` (содержащий код XSLT-процессора) и `xerces.jar` (содержащий синтаксический анализатор XML), оба они находятся в подкаталоге `bin`. Чтобы Java VM была способна найти эти файлы, их следует указать в переменной окружения `CLASSPATH`. Можно задать значение этой переменной в файле `autoexec.bat` или в другом сценарии, который выполняется при запуске операционной системы; в Windows NT и Windows 2000 можно изменять значения системных переменных через Панель управления – значок Система (System в Control Panel). Кроме того, можно ввести путь к этим файлам как параметр команды, запускающей XSLT-процессор.

Удобно поместить все необходимые `.jar`-файлы, в каталог типа `c:\jars`, а затем установить `CLASSPATH` следующим образом:

```
SET CLASSPATH=.;c:\jars\xalan.jar;c:\jars\xerces.jar
```

Заметьте, что в переменной окружения `CLASSPATH` должны указываться конкретные `.jar`-файлы, а не каталоги, в которых они содержатся.

Для запуска преобразования "hello world" с помощью процессора Xalan следует ввести команду:

```
java org.apache.xalan.xslt.Process -in hello.xml -xsl hello.xsl
```

Это должно привести к тому же результату, что и при работе с Saxon. При использовании заданной по умолчанию таблицы стилей эту команду можно упростить:

```
java org.apache.xalan.xslt.Process -in hello.xml
```

При желании направить выходные данные в конкретный файл используйте опцию `-out`.

MSXML3

Наконец, можно использовать таблицы стилей в броузере Internet Explorer.

Установив MSXML3 в качестве используемого по умолчанию XSLT-процессора, нужно просто дважды щелкнуть кнопкой мыши по значку фай-

ла hello.xml, что запустит Internet Explorer и загрузит hello.xml в броузер. Он прочитает XML-файл, обнаружит, что необходимо применить таблицу стилей, загрузит ее, выполнит преобразования и отобразит результат в формате HTML. Если на экране виден только XML-файл, а не текст "Hello, world!", значит, система использует исходный WD-xsl интерпретатор Microsoft, встроенный в IE5, а не MSXML3. Если отображается таблица стилей, это также указывает на неправильное завершение процесса инсталляции. Не забудьте выполнить программу xmlinst.exe.

Как это работает

Если читатели успешно справились с этим примером или даже просто не потеряли интерес к этой книге, наверняка они желают знать, как это все работает. Давайте разберемся:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Это – просто стандартный заголовок XML. Следует отметить, что таблицы стилей XSLT являются самостоятельными XML-документами. Позже в этой главе о них будет рассказано подробнее. Здесь используется кодировка iso-8859-1 (это официальное название набора символов, который Microsoft иногда называет «ANSI»), так как в Западной Европе и Северной Америке этот набор символов поддерживает большинство текстовых редакторов. Однако можно работать и с другими кодировками, которые поддерживают ваши текстовые редакторы, например набором символов UTF-8.

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Это – стандартный заголовок XSLT. В терминологии XML – это тег начала элемента, и он означает, что данный документ является таблицей стилей. Атрибут xmlns:xsl – XML-объявление пространства имен, которое указывает, что для элементов, определенных в спецификации XSLT консорциума W3C, будет использоваться префикс xsl. В XSLT часто используется пространство имен XML, и все имена элементов, определенных в стандарте, будут отмечены этим префиксом, чтобы избежать путаницы с именами, использованными в исходном документе. Атрибут version указывает, что таблица стилей использует только возможности версии 1.0 стандарта XSLT.

Анализируем дальше:

```
<xsl:template match="/">
```

Элемент <xsl:template> определяет шаблонное правило, которое будет инициировано при обработке определенной области исходного документа. Атрибут match="/" указывает, что это конкретное правило запускается в самом начале разбора документа. Здесь «/» является выражением XPath, которое задает **корневой узел** документа: XML-документ имеет

иерархическое строение и, так же как в UNIX, специальное имя файла «/» обозначает корневой каталог файловой системы, XPath использует «/» для представления корневого узла структуры XML-содержимого. В DOM-модели это называется объектом Document, но в языке XPath это называется корнем.

```
<html>
<head>
   <title>Today's greeting</title>
</head>
<body>
   <p><xsl:value-of select="greeting"/></p>
</body>
</html>
```

Как только инициируется данное правило, содержание шаблона определяет, что должно генерироваться на выходе. Здесь тело шаблона представлено, в основном, последовательностью элементов разметки HTML и текста, которые будут скопированы в выходной файл. Единственное исключение – элемент `<xsl:value-of>`, который является инструкцией XSLT, так как использует префикс пространства имен `xsl`. Эта конкретная инструкция требует копировать значение узла структуры исходного документа в выходной документ. Атрибут `select` задает узел, значение которого должно быть определено. Выражение XPath «`"greeting"`» означает: найти все элементы `<greeting>`, являющиеся непосредственными потомками узла, который обрабатывается в настоящее время этим шаблонным правилом. В данном случае нужно найти ближайший в исходном документе элемент «`greeting`». Инструкция `<xsl:value-of>` требует извлечь текстовый узел этого элемента и скопировать его в надлежащее место выходного документа, другими словами, внутрь формируемого элемента `<p>`.

Остается только завершить работу тем, с чего начинали:

```
</xsl:template>
</xsl:stylesheet>
```

На самом деле в простой таблице стилей, подобной продемонстрированной выше, можно опустить некоторые формальности. Так как имеется только одно шаблонное правило, элемент `<xsl:template>` можно не указывать. Ниже показана полноценная таблица стилей, эквивалентная предшествующей:

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<head>
   <title>Today's greeting</title>
</head>
<body>
   <p><xsl:value-of select="greeting"/></p>
</body>
</html>
```

Задача упрощенного синтаксиса – облегчить восприятие XSLT для людей, знакомых с каким-либо языком шаблонов, который позволяет сформировать каркас HTML-страницы с помощью специальных тегов (аналогичных `<xsl:value-of>`) для вставки нужных данных в соответствующие позиции. Но, как будет показано далее, язык XSLT гораздо мощнее подобных языков.

Для чего может понадобиться помещать приветствие в отдельный XML-файл и отображать его с помощью таблицы стилей? Одна из возможных причин – желание отображать приветствие различными способами в зависимости от контекста: например, его можно было бы показывать по-разному на разных устройствах. В этом случае можно создать другую таблицу стилей, чтобы по-другому преобразовывать тот же самый исходный документ. Сразу возникает вопрос: как же при анализе документа происходит выбор таблицы стилей? На этот вопрос нет однозначного ответа. Как было показано ранее, интерфейсы процессоров Saxon и Xalan позволяют указать и таблицу стилей, и исходный документ, которые нужно использовать. Того же самого можно добиться и в случае XSLT-процессора от Microsoft, хотя это требует предварительного создания HTML-страницы, содержащей специальный сценарий для управления преобразованием. Использованная в примере инструкция обработки `<?xml-stylesheet?>` работает только с одной и той же таблицей стилей.

А теперь пришло время взглянуть поближе на связь между XSLT и языком XPath, а также другими XML-технологиями.

Положение XSLT в технологиях XML

Язык XSLT выпущен консорциумом World Wide Web (W3C) и согласован со стандартами технологий XML, большинство из которых также разработано W3C. В этом разделе будет предпринята попытка раскрыть связь XSLT, порой весьма запутанную, с другими стандартами и спецификациями.

XSLT и XSL

Язык XSLT возник как часть языка более высокого уровня – **XSL (Extensible Stylesheet Language, расширяемый язык таблиц стилей)**. Как можно понять из названия, задачей XSL было (и есть) описание форматирования и отображения XML-документов на экране, на бумаге или с помощью синтезаторов речи. По мере развития XSL стало ясно, что процесс его работы, как правило, двухстадийный: сначала выполняется структурное преобразование, в котором элементы выбираются, группируются и упорядочиваются, а затем происходит форматирование, при котором полученные элементы отображаются в виде печатаемых на бумаге знаков или отображаемых на экране пикселей. Очевидно, что эти две стадии довольно независимы, поэтому язык XSL был разбит на две части: описание преобразований, или сам язык XSLT, и «остальное» для стадии форматирования – официально это все еще язык XSL, хотя некоторые предпочитают называть его **Форматирующими объектами XSL**, или **XSL-FO (XSL Formatting Objects)**.

Форматирующие объекты XSL являются обычным XML-словарем, в котором описываемыми объектами являются области печатаемой страницы и их особенности. Поскольку это – всего лишь другой XML-словарь, в XSLT не нужны специальные возможности для генерирования его в качестве конечного результата. Форматирующие объекты XSL не являются предметом обсуждения этой книги. Это – отдельная большая область (доступные сейчас рабочие варианты ее спецификации гораздо длиннее спецификации XSLT), окончательно этот язык еще не стандартизирован, а программные продукты, реализующие его, еще будут дорабатываться. Можно добавить также, что большинству читателей книги этот язык понадобится с гораздо меньшей вероятностью, чем XSLT. Форматирующие объекты XSL являются замечательным средством достижения высококачественного типографского представления документов. Однако для большинства людей, преобразующих документы в HTML-формат для отображения их в окне стандартного броузера, это излишняя роскошь, и свои задачи они могут решить с помощью только XSLT или, в случае необходимости, используя XSLT вместе с каскадными таблицами стилей CSS или CSS2 (Cascading Style Sheets), о которых будет рассказано чуть позже.

> *Спецификации форматирующих объектов XSL, которые на момент написания книги имеют статус кандидатов в рекомендации (Candidate Recommendation), можно найти на веб-странице http://www.w3.org/TR/xsl. Статус кандидатов в рекомендации дается спецификациям, относительно которых в рабочей группе W3C достигнуто согласие, достаточное для представления их на рассмотрение общественности. Условием получения спецификациями форматирующих объектов XSL статуса окончательных рекомендаций является проверка на опыте, что все конструкции языка успешно реализованы и что различные реализации могут взаимодействовать.*

XSLT и XPath

В ходе разработки XSLT обнаружилось существенное пересечение синтаксиса выражений для выбора областей документа в языке XSLT и языке XPointer, который служит для связывания одного документа с другим. Чтобы избежать наличия двух отдельных, но пересекающихся языков выражений, два комитета решили объединить усилия и определить единый язык, **XPath**, который решал бы обе задачи. Версия 1.0 языка выражений XPath была опубликована в один день с публикацией XSLT, 16 ноября 1999 года.

XPath функционирует как вспомогательный язык в рамках таблицы стилей XSLT. XPath-выражениями можно воспользоваться для числовых расчетов, обработки строк или для проверки логических условий, но его наиболее типичным применением (которому он и обязан своим названием) является задание областей входного документа, подлежащих обработке. Например, следующая инструкция выводит среднюю цену всех книг во входном документе:

```
<xsl:value-of select="sum(//book/@price) div count(//book)"/>
```

Здесь элемент <xsl:value-of> является инструкцией, определенной в стандарте XSLT, которая требует вывести данное значение в выходной документ. Атрибут select содержит выражение XPath, которое вычисляет требуемое значение: в данном случае сумму атрибутов price всех элементов <book>, разделенную на число элементов <book>.

Спецификация XPath с возрастающими темпами развивается самостоятельно, отдельно от XSLT. Например, несколько реализаций DOM (включая реализацию от Microsoft) позволяют выбирать узлы в древовидной структуре моделей DOM, используя метод selectNodes(XPath), и предполагается, что эта особенность будет предусмотрена в следующей версии стандарта, DOM3. Кроме того, XPath вводится и в другие спецификации консорциума W3C, включая стандарты для XPointer и XQuery.

Отделение языка XPath от XSLT – в целом положительное явление, но в некоторых ситуациях это разбиение вызывает затруднения, причем трудно решить, где можно найти ответ на возникшие вопросы. Например, выражение XPath может содержать ссылку на переменную, но создание этой переменной и указание ее начального значения – задача XSLT. Другой пример: выражения XPath могут вызывать функции, и существует ряд функций, определенных в стандартах. Те функции, действия которых совершенно автономны, например функция string-length(), определены в спецификации XPath, в то же время дополнительные функции, поведение которых зависит от определений XSLT, например функция key(), описаны в спецификации XSLT.

Поскольку разбиение вносит неудобства, условимся в рамках этой книги считать XSLT+XPath единым языком. Поэтому, например, все стандартные функции описаны вместе в главе 7. В разделах со ссылками, по возможности, указано, в каких первоначальных стандартах определена каждая функция или иная конструкция, но в остальном договоримся, что речь идет об обоих языках вместе и неважно, где кончается использование одного и начинается использование другого. Единственный недостаток такого подхода в том, что при необходимости работы с XPath как с независимым языком, например, при применении метода selectNodes() в модели DOM, придется свериться с описанием функций в главе 7, чтобы определить, является ли нужная функция базовым компонентом XPath или это добавление XSLT.

XSLT и Internet Explorer 5

Еще в 1998 году, вскоре после опубликования первого рабочего проекта языка XSL, компания Microsoft выпустила его пробную реализацию для использования с IE4. Впоследствии, когда вышла новая версия броузера, IE5, пробная реализация была существенно изменена. Эта вторая реализация, известная как MSXSL, оставалась, по существу, неизменной до совсем недавнего времени, и все еще поставляется с каждой копией IE5 и IE5.5, а также с Windows 2000. К сожалению, однако, Microsoft несколько опередил события, а стандарт XSLT развивался и изменился до такой степени, что когда 16 ноября 1999 года была опубликована версия 1.0 XSLT в статусе рекомендации, она очень отдаленно соответствовала первоначальному продукту Microsoft.

Рекомендация является наиболее авторитетным из документов, выпускаемых консорциумом W3C. Формально это еще не стандарт, так как стандарты издаются только после утверждения государственными органами стандартизации. Но в этой книге мы будем рассматривать рекомендации как фактические стандарты.

Многие изменения, например в ключевых словах, не очень существенны, но некоторые довольно глубоки, в частности изменения в способе определения оператора равенства.

По этой причине диалект XSL, который встроен в Microsoft IE5 и который в этой книге упоминается под названием WD-xsl, тоже не является здесь предметом обсуждения. Пожалуйста, не считайте, что в этой книге что-либо соответствует первоначальной реализации Microsoft XSL, поскольку даже в тех случаях, когда синтаксис кажется подобным синтаксису XSLT, значение конструкции может быть совершенно иным.

Информацию о WD-xsl можно найти в книге издательства Wrox Press «XML IE5 Programmer's Reference» (Справочник программиста по XML IE5), ISBN 1-861001-57-6.

Как уже говорилось, XSLT-процессор MSXML3 от Microsoft полностью отвечает спецификации XSLT 1.0, но на момент написания книги его требуется специально загрузить с веб-сайта Microsoft и установить. В действительности MSXML3 поддерживает и старый диалект WD-xsl, и более новый язык XSLT 1.0, определяя, какой из них используется, по идентификатору URI пространства имен, соответствующему префиксу элемента `<xsl:stylesheet>`.

Microsoft выпустил также конвертор для обновления таблиц стилей WD-xsl до XSLT 1.0. Однако это еще не решает всех проблем, так как остаются миллионы установленных копий IE5, которые поддерживают только старую версию. Если нужно создать веб-сайт, на котором задача преобразования XML-документов будет возложена на броузер и его интерпретацию таблиц стилей XSLT, то необходимо добиться того, чтобы все пользователи смогли иметь к нему доступ.[1] Для более удобной загрузки MSXML3 на веб-сайте Microsoft выложен инсталлятор в виде CAB-файла, – подробности можно узнать на сайте MSDN.

Для тех, кто использует технологию Microsoft на сервере, имеется расширение ISAPI под названием XSLISAPI. Это расширение позволяет производить преобразования в броузерах, которые его поддерживают, или на сервере, если броузер не поддерживает XSLT. Однако если у вас нет возможности управлять конфигурацией броузеров пользователей, то единственной практической возможностью для серьезного проекта в настоящее время является преобразование **на стороне сервера** XML в HTML с помощью ASP-страниц или Java-сервлетов.

---

[1] В настоящее время создание таких сайтов приемлемо только в сетях интранет, так как существует значительное количество броузеров, пока еще не поддерживающих XSLT, и такой сайт окажется для них совершенно недоступным. – *Примеч. науч. ред.*

В приложении А более подробно рассказано об XML-продуктах Microsoft, но не забывайте, что подобная информация очень быстро устаревает.

XSLT и XML

По существу, XSLT – это инструмент для преобразования XML-документов. В начале этой главы обсуждалось, почему это важно, а теперь следует более внимательно рассмотреть взаимосвязь между обоими языками. В языке XML есть два основных аспекта, с которыми очень тесно связан XSLT: это пространства имен XML и информационное множество XML (XML Information Set). Они обсуждаются в следующих разделах.

Пространства имен XML

При разработке XSLT учитывалось, что **пространства имен XML (XML namespaces)** являются неотъемлемой частью стандарта XML. Поэтому, когда XSLT имеет дело с XML-документами, подразумевается, что это XML-документы, отвечающие спецификации XML Namespaces, которую можно найти по адресу *http://www.w3.org/TR/REC-XML-names*.

> *Детальное обсуждение пространства имен XML приведено в главе 7 книги издательства Wrox Press «Professional XML» («XML для профессионалов»), ISBN 1-861003-11-0.*

Пространства имен играют важную роль в языке XSLT. Их основная задача – обеспечить возможность смешивать в одном и том же XML-документе теги из нескольких различных словарей разметки. Например, в одном словаре элемент `<table>` может означать двумерный массив данных, в то время как в другом словаре под элементом `<table>` подразумевается предмет мебели. Напомним вкратце особенности пространства имен:

- Пространства имен задаются унифицированным идентификатором ресурса URI (Uniform Resource Identifier). Этот идентификатор может быть представлен в различных формах. Одна из них – привычный URL, например `http://www.wrox.com/namespace`. Другая форма, не стандартизированная окончательно, но используемая в некоторых XML-словарях (см., например, *http://www.biztalk.org*) – URN, например `urn:java:com.icl.saxon`. Конкретная форма записи URI не имеет значения, но стоит выбирать URI, который будет однозначно определять пространство имен. Хорошим вариантом может быть использование URL вашего собственного веб-сайта. Однако не думайте, что на веб-сайте для этого должен быть организован какой-то специальный ресурс, на URL которого можно ссылаться. URI пространства имен – просто последовательность символов, выбранных вами, чтобы различать URI разных пространств имен; он не должен обязательно указывать на что-то реальное.
- Поскольку URI пространств имен часто довольно длинные и используют специальные символы типа «/», их не приводят полностью в именах элементов и атрибутов. Вместо этого каждому пространству имен, используемому в документе, можно давать короткие псевдонимы, употребляя их с

именами атрибутов и элементов в виде префиксов. Неважно, какой будет выбран префикс, потому что реальное имя элемента или атрибута определяется только по URI его пространства имен и по его локальному имени (по части имени, идущей после префикса). Например, в этой книге во всех примерах используется префикс `xsl`, являющийся псевдонимом URI пространства имен `http://www.w3.org/1999/XSL/Transform`, но с тем же успехом можно было бы использовать и префикс `xslt`; главное здесь – последовательность и однозначность.

- Существует также возможность указать пространство имен по умолчанию для элементов, которое будет связано с элементами без префикса. Однако на атрибуты это не распространяется.

Префикс пространства имен объявляется в рамках любого тега элемента с помощью специального псевдоатрибута:

```
xmlns:prefix = "namespace-URI"
```

Таким способом объявляется префикс пространства имен, который может использоваться в имени этого элемента, его атрибутов и в имени любого элемента или атрибута, содержащихся в данном элементе. Значение по умолчанию для пространства имен, связываемого с элементами, в именах которых префикс не указан явно (но не с атрибутами), назначается аналогично с помощью псевдоатрибута:

```
xmlns = "namespace-URI"
```

XSLT нельзя использовать для обработки XML-документов, не соответствующих рекомендации XML Namespaces. На практике это не проблема, поскольку большинство людей считает XML Namespaces неотъемлемой частью стандарта XML, а не дополнительной добавкой к нему. Хотя какую-то роль это все же играет. В частности, активное использование пространств имен фактически несовместимо с серьезным использованием определений типа документа DTD (Document Type Definitions), поскольку DTD не придают особого значения префиксам в именах элементов. Таким образом, поддержка пространств имен ограничивает для XSLT использование DTD, и вместо этого приходится ждать появления средства, способного заменить DTD. Это средство – XML-схемы (XML Schemas).

Информационное множество XML

Язык XSLT предназначен для работы с информацией, которая заложена в XML-документе, а не с самим документом непосредственно. Это означает, что XSLT-программист работает с древовидным представлением структуры исходного документа, в котором некоторые аспекты являются видимыми, а некоторые – нет. Например, можно видеть имена и значения атрибутов, но не видно, как записаны эти атрибуты – в одинарных или двойных кавычках, в каком порядке, на одной и той же строке или нет.

Неприятным моментом остается то, что предпринималось уже немало попыток точно определить, что именно составляет **существенную** информацию

правильно построенного XML-документа, а что служит лишь несущественной пунктуацией. До сих пор однозначных критериев нет. Самой свежей и наиболее конструктивной попыткой создания общего словаря содержимого XML-документов является определение **информационного множества XML (XML Information Set** definition), которое обычно называют "infoset". Это определение можно найти по адресу *http://www.w3.org/TR/xml-infoset*.

К сожалению, это определение появилось слишком поздно, чтобы быть учтенным во всех стандартах. В результате одни считают комментарии существенным компонентом информации, другие – нет; некоторые придают выбору префиксов для пространства имен большое значение, а другие считают их несущественными. В главе 2 будет подробно рассказано, как XSLT (или точнее XPath) определяет модель дерева XML и в чем ее детальное отличие от некоторых других моделей, например от объектной модели документа DOM.

В последнее время все чаще можно услышать формулировку **информационное множество** после проверки соответствия схеме (PSVI, **post-schema-validation infoset**). В нем находится существенная информация исходного документа, дополненная информацией из его XML-схемы. Это позволяет выяснить не только само значение атрибута (например, «17.3»), но также и то, что этот атрибут описан в схеме как неотрицательное десятичное число. Пока еще XSLT не может оперировать с подобной информацией, но такая возможность учтена в опубликованном списке требований к следующей версии спецификации языка, XSLT 2.0.

XSL и CSS

Почему существует одновременно два языка таблиц стилей: XSL (точнее XSLT плюс XSL-FO) и каскадные таблицы стилей CSS и CSS2 (Cascading Style Sheets)?

Честно говоря, только в идеальном мире можно надеяться иметь для этих целей единственный язык, а причиной наличия сразу двух является то, что никому не удалось изобрести нечто, сочетающее простоту и экономичность CSS для выполнения простых вещей с мощностью XSL для выполнения более сложных задач.

Язык CSS (здесь подразумевается и CSS2, который значительно расширяет степень контроля над окончательным видом страницы), главным образом, используется для визуализации HTML, но его можно применить и для визуализации непосредственно XML, определив характеристики отображения каждого элемента XML. Однако этот язык имеет серьезные ограничения. Он не может переупорядочивать элементы в исходном документе, не может добавлять текст или изображения, не может решать, какие элементы должны быть отображены, а какие опущены, не может вычислять суммы, средние значения или порядковые номера. Другими словами, этот язык можно использовать только в тех случаях, когда структура исходного документа очень близка к окончательно отображаемой форме.

С другой стороны, язык CSS прост и очень экономичен в отношении используемых машинных ресурсов. Он не переупорядочивает документ, следовательно, ему не нужно формировать в памяти древовидное представление документа, и поэтому он может начать отображение документа, как только получит по сети начало текста. Пожалуй, самое важное его преимущество – его удивительная простота для авторов страниц HTML, совсем не имеющих навыков программирования. По сравнению с CSS, язык XSLT гораздо более мощен, но он потребляет и намного больше памяти и ресурсов процессора, а кроме того, требует расходов на обучение.

Иногда удобно применять оба инструментальных средства вместе. Создайте с помощью XSLT представление документа, близкое к его окончательной форме, расположив все тексты в правильном порядке, а затем воспользуйтесь CSS, чтобы добавить последние штрихи: цвета и размеры шрифтов и так далее. Типичный подход (на сегодняшний день) – работать с XSLT на сервере, а с CSS – на клиенте (прямо в броузере), дополнительное преимущество этого подхода заключается в уменьшении объема данных, которые требуется передавать по сети, что улучшает время отклика для пользователей и, возможно, отсрочит необходимость расходов на очередное увеличение пропускной способности сети.

История развития XSL

Подобно большинству стандартов семейства XML, спецификация языка XSLT была разработана Консорциумом World Wide Web (W3C) – коалицией компаний, которая была организована Тимом Бернерс-Ли (Tim Berners-Lee), изобретателем WWW. Истории создания XSL, а также проектам языков стилей в целом посвящена интересная веб-страница *http://www.w3.org/Style/History/*.

Предыстория

Первоначально Бернерс-Ли (Berners-Lee) замысливал язык HTML как набор тегов для разметки логической структуры документа: заголовков, абзацев, ссылок, цитат, фрагментов кода и т. п. Очень скоро людям захотелось иметь больше контроля над визуальным представлением документа, им хотелось так же управлять внешним видом получаемой публикации, как это возможно при печати на бумаге. В силу этого в язык HTML добавлялось все больше тегов и атрибутов, регулирующих визуальное представление документа: шрифты, поля, таблицы, цвета и другие оформительские детали. По мере развития этого процесса публикуемые документы становились все более зависимыми от программы просмотра, и оказалось, что первоначальные цели простоты и универсальности ушли на задний план.

Возникла насущная потребность разделить информацию и ее отображение. Это не новая концепция; она была хорошо разработана в восьмидесятые годы при развитии **SGML (Standard Generalized Markup Language, Стандартный язык обобщенной разметки)**, на архитектуру которого повлиял в свою

очередь детально разработанный (но так и не реализованный) проект открытой архитектуры систем документооборота ODA (Open Document Architecture), выполненный организацией ISO.

Аналогично возникновению XML как значительно упрощенному подмножеству языка SGML, язык XSLT также обязан своей природой SGML-основанному стандарту под названием **DSSSL (Document Style Semantics and Specification Language, Язык семантики и спецификации стиля)**. Язык DSSSL (я произношу это как «Dissel») был предназначен, прежде всего, для удовлетворения потребности в стандартном, не зависящем от аппаратных устройств языке, который может определять визуальное представление документов SGML, особенно для высококачественной типографской печати. DSSSL появился в начале 1990-х годов, а до той поры в течение длительного времени использовался только SGML, а для оформления документов существовали патентованные и часто чрезвычайно дорогие инструментальные средства, нуждающиеся также в не менее дорогих фотонаборных устройствах, так что эта технология была доступна только очень крупным издательствам.

На конференции WWW'94 в Чикаго С.М. Сперберг-МакКуин (C.M. Sperberg-McQueen) и Роберт Ф. Гольдстейн (Robert F. Goldstein) выступили с очень важным докладом под названием «Манифест о привнесении в Сеть логики *SGML*» («A Manifesto for Adding SGML Intelligence to the World-Wide Web»). Можно найти этот материал по адресу *http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Autools/sperberg-mcqueen/sperberg.html*.

Авторы представили набор требований к языку таблиц стилей, очень хорошо обобщивший все задачи, которые должны были решить проектировщики XSL. Как и в других проектах того времени, концепция отдельного языка преобразований еще не возникла, и большая часть публикации была посвящена возможностям языка в области визуализации документов. Возникло много конструктивных идей, включая концепцию обработки отказа (fallback processing), чтобы справляться с ситуациями, когда требуемые возможности не доступны в текущей среде.

Стоит процитировать здесь некоторые выдержки из доклада:

> *В идеале язык таблиц стилей должен быть описательным, не процедурным, и он должен позволить таблицам стилей максимально использовать структуру документов SGML. Стили должны меняться в зависимости от локализации элемента в структуре: абзацы примечаний должны форматироваться иначе, чем абзацы основного текста. Стили должны меняться при изменении значений атрибутов рассматриваемого элемента: цитирование типа «display» должно форматироваться иначе, чем «inline». В некоторых случаях они должны меняться даже при смене значений атрибутов у разных элементов: пункты в нумерованных списках должны выглядеть иначе, чем пункты в маркированных списках.*
>
> *В то же время, язык должен достаточно легко интерпретироваться процедурным способом: реализация языка таблиц стилей не должна стать главной трудностью при реализации веб-клиента.*
>
> *Семантика языка должна быть аддитивной: пользователи должны иметь возможность создавать таблицы стилей, добавляя новые детали к какой-либо уже су-*

ществующей (возможно, эталонной) таблице стилей без копирования всей эталонной таблицы; вместо этого пользователь должен иметь возможность сохранять локально только собственные изменения в эталонной таблице стилей, и они должны добавляться во время просмотра. Особенно важно поддерживать локальные изменения стандартных определений DTD.

Синтаксически язык таблиц стилей должен быть очень прост, предпочтительно тривиален для анализа. Напрашивается очевидная возможность: разрабатывать язык таблиц стилей как SGML DTD, таким образом, чтобы каждая таблица стилей была SGML-документом. Так как броузер уже способен анализировать SGML, не потребуется никаких дополнительных усилий в этом плане.

Для создания таблиц стилей и использования их в Сети мы настоятельно рекомендуем работать с подмножеством языка DSSSL. С завершением стандартизации DSSSL совершенно не имеет смысла изобретать с самого начала альтернативные языки таблиц стилей. Возможно, стандарт DSSSL слишком труден, чтобы реализовывать его во всей полноте, но даже если и так, это служит только дополнительным аргументом в пользу выделения подмножеств DSSSL, которые стоит поддерживать, а не в пользу создания собственных языков. В отличие от примитивных доморощенных спецификаций, подмножество стандартов имеет продуманные и предопределенные пути развития. Мы предполагаем работать над созданием удобных, реализуемых подмножеств DSSSL для использования в таблицах стилей для WWW и приглашаем всех заинтересованных присоединиться к нашим усилиям.

В конце 1995 года в Париже был проведен спонсируемый консорциумом W3C семинар по языкам таблиц стилей. Принимая во внимание последующую роль Джеймса Кларка (James Clark) как редактора рекомендации XSLT, интересно прочитать его заметки, свидетельствующие о его вкладе в области DSSSL. Эти материалы можно найти по адресу *http://www.w3.org/Style/951106_Workshop/report1.html#clark*.

Ниже приведены некоторые выдержки из этих заметок:

DSSSL содержит в себе как язык преобразований, так и язык форматирования. Первоначально преобразования были необходимы, чтобы сделать возможными некоторые виды стилей (такие как оглавление документа). Сейчас это стало заботой языка запросов, но язык преобразований выживет, потому что он нужен и для других целей.

Сохраняется необходимость как в простом, так и в сложном дизайне, а также в стилях, подходящих и для пакетного форматирования, и для интерактивных приложений. Существующие системы должны быть способны поддерживать DSSSL, возможно, лишь с минимальными изменениями (очевидно, необходим синтаксический анализатор DSSSL).

Язык должен быть строго декларативным, чего можно достичь, используя функциональное подмножество языка Scheme. Должны быть доступны интерактивные редакторы таблиц стилей.

Таблица стилей DSSSL очень точно описывает функцию, отображающую SGML в дерево потоковых объектов (flow object tree). Это позволяет объединять отдельные стилевые таблицы ('каскадировать' как в CSS): определенное правило может отменять некоторое другое правило, на основании явных и неявных приоритетов, но не существует никакого смешения конфликтующих стилей.

Джеймс Кларк закончил свою речь следующим замечанием:

> *Создание хорошего, расширяемого языка стилей – дело трудное!*

Есть подозрение, что редактирование рекомендации XSLT не заставило его изменить свое мнение.

Первый проект XSL

Прослеживая эти ранние дискуссии, консорциум W3C выдвинул официальную инициативу создать язык таблиц стилей. Обусловленным требованием к группе разработчиков было создание языка на основе DSSSL.

Результатом этой инициативы стало первое официальное предложение XSL, датированное 21 августа 1997 года. Его можно найти по адресу *http://www.w3.org/TR/NOTE-XSL.html*.

Там перечислены одиннадцать авторов. В их число вошло пять человек из Microsoft, три из Inso Corporation, а кроме того, Пауль Гроссо (Paul Grosso) из ArborText, Джеймс Кларк (который представлял самого себя) и Генри Томпсон (Henry Thompson) из эдинбургского университета.

Стоит почитать раздел, описывающий цель создаваемого языка:

> *XSL – это язык таблиц стилей, разработанный для Веб-сообщества. Он предоставляет возможности, не доступные CSS (например, переупорядочение элементов). Мы предполагаем, что CSS сохранит свое значение для отображения простых по структуре XML-документов, а XSL будет использоваться в тех случаях, когда требуются мощные возможности форматирования или для форматирования высоко структурированной информации, например структурированных данных XML или XML-документов, которые содержат структурированные данные.*
>
> *Авторы разработок для Веб создают содержимое трех различных уровней сложности:*
>
> - *разметка: опирается исключительно на декларативный синтаксис*
> - *сценарий: дополнительно использует фрагменты кода для более сложного поведения*
> - *программа: использует полноценный язык программирования*
>
> *Предусмотрена доступность XSL для пользователей уровня «разметки», поскольку обеспечено декларативное решение для большинства задач описания данных и их визуализации. Менее типичные задачи решаются изящным переходом к привычной среде создания сценариев. Такой подход знаком издательскому сообществу Сети, так как он смоделирован аналогично среде HTML/JavaScript.*
>
> *Мощные возможности, обеспечиваемые XSL, позволяют:*
>
> - *форматирование исходных элементов на основании иерархических отношений родитель/потомок, расположения и уникальности*
> - *создание конструкций форматирования, включая генерирование текста и графики*
> - *описание макрокоманд форматирования для повторного использования*
> - *таблицы стилей, независимые от направления текста в документе*
> - *расширяемый набор форматирующих объектов*

Авторы подробно объяснили, почему они почувствовали необходимость отойти от стандарта DSSSL и почему склонились к идее о создании отдельного от каскадных таблиц стилей CSS языка.

Затем они сформулировали некоторые принципы проектирования:

- *XSL должен быть пригоден для использования непосредственно через Интернет.*
- *XSL должен быть выражен в синтаксисе XML.*
- *XSL должен предоставить декларативный язык, позволяющий выполнять все стандартные задачи форматирования.*
- *XSL должен обеспечивать интеграцию с языком сценариев для выполнения более сложных задач форматирования, а также для расширяемости и завершенности.*
- *XSL будет являться подмножеством DSSSL с предложенной поправкой. Поскольку XSL уже не был подмножеством DSSSL, авторы хитроумно предложили отредактировать DSSSL, чтобы он стал надмножеством XSL.*
- *Должна существовать возможность автоматического преобразования таблицы стилей CSS в таблицу стилей XSL.*
- *XSL должен быть знаком пользователю языка таблиц стилей FOSI.*
- *Число необязательных опций в XSL должно сводиться к минимуму.*
- *Таблицы стилей XSL должны быть вполне понятными для человека и четкими.*
- *Проектирование XSL должно быть выполнено быстро.*
- *Таблицы стилей XSL должно быть легко создавать.*
- *Лаконичность разметки XSL не представляет особой важности.*

Как формулировка требований этот документ выполнен плохо. Он не читается как простой перечень потребностей, наподобие тех, которые можно получить, выясняя, в чем нуждаются пользователи. Это гораздо больше походит на список, который пишут проектировщики, когда они знают, что они хотят создать, и включают также несколько политических уступок людям, которые могли бы выдвинуть возражения. Но для тех, кто желает понять, почему XSLT стал таким языком, каким он стал, этот список – безусловное свидетельство серьезных размышлений.

Язык, описанный в этом первом проекте, содержит многие ключевые концепции XSLT в его сегодняшнем варианте, но синтаксис фактически неузнаваем. Было уже ясно, что язык должен быть основан на шаблонах, которые будут обрабатывать узлы исходного документа, соответствующие определенному образцу, и что язык должен быть свободен от побочных эффектов, чтобы сделать возможным «постепенное отображение и обработку больших документов». Значение этого требования будет более подробно обсуждаться в разделе «Никаких побочных эффектов» этой главы, а его отношение к проектированию таблиц стилей – в главе 9. Основная идея состоит в том, что, если таблица стилей выражена как набор полностью независимых действий, каждое из которых не имеет никаких внешних проявлений, кроме формирования фрагмента выходного документа из исходного (к примеру, оно не может модифицировать глобальные переменные), тогда становится возможным независимое генерирование любого фрагмента вывода, если изменился

соответствующий ему исходный фрагмент. Вопрос, достигает ли язык XSLT этой цели в действительности, все еще остается открытым.

В январе 1998 года, через пять месяцев после появления этого проекта, компания Microsoft выпустила свою первую предварительную версию.

Чтобы дать консорциуму W3C возможность оценить проект, Норман Волш (Norman Walsh) подготовил перечень требований, который был опубликован в мае 1998 года. Он доступен по адресу *http://www.w3.org/TR/WD-XSLReq*.

Большая часть этого документа сводилась к длинному списку типографских функций, которые должен поддерживать язык, – это дань традиции, и прежней, и последующей, по которой аспектам форматирования в языке отводится гораздо больше строк, чем аспектам преобразования. Но нас как фанатов XSLT это не должно беспокоить, так как успех стандартов всегда был обратно пропорционален их длине.

То, что Волш написал по поводу аспектов преобразования в языке, было особенно кратким, и хотя у него явно были аргументы считать эти функции необходимыми, очень плохо, что он не объяснил, почему он внес в этот перечень одни из них и умолчал о других, таких как сортировка, группировка и суммирование:

- Родительские элементы, потомки, элементы одного уровня, атрибуты, содержимое, логическое сложение, отрицание, подсчет, автоматическая выборка на основании выражений произвольных запросов.
- Арифметические выражения, арифметические операции, простые логические сравнения, булева логика, подстроки, сцепление строк.
- Типы данных: скалярные типы, единицы измерений, потоковые объекты (Flow Objects), объекты XML.
- Побочные эффекты: никаких глобальных побочных эффектов.
- Стандартные процедуры: язык выражений должен иметь набор процедур, встроенных в язык XSL. Они пока еще не определены.
- Функции, определяемые пользователем: для повторного использования; параметризованные, но не рекурсивные.

В результате этой инициативы 18 августа 1998 года был опубликован первый Рабочий проект (Working Draft) XSL (не путайте с проектом), и язык начал принимать форму, которая постепенно приобрела окончательный вид, утвержденный 16 ноября 1999 в статусе рекомендации, и прошедший до этого через ряд рабочих проектов. Каждый из этих проектов вносил в язык радикальные изменения, но сохранял первоначальные принципы проектирования.

Рекомендация XSLT 1.0 на момент написания книги все еще остается текущей версией, и ей соответствуют самые современные программные продукты, хотя рабочий проект XSLT 1.1 был опубликован 12 декабря 2000 года. В этой книге упоминаются новые возможности XSLT 1.1, так как они уже начали использоваться в некоторых продуктах; но в книге оговаривается, что это особенности XSLT 1.1, чтобы читателю было понятно, почему их реали-

зуют не все изделия. Однако в большинстве случаев эти возможности основаны на расширениях, которые обеспечивают в своих продуктах XSLT 1.0 многие производители, так что нетрудно найти что-либо похожее.

Теперь давайте рассмотрим существенные особенности XSLT как языка.

XSLT как язык

Каковы наиболее значительные особенности XSLT как языка, которые отличают его от других языков? В этом разделе остановимся на трех наиболее выдающихся из них: во-первых, он написан в синтаксисе XML, во-вторых, он является языком, свободным от побочных эффектов, и, в-третьих, обработка описывается как набор независимых правил сопоставления с образцом.

Использование синтаксиса XML

Как было показано, использование синтаксиса SGML для таблиц стилей было предложено еще в 1994 году, и похоже, что эта идея постепенно завоевала полное признание. Трудно точно отследить, какие аргументы взяли верх, но когда обнаруживаешь себя пишущим нечто такого типа:

```
<xsl:variable name="y">
   <xsl:call-template name="f">
      <xsl:with-param name="x"/>
   </xsl:call-template>
</xsl:variable>
```

чтобы выразить то, что на других языках записывалось бы как «`y = f(x);`», тогда начинаешь удивляться, почему был выбран этот способ.

На самом деле могло быть и хуже: в самых ранних рабочих проектах синтаксисом для того, что теперь является выражениями XPath, был синтаксис XML, поэтому вместо выражения `select="книга/автор/фамилия"` пришлось бы расписывать все по строкам:

```
<select>
   <path>
      <element type="книга">
      <element type="автор">
      <element type="фамилия">
   </path>
</select>
```

Наиболее очевидными аргументами для выражения таблиц стилей XSLT на языке XML являются, возможно, следующие:

- В броузер уже встроен синтаксический анализатор XML, поэтому его размер не слишком увеличится, если анализатор можно использовать повторно.

- Все уже сыты по горло синтаксическими противоречиями между HTML/XML и CSS и не хотят, чтобы это снова повторилось.
- Лиспоподобный синтаксис DSSSL всегда считался помехой к его принятию; лучше иметь синтаксис, уже знакомый целевому сообществу.
- Многие популярные языки шаблонов (включая простые ASP- и JSP-страницы) выражаются в виде схем выходного документа с вложенными инструкциями, так что это уже знакомая концепция.
- Весь лексический аппарат также можно использовать повторно, например поддержку Unicode, ссылки на символы и сущности, обработку пробельных символов, пространств имен.
- Иногда требуется, чтобы таблица стилей была исходными или конечными данными преобразования (подтверждением тому служит, например, преобразователь XSL от Microsoft), так что это замечательно, если таблица стилей может читать и записывать другие таблицы стилей.
- Предоставление визуальных инструментов разработки легко устранит неудобство использования большого количества угловых скобок. (Такие инструментальные средства уже становятся доступными, и некоторые из них обсуждаются в приложении E.)

Нравится это или нет, но синтаксис, основанный на XML, является теперь неотъемлемым свойством языка, который имеет как достоинства, так и недостатки. Он требует много печатания, но, в конечном счете, число нажатий на клавиши не очень отражается на легкости или трудности решения конкретных проблем преобразования.

Никаких побочных эффектов

Идея, что XSL должен быть декларативным языком, свободным от побочных эффектов, неоднократно повторялась в ранних заявлениях о целях и принципах проектирования языка, но, кажется, никто никогда не объяснял, *зачем*: что это дает пользователям?

Считается, что функции или процедуры в языке программирования имеют побочные эффекты, если они производят изменения в своей среде; например, если они могут модифицировать глобальную переменную, которую использует другая функция или процедура, или если они могут записывать сообщения в регистрационный журнал, или запрашивать данные у пользователя. Если функции имеют побочные эффекты, имеет значение количество и порядок их вызова. Функции, не имеющие никаких побочных эффектов (иногда называемые чистыми функциями), могут вызываться любое число раз и в любом порядке. Не имеет значения, сколько раз вычислялась площадь треугольника, ответ будет всегда один и тот же; но если функция, производящая эти вычисления, имеет побочный эффект, например изменяет размеры треугольника, или если неизвестно, имеет она побочные эффекты или нет, тогда становится важно вызывать ее только один раз.

Эта тема будет продолжена в разделе «Вычислительные таблицы стилей» главы 9.

Можно найти объяснение тому, почему в заявлении считалось желательным, чтобы язык подходил и для пакетного, и для интерактивного использования, и чтобы он был способен к постепенному отображению (**progressive rendering**). Существует неприятный момент, когда при загрузке большого XML-документа нельзя ничего увидеть на экране, пока с сервера не будет получено все до последнего байта. Точно так же, внеся в XML-документ небольшое изменение, было бы неплохо иметь возможность определить, какие изменения нужно произвести в экранном отображении, не вычисляя все заново с самого начала. Если язык имеет побочные эффекты, тогда в языке должен быть определен порядок выполнения инструкций, иначе окончательный результат будет непредсказуемым. При отсутствии побочных эффектов инструкции могут выполняться в любом порядке, а это означает, что в принципе можно обрабатывать фрагменты таблицы стилей выборочно и независимо.

Достиг ли XSLT этих целей на самом деле – вопрос спорный. При гибкости выражений и моделей, разрешенных теперь в языке, безусловно, не просто выяснить, на какую область выходного документа воздействует небольшое изменение в каком-то фрагменте входного документа. Кроме того, все существующие XSLT-процессоры требуют, чтобы весь документ был загружен в память. Однако было бы ошибкой ожидать слишком многого так скоро. Когда Е. Ф. Кодд (E. F. Codd) опубликовал в 1970 году реляционное исчисление, он заявил, что декларативный язык был желателен, потому что его можно оптимизировать, что не было возможно с навигационными языками доступа к данным, используемыми в то время. Фактически потребовалось еще пятнадцать лет на то, чтобы методы реляционной оптимизации (и, если быть честным, цена аппаратных средств) достигли уровня, при котором крупные реляционные базы данных стали коммерчески оправданными. Но в конечном счете он оказался прав, и есть надежда, что те же самые принципы со временем тоже приведут к успехам в области языков преобразований и оформления.

Несомненно, всегда будут существовать преобразования, для которых должен быть доступен весь документ, прежде чем можно будет получить любой вывод; например, когда таблица стилей сортирует данные или когда она работает с оглавлением. Но во множестве других преобразований порядок вывода прямо отражает порядок ввода, и в таких случаях должна существовать возможность постепенного отображения. Во время написания книги стали появляться первые признаки того, что конструкторы используют эту возможность: процессор Xalan-Java 2, например, выполняет поток преобразования параллельно с потоком анализа, следовательно, преобразователь может производить вывод еще до окончания работы анализатора; и процессор MSXML3 (судя по его API), кажется, разработан по такому же принципу. Средство отладки Stylus Studio, которое описано в приложении Е, прослеживает зависимости между областями выходного документа и шаблонными правилами, которые использовались для их формирования, так что просматривается потенциальная возможность производить выборочный вывод после выполнения небольших изменений в документе.

На практике отсутствие побочных эффектов означает, что нельзя модифицировать значение переменной. Сначала может показаться, что это ограничение очень неудобно и является слишком дорогой ценой за довольно призрачные выгоды. Но по мере привыкания к языку и приобретения навыков работы с ним таким способом, для которого он проектировался, а не так, как было привычно работать с другими языками, вы перестанете считать это ограничением. На самом деле в этом есть и большой плюс – это устранит целый класс ошибок в вашем коде! Это обсуждение будет продолжено в главе 9. Там приведены некоторые распространенные образцы проектирования таблиц стилей XSLT и, в частности, описано, как использовать рекурсивный код, чтобы справиться с ситуациями, в которых в других языках обычно прибегают к модифицируемым переменным, чтобы следить за текущим состоянием.

Основанный на правилах

Основная особенность типичной таблицы стилей XSLT в том, что она состоит из последовательности шаблонных правил, каждое из которых описывает, как должны обрабатываться элементы конкретного типа или другие конструкции. Правила не организуются в каком-то особом порядке; они не должны соответствовать последовательности элементов структуры входного или выходного документа. Фактически подразумевается, что у автора таблицы стилей нет даже никаких догадок о порядке и вложенности элементов исходного документа. Именно это делает XSLT декларативным языком: необходимо задавать, каким должен быть вывод при появлении заданного образца в исходных данных, в отличие от процедурной программы, где указывается, какие задачи должны быть выполнены и в каком порядке.

Такая структура, основанная на правилах, очень близка структуре CSS. Однако главное отличие XSLT в том, что и образцы (указание, к какому узлу применяется правило), и действия (указание, что должно быть сделано, когда найдено соответствие правилу) гораздо богаче по своим функциональным возможностям.

Пример: Отображение стихотворения

Давайте рассмотрим, как можно использовать подход, основанный на правилах, при форматировании стихотворения. Пока еще не представлены многие концепции, так что каждая подробность объясняться пока не будет, но, тем не менее, полезно увидеть, на что похожи шаблонные правила.

Исходный документ

Пусть стихотворение будет исходным документом XML. Исходный файл называется `poem.xml`, а таблица стилей – `poem.xsl`.

```
<стихотворение>
   <автор>Rupert Brooke</автор>
```

```
    <дата>1912</дата>
    <заголовок>Song</заголовок>
   <строфа>
      <строка>And suddenly the wind comes soft,</строка>
      <строка>And Spring is here again;</строка>
      <строка>And the hawthorn quickens with buds of green</строка>
      <строка>And my heart with buds of pain.</строка>
    </строфа>
    <строфа>
      <строка>My heart all Winter lay so numb,</строка>
      <строка>The earth so dead and frore,</строка>
      <строка>That I never thought the Spring would come again</строка>
      <строка>Or my heart wake any more.</строка>
    </строфа>
    <строфа>
      <строка>But Winter's broken and earth has woken,</строка>
      <строка>And the small birds cry again;</строка>
      <строка>And the hawthorn hedge puts forth its buds,</строка>
      <строка>And my heart puts forth its pain.</строка>
    </строфа>
</стихотворение>
```

Результат

Напишем такую таблицу стилей, чтобы этот документ выглядел в окне броузера так, как показано на рис. 1.3.

***Рис. 1.3.** Отображение конечного документа в окне броузера*

Таблица стилей

Таблица начинается со стандартного заголовка:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
```

Теперь запишем по одному шаблонному правилу для каждого типа элементов в исходном документе. Правило для элемента <стихотворение> создает каркас вывода HTML, определяя упорядочение элементов в конечном документе (порядок не обязательно должен быть таким же, как в исходном файле). Инструкция <xsl:value-of> вставляет значение выбранного элемента в данную точку конечного документа. Инструкции <xsl:apply-templates> вызывают обработку выбранных дочерних элементов с использованием собственного шаблонного правила для каждого.

```
<xsl:template match="стихотворение">
   <html>
   <head>
      <title><xsl:value-of select="заголовок"/></title>
   </head>
<body>
      <xsl:apply-templates select="заголовок"/>
      <xsl:apply-templates select="автор"/>
      <xsl:apply-templates select="строфа"/>
      <xsl:apply-templates select="дата"/>
   </body>
   </html>
</xsl:template>
```

Шаблонные правила для элементов <заголовок>, <автор> и <дата> очень просты: взять содержимое элемента (обозначенное как «select="."») и поместить его внутрь соответствующих HTML-тегов, определяющих стиль его отображения:

```
<xsl:template match="заголовок">
   <div align="center"><h1><xsl:value-of select="."/></h1></div>
</xsl:template>

<xsl:template match="автор">
   <div align="center"><h2><xsl:value-of select="."/></h2></div>
</xsl:template>

<xsl:template match="дата">
   <p><i><xsl:value-of select="."/></i></p>
</xsl:template>
```

Шаблонное правило для элемента <строфа> помещает каждую строфу в абзац HTML, а затем вызывает обработку строк в пределах строфы в соответствии с шаблонным правилом для строк:

```
<xsl:template match="строфа">
```

```
    <p><xsl:apply-templates select="строка"/></p>
  </xsl:template>
```

Правило для элементов <строка> несколько сложнее. Если номер строки в пределах строфы – четное число, она предшествует строке с двумя неразрывными пробелами ( ). Инструкция <xsl:if> проверяет логическое условие, которое в данном случае вызывает функцию position(), чтобы определить относительное положение текущей строки. Затем выводится содержимое строки, сопровождаемое пустым HTML-элементом <br>, заканчивающим строку.

```
  <xsl:template match="строка">
    <xsl:if test="position() mod 2 = 0">&amp;#160;&amp;#160;</xsl:if>
    <xsl:value-of select="."/><br/>
  </xsl:template>
```

И в завершение закрываем элемент <xsl:stylesheet>:

```
  </xsl:stylesheet>
```

Хотя шаблонные правила являются характерной особенностью языка XSLT, это, как будет показано далее, не единственный способ записи таблицы стилей. В главе 9 описываются четыре различных модели проектирования таблиц стилей XSLT, и только одна из них широко использует шаблонные правила. К примеру, таблица стилей *Hello World*, представленная ранее в этой главе, реально не использует шаблонных правил: она отвечает образцу проектирования, названному автором *заполнением бланков* (*fill-in-the-blanks*), так как, по существу, эта таблица содержит неизменяемую часть вывода с вложенными инструкциями, указывающими, где получить данные, которые нужно вставить в изменяемые части.

За рамками XSLT 1.0

Здесь говорилось, откуда возник XSLT 1.0, а что же идет вслед за ним?

После опубликования спецификации XSLT 1.0 рабочая группа XSL, ответственная за разработку языка, решила разбить требования к расширениям на две категории: XSLT 1.1 должен стандартизировать несколько крайне необходимых дополнительных возможностей, которые некоторые производители уже сочли нужным добавить в свои продукты в качестве расширений языка, в то время как XSLT 2.0 должен отвечать более глубоким стратегическим требованиям, которые еще нуждаются в изучении.

XSLT 1.1

Рабочий проект XSLT 1.1 был опубликован 12 декабря 2000 года. Он описывает четыре расширения спецификации XSLT 1.0:

- Несколько выходных документов: инструкцию `<xsl:document>`, введенную на основе расширений, впервые реализованных в процессоре Saxon, а впоследствии и в других продуктах, включая xt, Xalan и Oracle. Эта инструкция позволяет производить из исходного документа несколько выходных документов. (Все эти программы описаны в приложениях книги.)
- Временные деревья: способность рассматривать дерево, созданное в одной фазе обработки в качестве входных данных для последующей фазы обработки. Это расширение было смоделировано на основе функции расширения `node-set()`, введенной сначала в xt и впоследствии воспроизведенной в других продуктах.
- Стандартные привязки функций расширения, написанных на языках Java и ECMAScript. Спецификация XSLT 1.0 допускала вызов внешних функций из таблиц стилей, но не уточняла деталей, так что в итоге функции расширения, написанные для процессора Xalan, не работали бы с XSLT-процессорами xt или Saxon и наоборот. XSLT 1.1 определяет общую концепцию привязок функций расширения, написанных на любом языке, а также конкретные привязки для Java и ECMAScript (официальное название JavaScript).
- Поддержка конструкции `xml:base`, запоздалого дополнения к основным стандартам XML, позволяющего указать в XML-документе базовый URI (URI по умолчанию), который должен использоваться для разрешения всех относительных URI, содержащихся в документе.

Во время написания книги XSLT 1.1 являлся лишь рабочим проектом, и не было никаких официальных заявлений о предполагаемом времени его завершения. Были предложения сначала глубже изучить требования к XSLT 2.0 перед выпуском окончательной спецификации XSLT 1.1, чтобы не помешать реализации этих требований в XSLT 2.0. Кроме того, некоторые считают, что производители еще только начинают брать XSLT 1.0 на вооружение, и поэтому пользователям сейчас больше нужна стабильность, чем расширения.

Несмотря на это, автор решил включить в книгу описание XSLT 1.1. Такое решение принято из расчета, что утвержденные спецификации не будут слишком отличаться от изданного рабочего проекта и что XSLT 1.1 будет реализован в продуктах некоторых поставщиков уже в 2001 году – есть признаки, что это уже происходит. Тем не менее, расширения XSLT 1.1 будут особо подчеркиваться, чтобы читатели могли сверить данные, приведенные в книге, с самыми свежими рекомендациями W3C и с документацией к выбранным для использования продуктам.

Пока книга готовилась к печати, консорциум W3C решил не развивать XSLT 1.1 далее рабочего проекта, а вместо этого использовать его как базис для разработки XSLT 2.

В тексте этой книги ссылки на «рабочий проект XSLT 1.1» подразумевают проект, датированный 12 декабря 2000 года.

XSLT 2.0 и XPath 2.0

14 февраля 2001 года рабочая группа XSL консорциума W3C, работая в тесном сотрудничестве с другими рабочими группами, опубликовала требования к языкам XSLT 2.0 и XPath 2.0; эти документы можно найти по следующим адресам:

http://www.w3.org/TR/xslt20req

http://www.w3.org/TR/XPath20req

Цель опубликования названных документов – ознакомление с ними широкой общественности и вовлечение пользователей в процесс стандартизации, так что было бы ошибкой расценивать предложенные расширения как окончательные. Тем не менее, эти документы позволяют уловить основное направление разработок. В общих чертах данные требования делятся на три категории:

- Возможности, отсутствие которых явно ощущается в текущем стандарте и которые заметно облегчили бы жизнь пользователям: например, средства для группировки связанных узлов, дополнительные функции обработки строк и числовые функции, возможность читать текстовые файлы так же, как XML-документы.
- Дополнительные возможности, разработанные рабочей группой XML Query. XQuery – это спецификация SQL-подобного языка, предназначенного для выполнения поиска данных, удовлетворяющих запросу, в наборе XML-документов и вывода результатов в виде XML-документа. Первый публичный рабочий проект был выпущен 15 февраля 2001 года (см. *http://www.w3.org/TR/2001/WD-xquery-20010215/*), но он основан на идеях из ряда предшествующих проектов, в особенности на идеях языка Quilt разработки Дона Чамберлина (Don Chamberlin), Джонатана Роби (Jonathan Robie) и Даниэлы Флореску (Daniela Florescu). Хотя XQuery предназначен для применения в другом контексте, его функциональные возможности во многом пересекаются с XSLT и XPath. Консорциум W3C по возможности стремится к согласованности своих стандартов, а в предложенной спецификации XQuery есть много особенностей, которые стоит добавить к спецификации XPath, чтобы они согласовывались друг с другом. В чем-то обеим сторонам придется идти и на компромисс, потому что ряд усовершенствований, на которые у группы разработчиков XML Query есть определенные виды, прекрасны в теории, но трудно приспосабливаемы к языку, который уже широко используется.
- Возможности, предусматривающие применение и интеграцию языка XML Schema. Спецификация XML Schema консорциума W3C сейчас находится на стадии усовершенствования (20 октября 2000 года она получила статус кандидата в рекомендации), и ее реализации начали появляться в программных продуктах. XML Schema служит заменой DTD, обеспечивая гораздо больше способов определения типов данных элементов и атрибутов, которые могут встретиться в документе. Идея в том, что

если схема определяет, что конкретный элемент содержит число или дату (например), то должна быть возможность использовать это знание при сравнении или сортировке дат в рамках таблицы стилей. Достижение этого без переворачивания XPath с ног на голову (в настоящее время это очень слабо типизированный язык) явится основной проблемой, и по одной только этой причине, по нашему мнению, разработка спецификаций XPath 2.0 может растянуться на длительное время.

В каких случаях необходимо использовать XSLT

В заключительном разделе этой главы сделана попытка оценить, какие задачи хорошо решаются с помощью XSLT, а для каких лучше подойдут другие инструменты. Здесь рассмотрены также альтернативные способы использования XSLT в рамках полной архитектуры всего приложения.

Вообще говоря, как обсуждалось в начале главы, существует два основных сценария, когда необходимо применять XSLT-преобразования: преобразование данных и их опубликование, каждый из которых будет рассмотрен в отдельности.

Приложения для преобразования данных

Необходимость в преобразовании данных не отпадает в связи с изобретением XML. Даже при том, что все больше данных, передаваемых между организациями или между приложениями в рамках одной организации, вероятно, будет кодироваться в XML, все равно будут сосуществовать различные модели данных, различные способы представления одной и той же информации, и различные подмножества информации, которые представляют интерес для разных людей (можно вернуться к примеру в начале главы, где демонстрировалось преобразование музыки в различные XML-представления и разные форматы отображения). Однако как бы мы ни восхищались XML, еще долго будут использоваться файлы с разделителями-запятыми, сообщения EDI (электронного обмена данными) и разных других форматов.

Для задач преобразования одного набора данных XML в другой (рис. 1.4) XSLT является очевидным выбором.

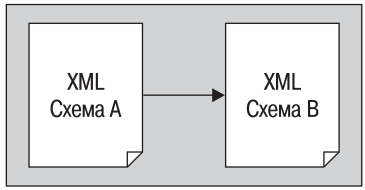


Рис. 1.4. *Преобразование одного набора данных XML в другой*

Такая задача может возникнуть при выборочном извлечении данных, их переупорядочении, превращении атрибутов в элементы или наоборот и т. п. Этим способом можно также просто проверять достоверность данных. Как язык XSLT лучше справляется со структурой информации, чем с ее содержимым: он хорош для обращения строк таблицы в столбцы, но для обработки текстовых строк (например, для удаления любого текста, который заключен в квадратные скобки) его использовать сложнее, чем языки, подобные JavaScript или Perl, которые поддерживают регулярные выражения. Однако всегда можно решить такие проблемы, вызывая прямо из таблицы стилей процедуры, написанные на других языках, таких как Java или JavaScript.

XSLT также полезен для преобразования XML-данных в любой текстовый формат (рис. 1.5), в частности в файлы с разделителями-запятыми или в различные форматы сообщений EDI. Текстовый вывод – это, практически, вывод XML без тегов, так что это не создает особых проблем для языка.

Возможно, более удивительным покажется то, что XSLT часто может быть полезен для преобразования других форматов в XML (рис. 1.6) или во что-то еще:

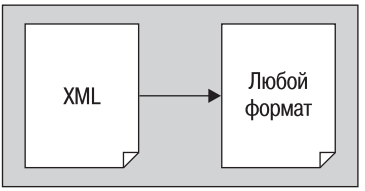


Рис. 1.5. *Преобразование XML в любой текстовый формат данных*

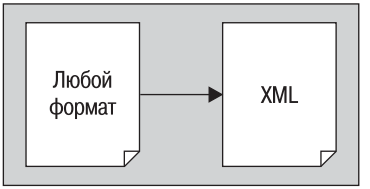


Рис. 1.6. *Преобразование других форматов данных в XML*

В этом случае нужно будет написать программу-анализатор, которая понимает входной формат; но это пришлось бы делать так или иначе. Выгода в том, что при наличии синтаксического анализатора остальную часть преобразования данных можно выразить на высокоуровневом языке. К тому же, такое разделение повышает вероятность того, что написанный синтаксический анализатор можно будет многократно использовать при работе с файлами данного входного формата. В разделе «Генеалогическое дерево» главы 10 приводится пример, где исходные данные находятся в довольно старомодном формате, явно не являющимся XML, широко используемом для обмена данными между пакетами генеалогических программ. Оказалось, что для его обработки даже не требуется преобразовывать данные в XML перед тем, как применить таблицу стилей XSLT: нужно лишь сделать ваш синтаксический анализатор похожим на синтаксический анализатор XML, сделав его реализацией одного из стандартных интерфейсов XML-анализатора: SAX или DOM. Большинство XSLT-процессоров примет ввод от программы, реализующей интерфейс SAX или DOM, даже если эти данные не представлены как XML.

Предостережение по поводу приложений для преобразования данных: во время выполнения преобразований все современные XSLT-процессоры сохраняют все данные в памяти. Древовидная структура в памяти может в де-

сять раз превышать первоначальный объем данных, поэтому практический предел объема данных для XSLT-преобразования – несколько мегабайт. Даже при таком объеме сложное преобразование может занять довольно много времени, в зависимости от конкретных задач обработки.

Одним из обходных способов в подобных случаях является разбиение данных на несколько порций и преобразование каждой из них отдельно – конечно, с сохранением соответствия между входными и выходными порциями данных. Но когда и это оказывается сложным, наступает момент, после которого XSLT – уже не лучший инструмент для подобной задачи. Тогда более подходящим подходом может оказаться, например, загрузка данных в реляционную или объектную базу данных и извлечение их обратно в нужной последовательности с помощью языка запросов к базе данных.

При необходимости периодической обработки больших объемов данных, например для извлечения избранных записей из файлов журнала розничных продаж, лучше написать приложение, используя интерфейс SAX. Возможно, его создание займет немного больше времени, чем написание эквивалентной таблицы стилей XSLT, но оно будет работать, вероятно, во много раз быстрее. А зачастую комбинация фильтрующего SAX-приложения, просто извлекающего данные, с последующим применением таблицы стилей XSLT для более сложных операций может стать лучшим решением в подобных случаях.

Опубликование

Разница между преобразованием и опубликованием данных в том, что в первом случае данные предназначены для ввода в другую программу, а во втором они предназначены для (хотелось бы надеяться) чтения людьми. В этом контексте опубликование означает не только чрезмерное количество текста и мультимедийные элементы оформления, но также и данные. Все – от традиционной деятельности по созданию и распространению отчетов с целью держать менеджеров в курсе состояния дел, до онлайнового формирования выписок из телефонных или банковских счетов для заказчиков и расписаний поездов для широкой публики. XML идеален для таких приложений, издающих данные, а также для более традиционного издания текстовой продукции, где первоначально доминировал SGML.

XML предназначен для хранения информации независимо от способа, которым она будет представляться, что иногда ведет людей к ошибочному мнению, что использовать XML для определения подробностей отображения не годится – по тем или иным причинам. Вовсе нет: если бы сейчас стояла задача разработки нового формата для загрузки шрифтов в принтер, ее, вероятно, решали бы с помощью технологий XML. Детали отображения можно закодировать в XML точно так же, как и любой другой вид информации. Так что можно сказать, что роль XSLT в издательском процессе – преобразовать данные без подробностей представления в «данные с представлением», где те и другие, по крайней мере в принципе, являются XML-форматами.

Два важных средства опубликования информации сегодня – печать на бумаге и Сеть. В области печати на бумаге – больше трудностей из-за высоких требований пользователей к визуальному качеству. Форматирующие объекты XSL (XSL Formatting Objects) пытаются определить основанную на XML модель файла для печати с высоким качеством отображения на бумаге или на экране. Из-за огромного количества параметров, необходимых для достижения этой цели, стандартизация еще не завершена и, вероятно, не скоро реализуется во всей полноте. В то же время Сеть – менее требовательная сфера, для которой нужно всего лишь преобразовать данные в формат HTML и предоставить броузеру отображение их наилучшим образом на имеющемся дисплее. HTML – это, конечно, не XML, но достаточно близко, чтобы простое преобразование было возможно. В настоящее время преобразование XML в HTML – основное применение XSLT. Фактически это двухстадийный процесс. Сначала происходит преобразование в основанную на XML модель, которая является структурным эквивалентом целевого HTML, а затем – вывод в формате HTML, а не строгом XML.

Появление XHTML 1.0, безусловно, еще более упорядочило этот процесс, потому что это уже настоящий XML-формат. Остается только узнать, как быстро пойдет принятие XHTML.

Когда выполнять преобразование?

Ниже схематично представлен процесс опубликования информации.

Рис. 1.7. *Схема процесса опубликования информации*

В такой системе есть несколько мест, где можно воспользоваться XSLT-преобразованиями:

- Информация, которую авторы вводят с помощью предпочитаемых ими инструментальных средств или специализированных интерфейсов для заполнения форм, может быть преобразована в XML-формат и сохраняться в таком виде в хранилище содержимого.
- XML-информация, поступающая от других систем, может быть преобразована в некоторую разновидность XML для помещения ее в хранилище содержимого. Например, она могла бы быть разбита на порции размером со страницу.
- XML может быть преобразован в HTML на сервере, когда пользователи запрашивают страницу. Этим процессом можно управлять с помощью таких технологий, как Java-сервлеты или серверные страницы Java (Java Server Pages). На сервере Microsoft можно использовать расширение XSL ISAPI, доступное по адресу *http://msdn.microsoft.com/xml*, или, если нужен больший контроль над приложением, можно вызывать преобразование из сценария в ASP-страницах.
- XML может быть передан в систему клиента и преобразован в HTML-формат прямо в броузере. Это может привести к улучшению интерактивности представления информации и существенно сократить нагрузку на сервер, но при этом необходимо, чтобы все пользователи имели броузеры, способные справиться с такой задачей.
- XML-данные также могут быть преобразованы в их окончательную форму отображения во время опубликования и сохраняться как HTML в хранилище содержимого. Это минимизирует работу, которая должна быть сделана во время отображения, и совсем хорошо, если одну и ту же преобразованную страницу можно предоставлять очень многим пользователям.

Выбор метода не может быть однозначно правильным, и часто имеет смысл комбинировать разные методы. Преобразование непосредственно в броузере – очень привлекательная идея, поскольку XSLT уже доступен во многих броузерах, но это все равно не совсем то, что нужно. Даже если это сделано, может потребоваться некоторая серверная обработка, чтобы передавать XML порциями и защищать конфиденциальную информацию. Преобразование на сервере во время доставки пользователю – популярный выбор, потому что он позволяет персонализацию, но увеличивает загрузку сайтов с большим трафиком. Некоторые сайты находят более эффективным заранее генерировать разные наборы HTML-страниц для разной потенциальной аудитории, чтобы во время запроса страницы оставалось только выбрать уже готовую страницу HTML.

Резюме

В этой вводной главе обсуждались все «почему» и «для чего» языка XSLT. Это была попытка ответить на вопросы:

- Какой это язык?
- Как он вписывается в семейство XML?
- Откуда он происходит и почему он был разработан именно таким?
- Где его можно использовать?

Сейчас читатели знают, что XSLT – это декларативный язык высокого уровня, предназначенный для преобразования структуры XML-документов, что есть две крупные области его применения: преобразование данных и их опубликование и что XSLT может использоваться в разных местах архитектуры всего приложения, включая этапы сбора данных, доставки сервером и отображения в броузере. Кроме того, читатели получили некоторое представление о том, почему XSLT стал именно таким, какой он есть.

Теперь настало время углубиться в сущность языка, чтобы понять, каким образом он выполняет свои задачи. В следующей главе будет обсуждаться, как выполняется преобразование при помощи представления структуры входных и выходных данных в виде деревьев, как используются образцы для поиска соответствующих им узлов в дереве входного документа и как определяется, какие узлы должны быть добавлены в этом случае к дереву конечного документа.

2

Модель обработки данных в XSLT

В этой главе приведен общий обзор работы XSLT-процессора. Начнем с обзора системы: какие данные вводятся в процессор и формируются на выходе.

Затем рассмотрим более подробно модель данных, в особенности представление структуры XML-документов в виде дерева. Здесь важно отметить, что XSLT-преобразования оперируют с XML-документами не как с текстом, а как с абстрактной древовидной информационной структурой, представленной текстом.

Установив модель данных, обсудим последовательность обработки, которая происходит, когда сводятся вместе исходный документ и стилевая таблица стилей. XSLT – это не традиционный процедурный язык: он состоит из набора шаблонных правил, определяющих, какой вывод формируется, когда отдельные правила соответствуют вводу. Как отмечалось в главе 1, эта структура обработки, основанная на правилах, – одна из отличительных особенностей языка XSLT.

Наконец, рассмотрим, каким образом могут использоваться в таблице стилей XSLT переменные и выражения, а также обсудим различные доступные типы данных.

XSLT: обзор системы

В этом разделе рассматривается структура процесса, выполняемого XSLT.

Беглый обзор

Основная задача XSLT-процессора состоит в том, чтобы применить таблицу стилей к исходному документу и сформировать конечный документ. Это показано ниже в упрощенной схеме (рис. 2.1):

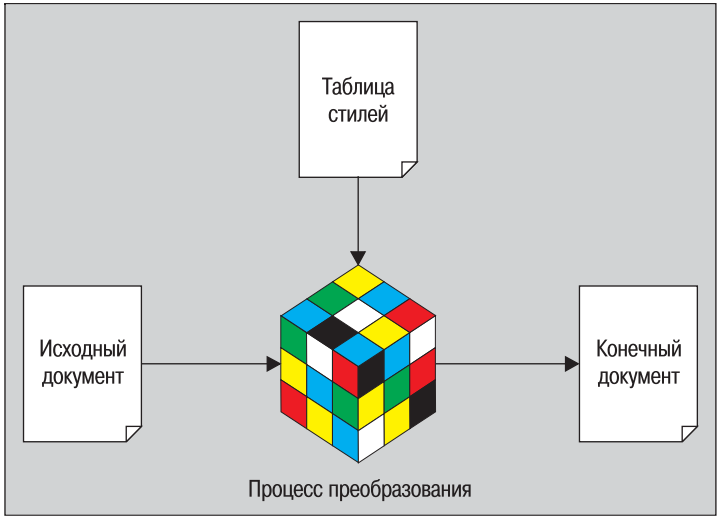


***Рис. 2.1.** Общая схема преобразования исходного документа в конечный с помощью таблицы стилей*

В первом приближении можно считать, что исходный документ, таблица стилей и конечный документ – все являются XML-документами. XSLT выполняет процесс **преобразования**, потому что **вывод** (конечный документ) и **ввод** (исходный документ) представляют собой объекты одного типа. Это дает прямые выгоды: например, сложное преобразование можно выполнять как ряд простых преобразований, кроме того, используя одну и ту же технологию, можно делать преобразования в любом направлении.

> *Выбор кубика Рубика для иллюстрации процесса преобразования вовсе не надуман. Математика кубика Рубика основана на теории групп, из которой пришло понятие «замкнутость»: каждая операция преобразует один экземпляр некоторого типа в другой экземпляр того же самого типа. Здесь преобразуются XML-документы, а не кубики, но принцип тот же самый.*

Название **таблица стилей** присвоено документу, который определяет преобразование, хотя пуристы предпочитают называть его **таблицей преобразований**. Название отражает сущность процесса, который происходит при самом общем типе преобразования, выполняемого с помощью XSLT, а именно: определить стиль отображения информации в исходном документе, причем так, чтобы конечный документ содержал информацию исходного документа, дополненную информацией, которая управляет способом его отображения на некотором устройстве вывода.

Деревья, а не документы

На практике не всегда нужно, чтобы ввод или вывод представляли собой XML в его канонической форме. Если требуется произвести вывод HTML (наиболее часто возникающая потребность), хотелось бы получить его непо-

средственно, а не через промежуточное формирование XML-документа. Точно так же может понадобиться брать исходные данные из базы данных или, скажем, из каталога LDAP, или из сообщений EDI, или из файла данных, использующего синтаксис значений, с разделителями-запятыми. Не хотелось бы тратить много времени, конвертируя их в XML-документы, если можно обойтись без этого, да и установки множества конверторов тоже неплохо было бы избежать.

XSLT вместо этого описывает необходимые действия в терминах особого представления XML-документа, которое называется **деревом**. Дерево – это абстрактный тип данных. Нет никакого особого API и никакого определенного представления данных; есть только концептуальная модель, которая определяет объекты в дереве, их свойства и их взаимосвязи. По концепции модель дерева близка модели DOM консорциума W3C, с той лишь разницей, что DOM имеет определенный API. В некоторых реализациях в качестве внутренней древовидной структуры используется именно модель DOM. В других выбрана структура данных, которая более близка модели дерева XPath, а также есть примеры использования внутренних структур данных, которые лишь отдаленно связаны с этой моделью. Таким образом, это – концептуальная модель, и она не является чем-то, что обязательно существует в каждой реализации.

Рассмотрение исходных и конечных данных XSLT-процессоров в виде деревьев дает следующую схему.

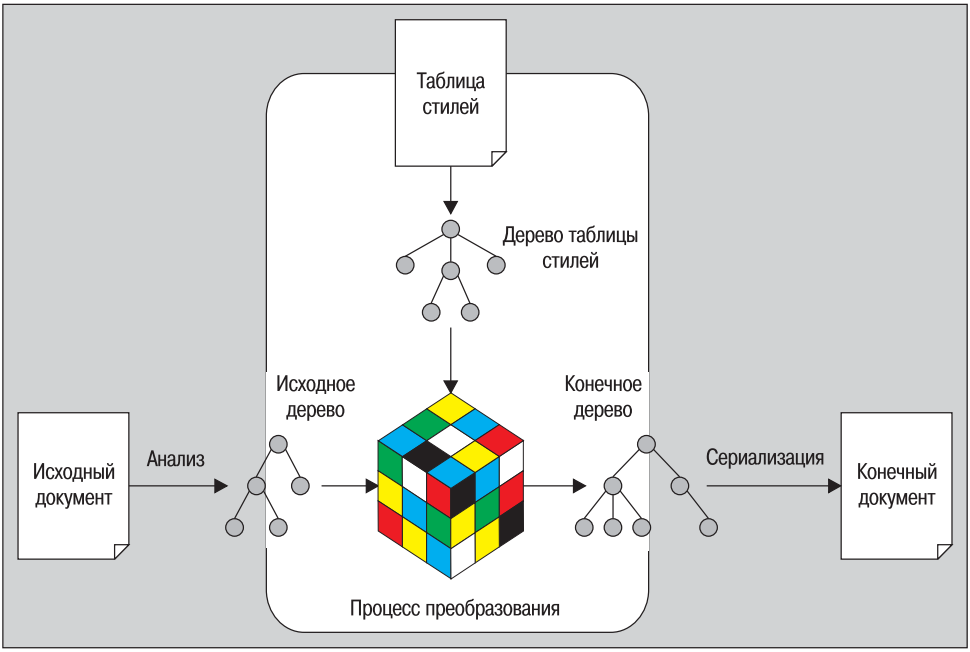


Рис. 2.2. *Общая схема преобразования документов, представленных древовидными структурами*

Формальные правила соответствия стандарту требуют, чтобы XSLT-процессор мог читать таблицу стилей и использовать ее для преобразования исходного дерева в конечное дерево. На схеме эта часть системы показана в рамке со скругленными углами. Нет никаких официальных требований по обработке части процесса, которая находится вне этой рамки, а именно: по созданию исходного дерева из исходного XML-документа (этот процесс называется анализом) или по созданию из конечного дерева конечного XML-документа (этот процесс часто называют сериализацией). Тем не менее, большинство существующих инструментальных средств, вероятно, выполнит и эти операции.

Позже в этой главе модель данных в виде деревьев будет обсуждаться более подробно и будет показано, как она отражает строение XML-документов.

Различные выходные форматы

Хотя заключительный процесс преобразования конечного дерева в конечный документ – вне требований соответствия стандарту XSLT, это не означает, что XSLT вообще не касается этого предмета. На самом деле в спецификации целый раздел посвящен этой стадии преобразования, и хотя он считается необязательным, большинство реализаций отвечает его требованиям. Основной контроль за этим процессом возложен на элемент `<xsl:output>`, который подробно описан в главе 4 в соответствующем разделе.

Элемент `<xsl:output>` определяет три выходных формата или метода вывода, а именно: `xml`, `html` и `text`. В каждом из этих случаев конечное дерево записывается в единственный выходной файл.

- В методе вывода `xml` выходным файлом является XML-документ. Позже будет показано, что это не обязательно должен быть законченный XML-документ, это также может быть фрагмент XML. Элемент `<xsl:output>` дает создателю таблицы стилей некоторый контроль над способом формирования XML, например над используемой кодировкой символов, и использованием разделов `CDATA`.
- В методе вывода `html` выходным файлом является документ HTML, обычно HTML 4.0, хотя могут поддерживаться и другие версии. В случае выходного формата HTML XSLT-процессор распознает многие конструкции HTML и формирует вывод в соответствии с ними. Например, он распознает такие элементы, как `<hr>`, которые имеют открывающий тег, но не имеют закрывающего тега, а также распознает специальные правила экранирования символов в пределах элемента `<script>`. Кроме того, он может (если это задано) генерировать ссылки на встроенные сущности типа «`&eacute;`».
- Метод вывода `text` предназначен для того, чтобы было возможно получить выходной документ **в любом текстовом формате**. Например, выходным файлом мог бы быть файл с разделителями-запятыми, файл в формате RTF или в формате переносимого документа PDF (Adobe) или это могло бы быть сообщение электронного обмена данными, или сценарий SQL или JavaScript. Здесь полная свобода выбора.

Выходной формат XHTML явным образом не оговорен, но так как XHTML – это, по сути, чистый XML, он может быть получен с помощью метода `xml` таким же образом, как любой другой тип XML-документа.

Если элемент `<xsl:output>` опущен, процессор делает обоснованное предположение, выбирая HTML, если вывод начинается с тега `<html>`, а по умолчанию – XML.

В конкретные реализации могут быть включены также другие методы, но это уже вне требований стандарта. Некоторые продукты поддерживают следующий механизм: конечное дерево направляется в предоставляемый пользователем обработчик документов. Как правило, этот обработчик соответствует интерфейсу `ContentHandler`, определенному в спецификации SAX2 API, которая доступна по адресу *http://www.megginson.com/*, или интерфейсу `DocumentHandler`, определенному в более раннем SAX1, описанном в книге издательства Wrox Press «Professional XML» («XML для профессионалов»).

Таким образом, хотя большая часть рекомендации XSLT посвящена процессу преобразования исходного дерева в конечное дерево, там есть один раздел (а именно раздел 16 «Вывод»), который описывает другой процесс – процесс вывода. Эту стадию часто называют сериализацией потому, что в это время происходит превращение древовидной структуры в поток символов (но это не та сериализация, которая происходит в распределенных объектных системах, например COM и Java, и которая заключается в генерации последовательных файлов с представлениями объектов COM или Java). XSLT-процессоры могут осуществлять эту стадию по своему усмотрению, как показано на схеме ниже.

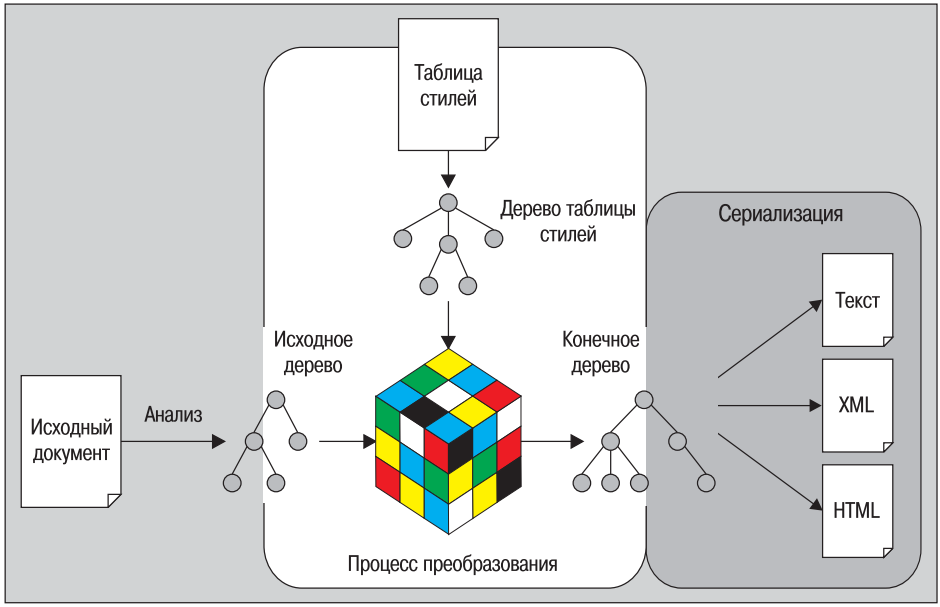


Рис. 2.3. *Общая схема процесса сериализации (превращения древовидной структуры в поток символов)*

Множественные документы ввода и вывода

В реальных ситуациях модель обработки еще усложняется, так как может быть сразу несколько исходных и конечных документов.

В частности:

- Возможны ситуации, когда исходных документов несколько. Таблица стилей может использовать функцию `document()` (описанную в главе 7 в соответствующем разделе), чтобы загрузить дополнительные исходные документы, URI которых находятся в исходном документе или в таблице стилей. Каждый исходный документ, в свою очередь, обрабатывается как дерево тем же самым способом, что и основной исходный документ. Можно также задавать дополнительные исходные документы в параметрах таблицы стилей.
- Таблица стилей тоже может состоять из нескольких документов. Определены две директивы, `<xsl:include>` и `<xsl:import>`, которые можно использовать в таблице стилей для загрузки дополнительных модулей таблицы и применения их в качестве расширения основного модуля. Таким образом, разбиение таблицы стилей обеспечивает модульность: для нетривиальных задач различные аспекты обработки могут быть описаны во вспомогательных таблицах стилей, которые можно вложить в несколько разных родительских таблиц стилей. Детальное обсуждение разбиения таблиц стилей на модули приведено в главе 3.

Один запуск XSLT-процессора также может произвести несколько конечных документов. Благодаря этому из одного исходного документа можно сформировать несколько выходных файлов: например, исходный файл мог бы содержать текст целой книги, а в результате были бы выведены отдельные HTML-файлы каждой главы, связанные соответствующими гиперссылками. Эта возможность, обеспечиваемая элементом `<xsl:document>`, который описывается подробно в главе 4, стандартизирована в рабочем проекте спецификации XSLT 1.1, хотя эквивалентные средства, осуществляемые другими способами, есть и в некоторых продуктах, реализующих XSLT 1.0.

Древовидная модель

Рассмотрим теперь древовидную модель, используемую в XSLT, более подробно. Фактически эта модель частично описана в стандарте XPath, а частично – непосредственно в XSLT; здесь для удобства просто будут объединены эти два описания.

Древовидная модель XSLT во многом подобна объектной модели документа XML (DOM). Однако имеется ряд различий в терминологии и некоторых деталях. Эти различия будут подчеркиваться по мере обсуждения.

XML в виде дерева

На простом уровне эквивалентность текстового представления XML-документа и представления в виде дерева достаточно очевидна.

Пример: XML-дерево

Рассмотрим следующий документ:

```
<определение>
   <слово>экспорт</слово>
   <часть-речи>гл</часть-речи>
   <значение>Вывоз товаров за границу.</значение>
   <этимология>
      <язык>латинский</язык>
      <части>
         <часть>
            <префикс>ex</префикс>
            <значение>наружу</значение>
         </часть>
         <часть>
            <слово>portare</слово>
            <значение>перевозить</значение>
         </часть>
      </части>
   </этимология>
</определение>
```

Если представить каждую область текста в виде листа дерева, а каждый элемент – как содержащий его узел, то можно сформировать из них эквивалентную древовидную структуру (рис. 2.4). На схеме изображено дере-

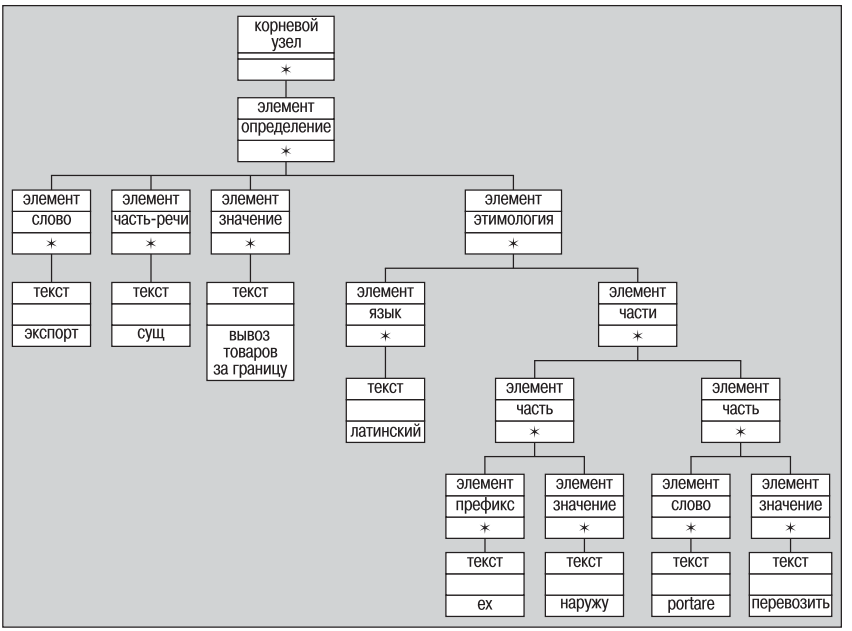


Рис. 2.4. *Дерево документа (с удаленными узлами пробельных символов)*

во после удаления всех узлов, состоящих из пробельных символов (этот процесс обсуждается в разделе «Пробельные символы» главы 3). На схеме каждый узел дерева разбит на три области, содержащие следующую информацию: в верхней ячейке – **тип** узла, в средней – **имя** узла, а в нижней – его **строковое значение**. Для корневого узла и элементов вместо строковых значений указан символ «*»: фактически строковое значение этих узлов определяется путем объединения строковых значений узлов всех элементов и текстовых узлов следующего уровня дерева.

Легко понять, как можно аналогично отразить в этом древовидном представлении другие аспекты XML-документа, например атрибуты и инструкции обработки, с помощью дополнительных узлов.

Вершиной каждого дерева является **корневой узел (root)**. Он выполняет ту же функцию, что и узел Document в модели DOM: он не соответствует никакой особой части исходного документа, а отражает весь документ в целом. Потомки корневого узла – это элементы самого верхнего уровня, комментарии, инструкции обработки и так далее.

> ***Сам по себе корневой узел не является элементом. В некоторых других спецификациях самый внешний элемент считается корнем документа, но в XPath или XSLT это не так. В модели XPath корень является родителем самого внешнего элемента, и он представляет весь документ в целом.***

С помощью древовидной модели XSLT можно представить любой правильно построенный XML-документ, а также структуры, которые не являются правильно построенными, согласно определению XML. В частности, в правильно построенном XML должен иметься единственный самый внешний элемент, который содержит в себе все другие элементы и текстовые узлы; этому элементу (спецификация XML называет его элементом документа, хотя XSLT не использует этот термин) могут предшествовать (и находиться за ним) комментарии и инструкции обработки, но ему не могут предшествовать другие элементы или текстовые узлы.

Древовидная модель XSLT не накладывает это ограничение; корень может иметь любых непосредственных потомков, которых мог бы иметь любой элемент, включая многочисленные элементы и текстовые узлы в любом порядке. Кроме того, корень может и не иметь потомков вообще. Это соответствует правилам XML в отношении содержимого **внешних общих анализируемых сущностей**, которые являются автономными фрагментами XML и которые могут быть встроены в правильно построенный документ посредством ссылки на сущность. В книге такие сущности иногда будут называться **сбалансированными**. Этот термин не используется в спецификации XSLT, он позаимствован из редко упоминаемой спецификации по обмену фрагментов XML (*http://www.w3.org/TR/xml-fragment*). Характерная особенность сбалансированного фрагмента XML в том, что каждый открывающий тег элемента уравновешен соответствующим закрывающим тегом.

Пример: Сбалансированный фрагмент XML

Здесь приведен пример фрагмента XML, который является сбалансированным, но неправильно построенным, так как в нем нет объемлющего элемента, в который вложены остальные:

```
<существительное>Кошка</существительное> <глагол>сидела</глагол> на
<существительное>циновке</существительное>.
```

А ниже показано соответствующее дерево XPath. В этом случае важно сохранить пробельные символы, поэтому они обозначаются символом ♦:

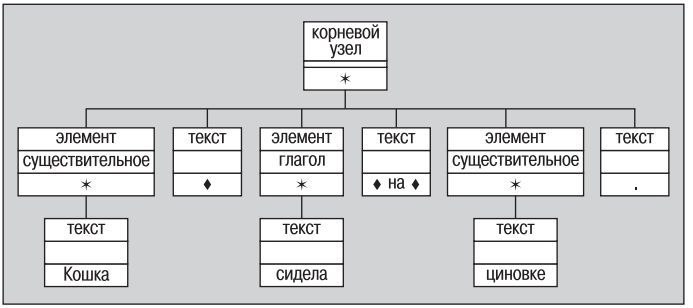


***Рис. 2.5.** XPath-дерево сбалансированного, но неправильно построенного XML-документа*

Строковое значение корневого узла в этом примере – просто: «Кошка сидела на циновке».

На практике исходными данными и результатом преобразования обычно будут правильно построенные документы, но корневые узлы временных деревьев, создаваемых в ходе обработки, очень часто будут иметь несколько непосредственных потомков.

Узлы в модели дерева

Дерево XPath строится из узлов, которые могут быть семи различных типов. Эти типы узлов довольно точно соответствуют компонентам исходного XML-документа:

Тип узла	Описание
Корневой узел	Корневой узел – это единичный узел; каждый документ имеет один такой узел. Корневой узел в модели XPath выполняет ту же функцию, что и узел Document в модели DOM. Но не путайте корень с элементом документа, который в правильно построенном документе является самым внешним элементом, включающим в себя все другие.

Тип узла	Описание
Узел элемента	Элемент – это часть документа, ограниченная открывающим и закрывающим тегами или представленная единственным тегом пустого элемента: <ТЕГ/>.
Текстовый узел	Текстовый узел – это непрерывная последовательность символов в разделе PCDATA элемента. Узлы всегда делаются настолько большими, насколько это возможно: в дереве никогда не будут присутствовать два соседних текстовых узла, потому что они будут объединены вместе. В терминологии DOM говорят, что текстовые узлы **нормализуются**.
Узел атрибута	Узел атрибута включает в себя имя и значение атрибута, записываемые в открывающем теге элемента (или теге пустого элемента). Атрибут, который не указан в теге, но для которого определено значение по умолчанию в DTD, также представляется как узел атрибута в каждом отдельном экземпляре элемента. Однако объявление пространства имен (атрибут, имя которого – xmlns или начинается с xmlns:) не представляется в дереве как узел атрибута.
Узел комментария	Узел комментария представляет собой комментарий, написанный в исходном XML-документе между разделителями «<!--» и «-->».
Узел инструкции обработки	Узел инструкции обработки представляет инструкцию обработки, записанную в исходном XML документе между разделителями «<?» и «?>». В качестве имени узла берется цель инструкции обработки исходного XML, а остальная часть содержимого – в качестве его значения. Заметьте, что XML-объявление <?xml version="1.0"?> не является инструкцией обработки и у него нет соответствующего узла в дереве, хотя оно и выглядит так же.
Узел пространства имен	Узел пространства имен представляет объявление пространства имен, кроме того, он копируется в каждый элемент, к которому оно относится. Таким образом, каждый узел элемента содержит по одному узлу пространства имен для каждого пространства имен, в области действия которого находится этот элемент. Узлы пространств имен, принадлежащие одному элементу, отличаются от узлов пространств имен, принадлежащих другому элементу, даже когда они происходят из одного и того же объявления пространства имен в исходном документе.

Существует несколько возможных путей классификации узлов дерева. Можно выделять в отдельную группу узлы, которые могут иметь потомков (элементы и корень); узлы, имеющие родителя (все, кроме корня); узлы, имеющие имя (элементы, атрибуты, пространства имен и инструкции обработки), или узлы, которые имеют свое собственное текстовое содержимое (атрибуты, текст, комментарии, инструкции обработки и пространства имен). Поскольку каждый из этих критериев дает различную возможную иерархию классов, древовидная модель XSLT оставляет иерархию простой,

а вместо этого четко определяет все эти характеристики для всех узлов. Если какая-то характеристика у конкретного узла отсутствует, для нее задается нулевое или пустое значение. Так, если выразить иерархию классов в нотации UML, получится простая схема, приведенная ниже.

Унифицированный язык моделирования UML (Unified Modeling Language) – это набор графических условных обозначений для объектно-ориентированного анализа и проектирования. Дополнительную информацию по UML можно найти на веб-странице http://www.omg.org/technology/uml/index.htm или в книге Пьера-Элейна Мюллера (Pierre-Alain Muller) «Instant UML» (Современный UML), Wrox Press, ISBN 1-861000-87-1.

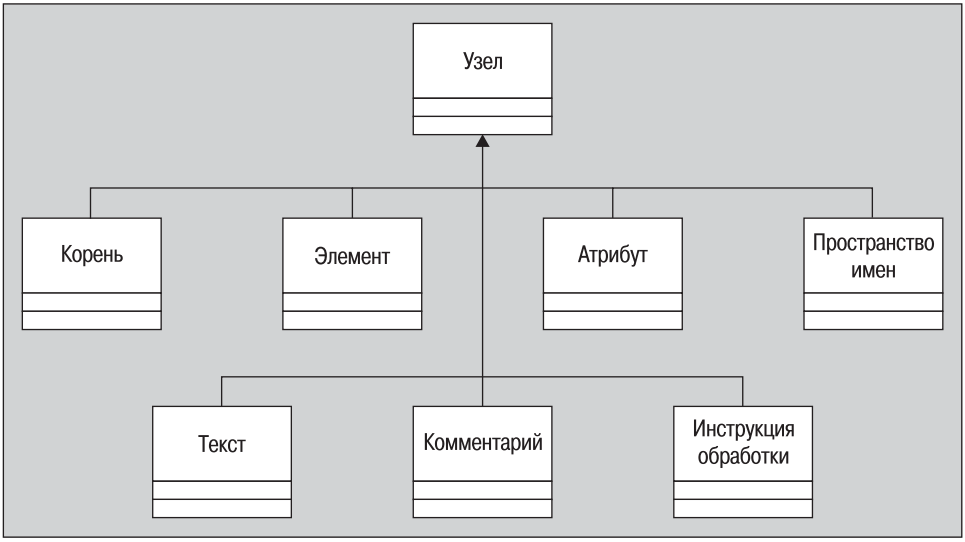


***Рис. 2.6.** Иерархия классов древовидной модели в нотации UML*

На первый взгляд эта схема похожа на дерево, показанное ранее, но в данном случае она иллюстрирует не конкретное дерево, а иерархию классов: прямоугольники представляют классы или типы, а стрелка отражает взаимоотношение типа «является подтипом»: например комментарий является подтипом узла. На предшествующей схеме был просто пример конкретного дерева, тогда как теперь рассматривается общая структура всех возможных деревьев.

Некоторые свойства и взаимосвязи разных узлов уже упоминались. Рассмотрим их теперь более подробно, а затем добавим их к схеме.

Имя узла

Любой узел имеет имя.

Для корневого узла, узлов комментариев и текстовых узлов – это просто пустая строка. (Заметьте, что это отличается от модели DOM, где используются имена типа `«#comment»`.)

Для элементов и атрибутов именами служат их имена из исходного текста XML, дополненные соответствующими объявлениями пространств имен. Для инструкции обработки именем служит цель из исходного текста XML, которая не подчиняется правилам пространства имен.

В соответствии с соглашением, за имя узла пространства имен принимается его префикс из объявления пространства имен (без строки «xmlns:»). Например, объявление пространства имен «xmlns:acme="http://acme.com/xml"» генерирует узел пространства имен с именем «acme», в то время как объявление пространства имен по умолчанию «xmlns="http://acme.com/xml"» генерирует узел пространства имен, именем которого является пустая строка.

Имена записываются в исходном, а также, конечно, и в конечном XML-документе как полные имена (QName, что является сокращением от **qualified name**). Полное имя имеет две части: префикс, которым служит часть полного имени перед знаком «:» из исходного XML, и локальную часть, которая является частью полного имени после знака «:». Если двоеточия нет, префикс является пустой строкой. Например, «xsl:stylesheet» – это полное имя с префиксом «xsl» и локальной частью «stylesheet».

В древовидной модели, однако, имена представляются в расширенном варианте. Расширенное имя также имеет две части, хотя никакого явного синтаксиса для их отображения нет. Этими двумя частями служат URI пространства имен и локальная часть. URI пространства имен получаются из префикса, приведенного в исходном документе путем отыскания соответствующих объявлений пространства имен в той области действия, где он используется, в то время как локальная часть имени – это опять часть полного имени после знака «:». Следовательно, расширенное имя, соответствующее полному имени «xsl:stylesheet», имеет URI пространства имен «http://www.w3.org/1999/XSL/Transform» (в предположении, что употребляются стандартные объявления пространства имен) и локальную часть «stylesheet».

Сам префикс не является формальной частью имени узла, представляемой в модели дерева. Когда отыскивается узел с конкретным именем, имеются в виду только URI пространства имен и локальное имя, которое нужно системе, но не префикс.

Бывают ситуации, когда система должна генерировать полное имя из расширенного имени, в частности, когда используется функция name(), описанная в главе 7, и когда дерево преобразуется в последовательную форму для формирования конечного XML-документа. В этих случаях система назначит префикс, основываясь на узлах пространств имен дерева. Обычно это дает такой же префикс, какой использовался в исходном документе, но может случиться и так, что префиксы будут отличаться. Например, если в исходном документе для одного и того же пространства имен использовались два различных префикса. Однако процессор всегда будет выводить тот префикс, который относится к верному URI пространства имен.

Строковое значение узла

Всякий узел имеет строковое значение, которое является последовательностью символов Unicode.

Для текстового узла это – текст, в том виде, как он приведен в исходном XML-документе, притом что сочетания конца строки вида `#xD #xA` уже заменены XML-анализатором на символ новой строки `#xA`.

Для комментариев это – текст комментария без разделителей.

Для инструкций обработки, это – раздел данных инструкции обработки в исходном документе, за исключением пробельных символов, которые отделяют их от цели инструкции обработки.

Для атрибута это – значение атрибута.

Для корневого узла или узла элемента строковое значение определяется сцеплением строковых значений всех дочерних элементов и текстовых узлов данного узла. Или иначе: сцеплением всех разделов `PCDATA`, содержащихся в элементе (или в документе – для корневого узла) после удаления всей разметки. (Это снова отличается от модели DOM, где свойство `nodeValue` в подобных случаях является пустым.)

Для узла пространства имен строковым значением, в соответствии с соглашением, является URI объявляемого пространства имен.

Базовый URI узла

Каждый узел имеет базовый URI. Его не следует путать с URI его пространства имен. Базовый URI узла зависит от URI источника, из которого был загружен XML-документ, или, точнее, от URI внешней сущности, из которой он был загружен, так как разные части одного документа могут происходить из различных сущностей XML. Базовый URI используется при определении значения относительного URI, который является частью значения для этого узла, например, атрибут `href` всегда интерпретируется относительно базового URI узла, из которого он происходит.

В XSLT 1.1 возможно изменить его, явно определив базовый URI с помощью атрибута `xml:base`. Например, если элемент имеет атрибут `«xml:base="../index.xml"»`, тогда базовым URI для этого элемента и для всех его потомков, если только они находятся в той же самой внешней XML-сущности, является файл `index.xml` в каталоге, уровнем выше родительского каталога файла, базовый URI которого использовался бы в противном случае.

Базовый URI хранится только для узлов элемента и узлов инструкций обработки. Для узлов атрибутов, комментариев и текстовых узлов базовый URI – тот же, что и URI их родительского узла. Для корневого узла это – URI сущности документа.

Для узла пространства имен базовый URI зависит от конкретной реализации. Что довольно любопытно. Единственная ситуация, когда может понадобиться базовый URI узла пространства имен, – когда URI пространства

имен используется как URI реального ресурса, например схемы. Но даже в этом случае он будет необходим, только если URI пространства имен задано относительным URI. Консорциум W3C после горячих споров решил, что относительные URI пространств имен не рекомендуется использовать и их значение является зависимым от реализации, поэтому казалось разумным, чтобы рабочая группа XSL уточнила его интерпретацию.

Текстовые узлы намеренно не имеют собственного базового URI, так как текстовый узел не обязательно происходит из той же самой внешней сущности, что и его родительский элемент; это отражает решение, что текстовые узлы должны объединяться, независимо от границ сущности.

Базовый URI узла в исходном документе служит только для одной цели: для разрешения ссылок на относительные URI при загрузке дополнительных входных документов через функцию `document()`, описанную в главе 7.

Потомки узла

Любой узел имеет список (упорядоченный набор) узлов непосредственных потомков. Эта связь одного со многими определяется для всех узлов, но список будет пустым для всех узлов, кроме корневого узла и узлов элементов. Так что можно запрашивать потомков атрибута, но при этом будет возвращен пустой набор узлов.

Непосредственными потомками элемента являются элементы, текстовые узлы, инструкции обработки и комментарии, которые содержатся между его открывающим и закрывающим тегами, при условии, что они не являются еще и потомками некоторого элемента более низкого уровня.

Непосредственные потомки корневого узла – все элементы, текстовые узлы, комментарии и инструкции обработки, которые не содержатся в другом элементе. В правильно построенном документе непосредственными потомками корневого узла будут элемент документа плюс любые комментарии или инструкции обработки, которые находятся до или после элемента документа.

Атрибуты элемента не расцениваются ни как его дочерние узлы; ни как его узлы пространства имен.

Родитель узла

Все узлы, кроме корня, имеют родителя. Родительские отношения не являются точной инверсией потомственных отношений: в частности узлы атрибутов и узлы пространств имен имеют в качестве родителя узел элемента, **но сами они не считаются его дочерними узлами**. В других случаях, однако, отношения симметричны: элементы, текстовые узлы, инструкции обработки и комментарии обязательно являются непосредственными потомками своего родительского узла, которым всегда будет или элемент, или корень.

Атрибуты узла

Ниже на схеме показано реально существующее отношение между узлами элементов и узлами атрибутов. Это отношение «один ко многим»: один эле-

мент может иметь ноль или более атрибутов. Фактически отношение атрибуты определено для всех узлов, но если запросить атрибуты любого узла, отличного от элемента, результатом будет пустой набор узлов.

Пространства имен узла

Ниже на схеме показано реально существующее отношение между узлами элементов и узлами пространств имен, как показано ниже на схеме. Это отношение «одного ко многим»: один элемент может иметь ноль или более узлов пространств имен. Подобно отношению атрибуты, отношение пространства имен определено для всех узлов; так, если запросить пространства имен любого узла, отличного от элемента, результатом будет пустой набор узлов.

Заметьте, что каждый узел пространства имен однозначно принадлежит только одному элементу. Если в области действия объявления пространства имен в исходном документе находится несколько элементов, то для каждого из этих элементов будет сгенерирован соответствующий узел пространства имен. Все эти узлы будут иметь одно и то же название и строковое значение, но при подсчете и использовании оператора объединения они будут рассматриваться как различные узлы.

Завершение диаграммы классов UML

Теперь можно начертить более полную диаграмму классов UML. В этом варианте:

- Контейнер вынесен как отдельный класс, чтобы отличать те узлы, которые имеют потомков (корень и элементы), от тех, которые не имеют их. Это сделано только для наглядности: явной концепции узла контейнера в формальной модели нет.
- Указано отношение родитель между каждым узлом и его родителем.
- Указаны отношения между элементом и его атрибутами и между узлом элемента и узлами его пространств имен.
- Определен дополнительный класс НеанализируемаяСущность. Это не самостоятельный узел дерева. Это соответствует объявлению неанализируемой сущности в рамках DTD документа, и эта информация доступна только при использовании функции `unparsed-entity-uri()`.
- Определены два дополнительных свойства узла элемента: БазовыйURI, являющийся URI сущности, в которой находятся открывающий и закрывающий теги элемента, и ID, который является значением ID-атрибута для этого элемента, если он его имеет (рис. 2.7).

Следует отметить, что древовидная модель XSLT никогда не использует значения null в том смысле, в каком их используют SQL или Java. Если узел не имеет никакого строкового значения, то возвращенным значением будет пустая строка – строка нулевой длины. Если узел не имеет никаких потомков, то возвращенным значением будет пустой набор узлов – набор, не содержащий никаких членов. Нет никакой разницы между пустой строкой и от-

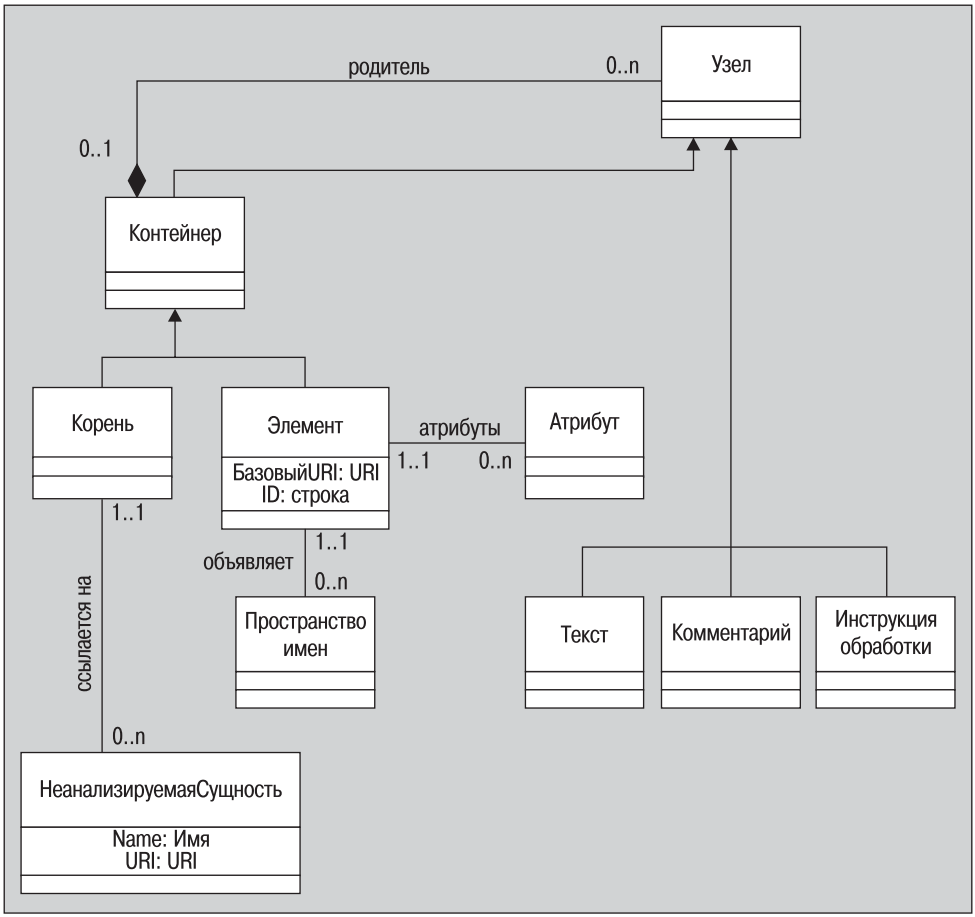


Рис. 2.7. *Дополненная диаграмма классов UML*

сутствующей строкой или между пустым набором и отсутствующим набором. В спецификации слова «неопределенный» и «пустой» используются взаимозаменяемо.

Давайте рассмотрим вкратце некоторые особенности этой модели.

Имена и пространства имен

При проектировании XSLT и XPath очень большое значение придавалось применению спецификации XML Namespaces, и хотя многие исходные документы могут мало использовать пространства имен или не использовать их совсем, понимание рекомендации XML Namespaces очень важно. Рекомендацию можно найти по адресу *http://www.w3.org/TR/REC-xml-names*.

В дополнение описания, данного в разделе «XSLT и XML» главы 1, приведем краткое изложение того, как работают пространства имен:

- Объявление пространства имен определяет префикс пространства имен и URI пространства имен. Однако префикс пространства имен должен быть уникальным только в пределах локальной области, а URI пространства имен должен быть уникальным глобально. Здесь «глобально» означает «глобально в самом широком смысле»: уникальность не только в документе, но и среди всех документов на планете. Для достижения этого можно посоветовать использовать URI, основанный на имени контролируемого вами домена, например «http://www.my-domain.com/namespace/billing». XSLT не налагает особых правил на синтаксис URI, хотя имеет смысл придерживаться стандартной схемы записи URI на случай, если это когда-либо изменится: в большинстве примеров в книге будут использоваться URI, начинающиеся с «http://». Во избежание неоднозначности, лучше также отказаться от относительных URI, подобных «billing.dtd». После горячих дебатов по этому поводу W3C выпустил запоздалый указ, не одобряющий использование относительных URI пространств имен в XML-документах и утверждающий, что их значение зависит от конкретной реализации программ. В действительности это просто означает, что они не смогли прийти к общему согласию. Однако для большинства XSLT-процессоров не имеет значения, соответствует ли URI какому-либо особому синтаксису. Например, строки «abc», «42» и «?!*» вполне приемлемы как URI пространств имен. Это просто последовательность символов, а два URI пространств имен считаются тождественными, если они содержат одинаковую последовательность символов Unicode.
- URI пространства имен не обязательно должен указывать на какой-либо ресурс, и хотя рекомендуется использовать URL, основанный на имени своего домена, не имеет никакого значения, есть ли что-нибудь интересное по данному адресу. Следующие строки: «c:\this.dtd» и «C:\THIS.DTD» – обе приемлемы как URI пространств имен, неважно, существуют или нет файлы с такими именами; и они представляют два разных пространства имен, даже при том, что как имена файлов они могли бы определять один и тот же файл.

  Тот факт, что каждая таблица стилей использует URI пространства имен `http://www.w3.org/1999/XSL/Transform`***, не подразумевает, что выполнять преобразование можно, только имея подключение к Интернету. Имя – это просто сложная константа, а вовсе не адрес, с которым процессор должен соединиться и загрузить с него что-то.***

- Запись объявления пространства имен с непустым префиксом показана ниже. Она связывает префикс пространства имен `"my-prefix"` с URI пространства имен `http://my.com/namespace`:

```
<a xmlns:my-prefix="http://my.com/namespace">
```

- Объявление пространства имен может использоваться и с пустым префиксом. Это – пространство имен по умолчанию. Следующее объявление делает URI `http://your.com/namespace` пространством имен по умолчанию:

```
<a xmlns="http://your.com/namespace">
```

- Область действия объявления пространства имен – это элемент, в котором оно появляется, и все его потомки, за исключением любого поддерева, где тот же самый префикс связан с другим URI. Эта область действия определяет, где доступен префикс. В пределах области действия любое имя с таким префиксом автоматически ассоциируется с данным URI пространства имен.
- Имя имеет три свойства: префикс, локальную часть и URI пространства имен. Если префикс – не пустой, имя написано в исходном документе в форме *prefix:local-part*; например, в имени `xsl:template` префикс – `xsl`, а локальная часть – `template`. URI пространства имен для данного имени ищется в наиболее глубоко вложенном элементе, который содержит объявление пространства имен, соответствующее префиксу. (Теоретически это позволяет использовать один и тот же префикс с разными значениями для различных частей документа, но это может понадобиться только при компоновке документа из частей, которые создавались отдельно.)
- Рекомендация XML Namespaces определяет два типа имен, которые могут уточняться с помощью префикса: имя элемента и имя атрибута. Рекомендация XSLT расширяет это на многие другие типы имен, которые встречаются в значениях атрибутов XML, например имена переменных, имена шаблонов, имена ключей и так далее. Все эти имена могут уточняться префиксом пространства имен.
- Если имя не имеет никакого префикса, то за значение URI его пространства имен принимается URI пространства имен по умолчанию – для имени элемента, или пустой URI – для имени атрибута или любого другого типа имен (например, имен переменных XSLT и имен шаблонов). Однако в пределах выражения XPath URI пространства имен по умолчанию никогда не используется для имен, не содержащих префикса, даже если это – имя элемента.
- Два имени считаются эквивалентными, если они имеют одинаковую локальную часть и один и тот же URI пространства имен. Комбинация локальной части с URI пространства имен называется **расширенным именем**. Расширенное имя практически никогда не пишется, и для его написания нет определенного синтаксиса; это некоторое условное значение, сформированное из двух компонентов.

  Интерфейс прикладного программирования TrAX, описанный в приложении F, позволяет записывать расширенные имена в виде «`{namespace.uri}local-name`».

- Префикс имени не участвует в сравнении имен и, в этом смысле, может выбираться произвольно; однако он доступен приложениям, поэтому может использоваться по умолчанию префикс в конечном дереве или в диагностических сообщениях.

В древовидной модели XPath существует два способа сделать видимыми объявления пространства имен:

- Для любого узла, такого как элемент или атрибут, все три компонента имени (префикс, локальная часть и URI пространства имен) доступны че-

рез функции name(), local-name() и namespace-uri(). Приложение не должно знать и не может выяснить, где было объявлено соответствующее пространство имен. Префикс фактически не сохраняется в дереве как часть имени элемента или атрибута, но он может быть восстановлен посредством вызова функции name(). В некоторых ситуациях (например, если документ использует несколько префиксов, относящихся к одному и тому же URI пространства имен) процессор может использовать не первоначальный префикс, а другой префикс, который обозначает тот же самый URI пространства имен. В ряде случаев – обычно, когда исходный документ использует один и тот же префикс для ссылок на несколько разных пространств имен, процессор может выбирать произвольный префикс.

- Для любого элемента можно определить все действующие для него объявления пространств имен, отыскивая связанные с ним узлы пространств имен. Они доступны, как если бы объявления пространств имен были повторены для данного конкретного элемента. Приложение, опять же, не может определить, где именно в исходном документе было произведено объявление пространства имен. Имя узла пространства имен является префиксом пространства имен, который был первоначально записан в исходном документе: в отличие от префикса, возвращаемого функцией name(), процессор не имеет права изменять его.

Хотя объявления пространств имен первоначально записаны в исходном документе в виде атрибутов XML, они не сохраняются в дереве как узлы атрибутов и не могут быть найдены среди всех узлов атрибутов. Точно так же невозможно генерировать узлы пространств имен в конечном дереве путем создания атрибутов с именами «xmlns:*», зарезервированными для объявлений пространств имен.

Идентификаторы

Идентификатор (ID) – это строковое значение, которое однозначно задает узел элемента в пределах документа. Если элемент имеет идентификатор, то обращаться к этому элементу очень просто и (есть надежда) эффективно, если значение идентификатора известно. Идентификатор всегда записывается как значение атрибута, объявленного в DTD атрибутом типа ID. Каждый элемент имеет не более одного идентификатора, и каждое значение идентификатора (если оно вообще есть) задает один элемент.

Например, в наборе данных XML, содержащем информацию о служащих, каждый элемент <сотрудник> мог бы иметь уникальный атрибут инн с индивидуальным номером налогоплательщика. Например:

```
<персонал>
<сотрудник инн="ИНН-123456789123">1
   <имя>Сидоров Иван Петрович</имя>
   ...
</сотрудник>
<сотрудник инн="ИНН-123456789124">
   <имя>Петрова Екатерина Ивановна</имя>
```

```
    ...
</сотрудник>
</персонал>
```

Поскольку атрибут инн уникален, он может быть объявлен в DTD как ID-атрибут посредством следующего объявления:

```
<!ATTLIST сотрудник инн ID #REQUIRED>
```

Атрибуты типа ID часто называют идентификаторами, подчеркивая их роль; к сожалению, иногда это вводит в заблуждение, что имя ID-атрибута должно быть каким-то особым. Это не так; ID-атрибут – это любой атрибут, определенный в DTD как атрибут типа ID, независимо от его имени.

Значение идентификатора должно соответствовать форме **имени** XML. Это означает, например, что оно должно начинаться с буквы и что оно не может содержать символы типа «/» или пробела.

В XML атрибуты также могут быть определены как атрибуты типа `IDREF` или `IDREFS`, если они содержат идентификаторы, используемые для указания на другие элементы в документе (атрибут `IDREF` содержит один идентификатор, атрибут `IDREFS` содержит набор идентификаторов, разделенных пробелом). Однако XPath никак не использует эту информацию. В языке XPath есть функция `id()` (см. соответствующий раздел в главе 7), с помощью которой можно найти элемент, имеющий данное значение идентификатора. Эта функция написана таким образом, что ей можно передавать атрибуты `IDREF` или `IDREFS`, так же как и любую другую строку с идентификатором. Поэтому атрибуты `IDREF` и `IDREFS` не встречаются в древовидной модели в явном виде.

Использование идентификаторов вносит некоторое осложнение, поскольку XPath не ограничивается обработкой только действительных XML-документов. Если XML-документ правильно построен (или просто сбалансирован), но не является действительным XML-документом, то предполагаемые идентификаторы могут дублироваться, и они могут не удовлетворять синтаксическим правилам для имен XML. Согласно спецификации XPath, если значение идентификатора появляется более одного раза, все его появления, кроме первого, игнорируются. Там не говорится особо о тех случаях, когда значение идентификатора содержит запрещенные символы, например пробелы, но вполне вероятно, что в этом случае функция `id()` будет не в состоянии найти элемент, хотя в остальном, кажется, будет работать правильно. При использовании идентификаторов, вероятно, стоит задействовать проверяющий анализатор XML, чтобы предотвратить такие ситуации.

Кроме такого особого использования ID-атрибутов, тип атрибутов, определенный в DTD, не является компонентом древовидной модели XPath. Нельзя выяснить, например, тип конкретного атрибута. Кроме того, ID-атрибуты могут встречаться только в дереве, полученном при анализе исходного документа. Они не могут встретиться во временном или в конечном дереве, потому что временные и конечные деревья не имеют определений DTD.

XSLT также предоставляет более гибкий подход к отысканию элементов (или других узлов) по содержимому, а именно – ключи. С помощью ключей

можно достичь всего, что возможно с идентификаторами, кроме обеспечения уникальности. Ключи объявляются в таблице стилей с помощью элемента <xsl:key> (см. главу 4, раздел «xsl:key»), и их можно использовать для отыскания узлов через функцию key() (см. главу 7, раздел «key»).

Символы

В определении **информационного множества XML (XML Information Set)** (*http://www.w3.org/TR/xml-infoset*) каждый индивидуальный символ считается объектом (или **информационным элементом**). Концептуально, это полезная модель, так как она позволяет говорить о свойствах символа и о его положении относительно других символов, но представлять каждый символ как отдельный объект в настоящей реализации дерева было бы слишком дорогим удовольствием.

В модели XPath не принято представлять символы как узлы. Если бы это было не так, тогда синтаксис XPath можно было бы расширить естественным путем для манипуляции символами в строках, но вместо этого проектировщики ввели отдельный набор функций для строковых манипуляций. Эти функции описаны в главе 7.

Строковое значение любого узла – последовательность символов, количество которых может быть больше или равно нулю. Каждый символ соответствует определению Char из стандарта XML. Можно сказать, что это символ Unicode, а точнее, одно из следующих:

- Один из четырех пробельных символов: табуляция #x9, перевод строки #xA, возврат каретки #xD или пробел #x20.
- Обыкновенный 16-разрядный символ Unicode в диапазоне от #x21 до #xD7FF или от #xE000 до #xFFFD.
- Расширенный символ Unicode в диапазоне от #x10000 до #x10FFFF. В процессоре такой символ обычно представляется как **замещающая пара (surrogate pair)**, состоящая из двух 16-разрядных кодов в диапазоне от #xD800 до #xDFFF, но XPath рассматривает это как один символ, а не два. Это оказывает влияние на функции, которые подсчитывают символы в строке или используют положение символа в строке, в частности функции string-length(), substring() и translate(). Здесь язык XPath отличается от языка Java, который рассматривает **замещающую пару** как два символа. (**Замещающие пары** Unicode в настоящее время не используются широко, но вероятно, что будут чаще встречаться в будущем.)

Заметим, что концы строк приводятся к символу перехода на новую строку #xA, независимо от того, в каком виде они присутствуют в исходном XML-файле.

В таблице стилей невозможно определить, как символ был записан в исходном файле. Например, с точки зрения модели данных XPath все следующие строки идентичны:

- >
- >

- >
- >
- >
- <![CDATA[>]]>

Анализатор XML поддерживает такое различное представление символов. В большинстве реализаций XSLT-процессоров используется стандартный анализатор XML, и XSLT-процессор не может отличать эти представления, даже если потребуется, потому что для анализатора они все эквивалентны.

Единственное исключение из этого правила – в вопросах обработки пробельных символов. Способ обработки пробельных символов в исходном XML-документе и при этом взаимодействие анализатора XML с XSLT-процессором – удивительно сложные темы, и они будут более детально обсуждаться в главе 3.

На временном дереве (описанном позже в этой главе в разделе «Временные деревья») для каждого символа текстового узла существует дополнительное свойство: флажок «запрещение экранирования выходных данных» (disable-output-escaping). При выводе с использованием инструкций <xsl:text> или <xsl:value-of> есть возможность установить флажок запрещения экранирования выходных данных, задав атрибут «disable-output-escaping="yes"»; это приведет к тому, что специальные символы, например амперсанд («&»), будут выводиться в виде «&», а не в виде «&», как они обычно выводятся. При направлении вывода во временное дерево этот флажок привязывается к каждому написанному символу, чтобы позднее, когда текстовый узел копируется из временного дерева в другое дерево посредством инструкций <xsl:copy> или <xsl:copy-of>, флажок тоже был скопирован вместе с символом. (Конечно, большинство реализаций вряд ли будут сохранять флажок с каждым символом, существуют и более эффективные пути выполнения этого. Но все равно эта особенность усложняет работу, и поскольку запрещение экранирования выходных данных является в стандарте необязательным, в некоторых продуктах оно, вероятно, не будет реализовано в полной мере.)

Что не включено в древовидную модель?

Дебаты по определению древовидной модели велись в отношении того, что можно не учитывать. Какая информация из исходного XML-документа является существенной, а какая – незначительной? Например, имеет ли значение то, что для записи текста использовалась нотация CDATA? Имеют ли значение границы сущности? Как насчет комментариев?

Многие новички в XSLT задают вопросы такого типа: «Как заставить процессор заключать значения атрибутов в одинарные кавычки, а не в двойные?» или: «Как получать на выводе « », вместо « »?», а ответ один: это невозможно, потому что считается, что такие вещи не должны заботить получателя документа, и поэтому они не были включены в древовидную модель XPath.

Вообще, особенности XML-документа можно разбить на три категории: явно существенные, явно незначащие и спорные. Например, порядок элементов явно существен; порядок атрибутов в пределах открывающего тега элемента явно несуществен, а значимость комментариев спорна.

Сам стандарт XML не очень четко определяет эти различия. Он определяет некоторые вещи, которые должны сообщаться приложению, и они, конечно, существенны. Есть и другие вещи, которые явно значимы (например, порядок элементов), но относительно которых стандарт ничего не говорит. Точно так же некоторые вещи в стандарте четко определены как незначащие, например выбор символов `CR-LF` или `LF` для окончаний строки, а о многих других он хранит молчание: например, о выборе между одинарными и двойными кавычками для заключения в них значений атрибутов.

В результате различные стандарты в семействе XML принимают каждый свое решение в этих вопросах, в том числе XSLT и XPath.

Разногласия возникают отчасти потому, что существует два вида приложений. Приложения, которые нацелены на извлечение информации из документа, обычно заинтересованы только в основном содержимом. Приложения, подобные инструментальным средствам для редактирования XML, заинтересованы также и в деталях того, как был написан XML, потому что когда пользователи не делают никаких изменений в разделах документа, они хотят, чтобы выходной документ как можно ближе соответствовал оригиналу.

Чтобы разрешить эти разногласия и прийти к некоторой общности в различных стандартах, W3C предпринял инициативу, цель которой – определить общую модель информации в XML-документе, так называемое информационное множество XML (XML Information Set), или **InfoSet**, как его часто называют. Текущие результаты публикуются на веб-странице *http://www.w3.org/TR/xml-infoset*. Самая последняя версия на момент написания книги датирована 2 февраля 2001 года; спецификация претерпела множество изменений и может измениться еще несколько раз, прежде чем будет принят окончательный вариант.

Информационное множество XML в сегодняшнем виде включает семнадцать различных типов информационных элементов, перечисленных ниже. В более ранних версиях спецификации Infoset некоторые из них были классифицированы как основные типы информации, а другие, соответственно, как второстепенные. Однако эта классификация, очевидно, оказалась слишком спорной, потому что в самой последней текущей версии спецификации от нее отказались.

Перечислим семнадцать типов информационных элементов:

- Ранее классифицируемые как основные: Документ, Элемент, Атрибут, Инструкция обработки, Нерасширенная сущность, Символ, Нотация, Объявление пространства имен.
- Ранее классифицируемые как второстепенные: Комментарий, Объявление типа документа, Внутренняя сущность, Внешняя сущность, Неана-

лизируемая сущность, Начальный маркер сущности, Конечный маркер сущности, Начальный маркер CDATA, Конечный маркер CDATA.

С введением информационного множества была предпринята попытка разложить содержимое XML-документов на три категории:

- Основные информационные элементы, в которых будет заинтересовано большинство приложений.
- Второстепенные информационные элементы, в которых могут быть заинтересованы некоторые приложения.
- Лексические подробности, которые не несут информацию (например, пробельные символы между значениями атрибутов в открывающем теге).

Похоже, что было очень трудно прийти к соглашению в этих вопросах, поэтому текущая спецификация предлагает только две категории: информацию, которая включена в Infoset, и информацию, которая исключена. И даже это не звучит как четкая директива. Единственные правила, которые она налагает, заключаются в том, что другие спецификации должны объяснять, какие информационные элементы делаются доступными приложению, а какие – нет.

Другая попытка определить основное информационное содержимое XML-документа отражена в спецификации Канонического XML (*http://www.w3.org/TR/xml-c14n*). Эта спецификация рассматривает данную проблему под другим углом: она пытается определить правила для решения вопроса, когда два лексически различных XML-документа имеют одинаковое информационное содержимое. Для этого она определяет преобразование, которому можно подвергнуть любой XML-документ, чтобы привести его к канонической форме; а если два документа имеют одинаковую каноническую форму, они считаются эквивалентными.

Процесс преобразования документа в каноническую форму можно коротко изложить следующим образом:

- Документ кодируется в UTF-8.
- Концы строк нормализуются в `#xA`.
- Значения атрибутов нормализуются в зависимости от типа атрибута.
- Раскрываются ссылки на символы и ссылки на анализируемые сущности.
- Разделы CDATA заменяются их символьным содержимым.
- Удаляется объявление XML и объявление типа документа.
- Теги пустых элементов (`<a/>`) преобразуются в пары (`<a></a>`).
- Нормализуются пробельные символы вне элемента документа и в пределах тегов.
- Ограничители значений атрибутов заменяются на двойные кавычки.
- Специальные символы в значениях атрибутов и символьном содержимом заменяются ссылками на символы.
- Удаляются избыточные объявления пространств имен.

- Атрибуты, для которых в DTD определено значение по умолчанию, дописываются ко всем соответствующим элементам.
- Атрибуты и объявления пространств имен сортируются в алфавитном порядке.

В этой спецификации также есть неопределенность: может или не может каноническая форма сохранять комментарии из исходного документа.

Таким образом, вместо определения Infoset появилось альтернативное определение основного информационного содержимого XML, а именно информация, которая остается после преобразования документа в каноническую форму.

На диаграмме (рис. 2.8) проиллюстрирована получившаяся классификация: центральное ядро – это информация, которая сохраняется в канонической форме; «периферийное» кольцо – информация, которая входит в Infoset, но не входит в канонический XML; в то время как внешнее кольцо

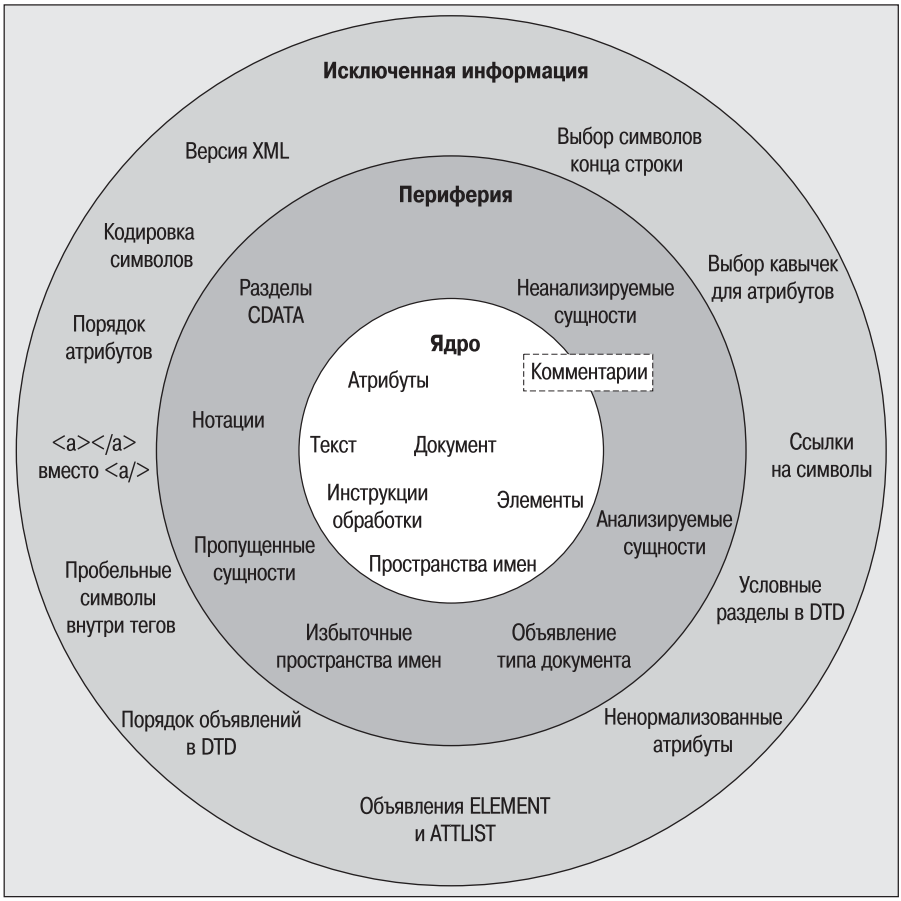


***Рис. 2.8.** Результирующая схема классификации информационного содержимого*

представляет особенности XML-документа, которые совсем исключены из информационного множества.

Выбор информационных элементов, которые присутствуют в модели дерева XSLT/XPath и которые поэтому доступны для таблиц стилей XSLT, довольно близко соответствует «ядру» на приведенной диаграмме, но есть и некоторые незначительные различия:

- Модель XSLT включает комментарии, которые считаются необязательными типами информации в каноническом XML. Это может привести к проблемам, когда XSLT-процессор основан на существующем программном обеспечении. Например, совместимый с SAX1 анализатор XML может считать комментарии второстепенной информацией и не передавать их. К счастью, это устранено в SAX2.
- Канонический XML сохраняет префиксы пространств имен, как первоначально определено в исходном документе. Большинство доступных сейчас XSLT-процессоров также сохраняет префиксы пространства имен, но теоретически, согласно спецификации, функция `name()` в XPath может использовать любой префикс пространства имен, соответствующий правильному URI пространства имен.
- Модель XSLT сохраняет базовый URI как свойство узла, но это свойство не сохраняется в каноническом XML.

Способы моделирования текстового содержимого объектов в XPath и Infoset различны: Infoset описывает каждый символ как отдельный информационный элемент, в то время как модель XPath сцепляет смежные символы в так называемое строковое значение узла. В случае символьного содержимого в пределах элементов это требует включения в модель также и текстовых узлов. Однако разница только в способах описания данных, а информационное содержимое в обеих моделях одно и то же. Причина, по которой Infoset делает каждый символ отдельным информационным элементом, в том, что это позволяет поместить между символами границы разделов CDATA и границы ссылок на сущности. Эти информационные элементы не сохраняются в каноническом XML, но они невидимы и в модели XPath.

Управление сериализацией

Процессор преобразования, который генерирует конечное дерево, обычно позволяет пользователям управлять выводом только основных информационных элементов и свойств (а также комментариев). Процессор вывода, или сериализатор, дает еще некоторый дополнительный контроль над тем, как именно дерево результатов преобразуется в последовательный XML-документ. В частности это позволяет управлять:

- Использованием разделов `CDATA`
- Версией XML
- Кодировкой символов
- Свойством `standalone` в XML-объявлении
- Объявлением `DOCTYPE`

Некоторые из этих вещей рассматриваются в приведенной выше классификации как второстепенные, а некоторые входят в исключенную из информационного множества категорию. Управление над процессом сериализации распространяется не на все второстепенные информационные элементы (например, невозможно генерировать ссылки на сущности) и, конечно, не распространяется на все исключенные информационные элементы. Например, нет возможности управлять порядком, в котором записываются атрибуты, или выбором между <a/> и <a> для представления пустых элементов, размещением пробельных символов в пределах открывающего тега или наличием символа новой строки в конце документа.

Говоря вкратце, набор вещей, которыми можно управлять на стадии сериализации, имеет некоторое соответствие классификациям информационных элементов информационного множества и канонического XML, но не такое большое, как можно было ожидать. Возможно, если бы информационное множество было определено ранее, различные спецификации W3C были бы более согласованными.

Чтобы подытожить сказанное, перечислим некоторые вещи, которые отсутствуют в исходном дереве, и некоторые вещи, которыми нельзя управлять в выходном XML-файле.

Невидимые различия

В приведенной ниже таблице конструкции в двух столбцах считаются эквивалентными, и в каждом случае автору таблицы стилей трудно определить, которая из них использовалась в исходном документе. Если одна из них по какой-то причине не подходит, можно и не пробовать другую – эффект будет тот же:

Конструкция	Эквивалент
<item/>	<item></item>
>	>
<e>"</e>	<e>"</e>
<![CDATA[a < b]]>	a < b
<b/>	<b xmlns="one.uri"/>
<rectangle x="2" y="4"/>	<rectangle y='4' x='2'/>

Во всех этих случаях, кроме CDATA, нет никакого контроля над форматом вывода. Поскольку варианты эквивалентны, неважно, который из них используется.

Почему сделано исключение для CDATA? Возможно, потому, что в случаях, когда преобразуемый текст содержит большое количество специальных символов, например в книге, где приводятся примеры по XML, использование ссылок на символы может затруднить чтение получившегося документа. В конце концов, это одно из главных достоинств XML и одна из причин

его успеха, что XML-документы легко читать и редактировать вручную. Еще одна возможная причина в том, что на самом деле существуют некоторые разногласия по поводу значения CDATA: например, возникали споры, допустима ли запись типа «<![CDATA[ ]]>» в местах, где XML позволяет находиться только пробельным символам.

Информация DTD

Проектировщики XPath решили не включать всю информацию DTD в дерево. Возможно, они предвидели появление схем XML, которые, как ожидается, заменят DTD и которые представляют информацию о логической структуре в форме XML, что позволяет использовать ту же самую древовидную модель.

XSLT-процессор (но не приложение) должен знать, какие атрибуты имеют тип ID, чтобы при использовании функции id() могли быть найдены соответствующие элементы. Информация о том, что некоторому элементу присвоен конкретный идентификатор, является частью древовидной модели, но кроме этого никакая другая информация о типе атрибутов в ней не присутствует.

Процесс преобразования

Было показано, насколько важен процесс, выполняемый XSLT, для преобразования исходного дерева в конечное дерево под управлением таблицы стилей и рассмотрено строение этих деревьев. Теперь пора обсудить, как фактически протекает процесс преобразования, т. е. заглянуть в саму таблицу стилей.

Шаблонные правила

Как обсуждалось в главе 1, большинство таблиц стилей содержит ряд шаблонных правил. Каждое шаблонное правило представлено в таблице стилей как элемент <xsl:template> с атрибутом match. Значение атрибута match является образцом. Этот образец определяет, каким узлам в исходном дереве соответствует шаблонное правило.

Например, образец «/» соответствует корневому узлу; образец «title» соответствует любому элементу <title>, а образец «chapter/title» соответствует любому элементу <title>, родителем которого является элемент <chapter>.

Когда XSLT-процессор вызывается, чтобы применить конкретную таблицу стилей к конкретному исходному документу, то первое, что он делает – производит чтение и анализ этих документов и создает в памяти их внутреннее древовидное представление. По завершении этой подготовительной работы можно начинать процесс преобразования.

Первым шагом процесса преобразования является поиск шаблонного правила, которое соответствует корневому узлу исходного дерева. Если найдено несколько возможных кандидатов, существуют правила разрешения кон-

фликтов, помогающие выбрать наиболее подходящего из них (см. раздел «Правила разрешения конфликтов» этой главы). Если не найдено никаких шаблонных правил, соответствующих корневому узлу, используется встроенный шаблон. Затем XSLT-процессор применяет это шаблонное правило.

Содержимое элемента <xsl:template> в таблице стилей представляет собой последовательность элементов и текстовых узлов. Комментарии и инструкции обработки в таблице стилей игнорируются, так же как и текстовые узлы, состоящие из пробельных символов, если только они не принадлежат элементу <xsl:text> или элементу с соответствующим атрибутом xml:space. Будем называть эту последовательность элементов и текстовых узлов **телом шаблона**. В спецификации XSLT ее называют просто шаблоном, но этот термин часто звучит неоднозначно, поэтому в книге он избегается.

Элементы в теле шаблона классифицируются как инструкции или как данные, в зависимости от их пространств имен. Текстовые узлы всегда классифицируются как данные. При применении шаблона инструкции, содержащиеся в теле шаблона, выполняются, а узлы данных копируются в конечное дерево. Элементы, которые классифицируются как данные, официально называются **конечными литеральными элементами**.

Содержимое тела шаблона

Рассмотрим следующее шаблонное правило:

```
<xsl:template match="/">
   <xsl:message>Запущена!</xsl:message>
   <xsl:comment>Создано при помощи XSLT</xsl:comment>
   <html>
      . . .
   </html>
   Конец
</xsl:template>
```

Тело этого шаблона состоит из двух инструкций (<xsl:message> и <xsl:comment>), конечного литерального элемента (элемента <html>) и некоторого текста («Конец»). Когда этот шаблон применяется, инструкции выполняются согласно правилам для каждой отдельной инструкции, а конечные литеральные элементы и текстовые узлы копируются (как узлы элементов и текстовые узлы, соответственно) в конечное дерево.

Проще всего рассматривать это как последовательный процесс, в котором применение тела шаблона заключается в применении каждого из его компонентов в том порядке, в каком они перечислены. На самом деле, поскольку XSLT в значительной степени свободен от побочных эффектов, они могут выполняться в другом порядке или параллельно. Важно понимать, что после применения этого тела шаблона конечное дерево будет содержать ниже его корневого узла узел комментария (сформированный инструкцией <xsl:comment>), узел элемента <html> (сформированный из конечного литерального элемента <html>) и текстовый узел «Конец».

Фактически <xsl:message> *– исключение из правила, подразумевающего, что язык XSLT свободен от побочных эффектов. Если в теле шаблона содержится несколько инструкций* <xsl:message>*, то последовательность, в которой должны появиться сообщения, не определена.*

Если бы в содержимое элемента <html> не было включено «...», на этом бы все и кончилось. Но когда применяется конечный литеральный элемент типа <html>, его содержимое тоже обрабатывается как тело шаблона, что происходит и в данном случае. Здесь снова могут содержаться инструкции, конечные литеральные элементы и текст.

Вложенные тела шаблонов

Предположим теперь, что шаблонное правило выглядит так:

```
<xsl:template match="/">
   <xsl:message>Запущена!</xsl:message>
   <xsl:comment>Создано при помощи XSLT</xsl:comment>
   <html>
      <head>
         <title>Моя первая сгенерированная HTML-страница</title>
      </head>
      <body>
         <xsl:apply-templates/>
      </body>
   </html>
   Конец
</xsl:template>
```

Здесь элемент <html> содержит два дочерних элемента, <head> и <body>. Оба они – конечные литеральные элементы, поэтому их применение состоит в копировании их из таблицы стилей в конечное дерево. Элемент <head> содержит конечный литеральный элемент <title>, в котором находится некоторый текст, поэтому вся конструкция копируется в конечное дерево.

Однако с элементом <body> дело обстоит иначе: он содержит инструкцию XSLT, а именно <xsl:apply-templates/>. Эта инструкция имеет особое значение, когда она записана, как здесь, без всяких атрибутов. Это означает, что нужно выбрать всех потомков текущего узла в исходном дереве, найти для каждого соответствующее шаблонное правило в таблице стилей и применить его.

Что в действительности произойдет в этом случае, зависит от того, что находится в исходном документе, и от того, какие другие шаблонные правила есть в таблице стилей. Обычно, поскольку сейчас обрабатывается корневой узел дерева исходного документа, он имеет только один дочерний узел – элемент документа (самый внешний элемент исходного XML-документа). Предположим, что это элемент <doc>. Тогда XSLT-процессор будет искать в таблице стилей шаблонное правило, соответствующее элементу <doc>.

В самом простом случае найдется только одно правило, соответствующее этому элементу, например такое:

```
<xsl:template match="doc">
```

Если же найдется более одного соответствующего шаблонного правила, снова должны использоваться правила разрешения конфликтов для выбора наиболее подходящего из них. Другая возможная ситуация – отсутствие соответствующего шаблонного правила. В этом случае вызывается встроенное шаблонное правило для узлов элемента, которое просто выполняет инструкцию <xsl:apply-templates/>; другими словами, выбираются потомки этого элемента и отыскиваются соответствующие им шаблонные правила. Имеется также встроенное шаблонное правило для текстовых узлов, которое заключается в копировании текстового узла в конечное дерево. Если элемент не имеет потомков, то <xsl:apply-templates/> не делает ничего.

Форсированная обработка

Таким образом, простейший способ обработки исходного дерева заключается в создании шаблонного правила для каждого типа узлов, которые могут встретиться, чтобы эти шаблонные правила генерировали любой требуемый вывод, а также вызывали инструкцию <xsl:apply-templates> для обработки потомков текущего узла.

Пример: Форсированная обработка

Здесь приведен пример таблицы стилей, которая как раз делает это.

Исходный документ

Исходный документ, книги.xml – это простой книжный каталог:

```
<?xml version="1.0"?>
<книги>
  <книга категория="справочник">
      <автор>Найджел Рис</автор>
      <название>Поговорки века</название>
      <цена>8.95</цена>
   </книга>
   <книга категория="беллетристика">
      <автор>Ивлин Во</автор>
      <название>Меч чести</название>
      <цена>12.99</цена>
   </книга>
   <книга категория="беллетристика">
      <автор>Герман Мильвиль</автор>
      <заголовок>Моби Дик</заголовок>
      <цена>8.99</цена>
   </книга>
   <книга категория="беллетристика">
      <автор>Дж. Р. Р. Толкиен</автор>
      <название>Властелин колец</название>
      <цена>22.99</цена>
   </книга>
</книги>
```

Таблица стилей

Предположим, требуется отобразить эти данные в форме нумерованного списка книг. Для этих целей подойдет следующая таблица стилей, книги.xsl:

```
<xsl:stylesheet
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
         version="1.0"
>
<xsl:template match="книги">
   <html><body>
   <h1>Список книг</h1>
   <table width="640">
   <xsl:apply-templates/>
   </table>
   </body></html>
</xsl:template>

<xsl:template match="книга">
<tr>
   <td><xsl:number/></td>
   <xsl:apply-templates/>
   </tr>
</xsl:template>

<xsl:template match="автор | название | цена">
   <td><xsl:value-of select="."/></td>
</xsl:template>

</xsl:stylesheet>
```

Что происходит в этом случае? Для корневого узла нет специального шаблона, поэтому вызывается встроенный шаблон. Он обрабатывает всех потомков корневого узла.

У корневого узла есть только один дочерний элемент – <книги>, следовательно, шаблон применяется к элементу <книги>. При этом в конечное дерево выводятся некоторые стандартные HTML-элементы, а затем вызывается инструкция <xsl:apply-templates/>, чтобы обработать непосредственных потомков элемента <книги>. Ими являются элементы <книга>, поэтому все они обрабатываются в соответствии с шаблонным правилом, образцом соответствия которого является «match="книга"». Это шаблонное правило выводит HTML-элемент <tr>, а внутри него – элемент <td>, который заполняется инструкцией <xsl:number/>, задача которой – получить порядковый номер текущего узла (элемента <книга>) в пределах его корневого элемента. Для этого еще раз вызывается инструкция <xsl:apply-templates/>, чтобы обработать дочерние элементы элемента <книга> в исходном дереве.

Непосредственными потомками элемента <книга> в исходном документе являются элементы <автор>, <название> или <цена>, так что все они соответ-

ствуют шаблонному правилу, чей образец соответствия – «match="автор | название | цена"» (знак «|» здесь читается как «или»). Это шаблонное правило выводит HTML-элемент <td>), который заполняется при выполнении инструкции <xsl:value-of select="."/>. Данная инструкция вычисляет выражение XPath и записывает его результат (строку) в виде текста в конечное дерево. Выражение «.» возвращает строковое значение текущего узла, которым является текстовое содержимое текущего элемента <автор>, <цена> или <название>.

Этот шаблон больше не вызывает инструкцию <xsl:apply-templates>, поэтому потомки обрабатываемых узлов не обрабатываются, и управление возвращается на уровень выше.

Конечный документ

```
<html>
   <body>
      <h1>Список книг</h1>
      <table width="640">
         <tr>
            <td>1</td>
            <td>Найджел Рис</td>
            <td>Поговорки века</td>
            <td>8.95</td>
         </tr>
         <tr>
            <td>2</td>
            <td>Ивлин Во</td>
            <td>Меч чести</td>
            <td>12.99</td>
         </tr>
   и т. д...
         </table>
   </body>
</html>
```

Этот стиль обработки называется **форсированной** обработкой (**push** processing). Процессор как будто выставляет узлы за дверь, предлагая всем заинтересованным заняться ими.

Выбор узлов для обработки

Простая форсированная обработка очень хороша, когда выходные данные должны иметь ту же самую структуру и последовательность, что и входные; и все, что требуется сделать – это добавить или удалить несколько тегов или произвести по ходу дела другое простое редактирование значений.

В вышеупомянутом примере дело бы осложнилось, если бы свойства каждой книги были менее предсказуемы, например, если бы некоторые книги не имели цены или если бы название книги и автор могли появляться в любом

порядке. В этом случае генерируемая таблица HTML не была бы столь же хорошо разбита на столбцы, потому что создание новой ячейки для каждого свойства, которое встречается, может оказаться неверным решением.

В таких обстоятельствах существует два варианта действий:

- Можно более определенно указать, какие узлы следует обрабатывать, а не запускать *обработку всех потомков текущего узла*.
- Можно более определенно указать, как именно их обрабатывать, а не просто надеяться на *выбор наиболее подходящего шаблонного правила*.

Рассмотрим первую возможность.

Пример: Управление последовательностью обработки

Можно получить большую степень управления выбором узлов, **которые** подлежат обработке, изменив шаблон <книга> в файле книги.xsl следующим образом:

```
<xsl:template match="книга">
   <tr>
   <td><xsl:number/></td>
   <xsl:apply-templates select="автор"/>
   <xsl:apply-templates select="название"/>
   <xsl:apply-templates select="цена"/>
   </tr>
</xsl:template>
```

Вместо выбора сразу всех дочерних элементов и поиска соответствующего шаблонного правила для каждого новое правило явно требует сначала выбирать дочерний элемент <автор>, потом дочерний элемент <название>, а после этого – дочерний элемент <цена>.

Это тоже будет работать, причем более надежно, чем предыдущая попытка, но все еще будет производить не идеальную таблицу, если попадется любой элемент <книга>, без элемента <автор>, например, или имеющий более одного автора.

Поскольку наша задача – добиться правильной структуры на выходе и поскольку о структуре исходного документа известно довольно много, в этой ситуации, вероятно, можно выиграть, определив всю обработку в шаблоне <книга>, а не полагаясь на шаблоны, соответствующие его дочерним элементам.

Пример: Выбор узлов явным образом

Можно получить большую степень управления тем, **как** узлы будут обрабатываться, если написать шаблон <книга> следующим образом:

```
<xsl:template match="книга">
   <tr>
```

```
      <td><xsl:number/></td>
      <td><xsl:value-of select="автор"/></td>
      <td><xsl:value-of select="название"/></td>
      <td><xsl:value-of select="цена"/></td>
      </tr>
   </xsl:template>
```

Некоторые называют это **извлекающей** обработкой **(pull** processing), потому что шаблон не выталкивает узлы за дверь, чтобы их обрабатывали другие шаблоны, а извлекает и обрабатывает их сам.

Стиль обработки, основанный на сопоставлении с образцами, или форсированный, – наиболее характерная особенность XSLT, и он очень хорошо работает в приложениях, где имеет смысл отдельно описывать обработку каждого типа узла в исходном документе. Однако существует много других методов, тоже не менее ценных. Из шаблонного правила для одного конкретного узла можно несколькими способами получить доступ к информации в других узлах:

- Вызвать инструкцию <xsl:apply-templates>, чтобы обработать узлы, используя соответствующие им шаблонные правила.
- Вызвать инструкцию <xsl:apply-templates> в особом **режиме** (см. ниже) для обработки узлов с использованием шаблонных правил для соответствующего режима.
- Вызвать инструкцию <xsl:value-of>, чтобы извлечь нужную информацию прямо из узлов.
- Вызвать инструкцию <xsl:for-each>, чтобы произвести обработку каждого из узлов по очереди.
- Вызвать инструкцию <xsl:call-template>, чтобы активизировать конкретный шаблон по имени, а не полагаться на выбор шаблона путем сопоставления с образцами.

Дальнейшее обсуждение различных подходов к созданию таблиц стилей включено в главу 9 «Образцы проектирования таблиц стилей».

Режимы

Иногда возникает необходимость обрабатывать один и тот же узел в исходном дереве более одного раза различными способами. Классическим примером могут служить оглавления. При создании оглавления названия разделов обрабатываются одним способом, а при создании тела документа – другим.

Одним способом решения этой задачи является использование в одном проходе форсированной обработки, а во всех других случаях – извлекающей обработки. Однако такой подход может оказаться слишком сковывающим. Вместо этого можно определить для каждого прохода разные режимы обработки. При вызове инструкции <xsl:apply-templates> можно передать название режима обработки, и тогда будут рассматриваться только те шаблонные

правила, которые определены для того же самого режима. Например, можно указать:

```
<xsl:apply-templates select="заголовок-1" mode="оглавление"/>
```

В этом случае выбранное шаблонное правило могло бы быть определено так:

```
<xsl:template match="заголовок-1" mode="оглавление">
. . .
</xsl:template>
```

Вопросы применения режимов подробно обсуждаются в разделе «Использование режимов» главы 4, а пример их использования при создании оглавления приведен в разделе «Оглавление» главы 10.

Встроенные шаблонные правила

Что происходит, когда для обработки узла вызывается инструкция <xsl:apply-templates>, а в таблице стилей нет никакого шаблонного правила, соответствующего этому узлу?

Вызывается **встроенное шаблонное правило**.

Для каждого типа узлов имеются встроенные шаблонные правила. Они работают следующим образом:

Тип узла	Встроенное шаблонное правило
корневой узел	Вызывает инструкцию <xsl:apply-templates>, чтобы обработать дочерние элементы корневого узла в режиме, который установлен в вызывающем шаблонном правиле
элемент	Вызывает инструкцию <xsl:apply-templates>, чтобы обработать дочерние элементы этого узла в режиме, который установлен в вызывающем шаблонном правиле
атрибут	Копирует значение атрибута в конечное дерево как текст, а не как узел атрибута
текст	Копирует текст в конечное дерево
комментарий	Не делает ничего
инструкция обработки	Не делает ничего
пространство имен	Не делает ничего

Встроенные шаблонные правила вызываются только в том случае, если в таблице стилей нет никакого правила, соответствующего узлу.

Нет возможности переопределить встроенный шаблон для узлов пространств имен, так как не существует образца, соответствующего узлу пространства имен. Если для обработки узлов пространств имен вызвать инструкцию <xsl:apply-templates>, то ничего не произойдет. Для обработки всех узлов пространств имен текущего элемента используйте инструкцию:

```
<xsl:for-each select="namespace::*">
```

Правила разрешения конфликтов

Что произойдет, когда, напротив, есть более одного шаблонного правила, соответствующего заданному узлу? Как упоминалось ранее, в игру входит стратегия разрешения конфликтов.

Это работает следующим образом:

- Сначала рассматривается **приоритет импортирования (import precedence)** каждого правила. Как будет показано в главе 3, одна таблица стилей может импортировать другую, используя элемент <xsl:import>, а правила говорят, что когда таблица стилей A импортирует таблицу стилей B, правила таблицы A обладают преимуществом над правилами таблицы B.
- Затем рассматривается **приоритет** каждого правила. Приоритет – это числовое значение, и чем оно больше, тем выше приоритет. Можно определить приоритет явным образом в атрибуте priority элемента <xsl:template> или можно предоставить системе право назначить приоритет по умолчанию. В этом случае система назначает приоритеты в зависимости от того, является ли образец общим или конкретным. Например, образец «подраздел/заголовок» (который соответствует любому элементу <заголовок>, чьим родителем является элемент <подраздел>) получит более высокий приоритет, чем образец «*», который соответствует любому элементу. Назначаемые системой приоритеты всегда лежат в диапазоне от -0.5 до +0.5, тогда как приоритеты, назначаемые пользователем, обычно будут иметь значения от 1 и выше, хотя никаких ограничений на их значение не существует. Более подробное описание элемента <xsl:template> приведено в соответствующем разделе главы 4.
- Наконец, если есть более одного правила с одинаковым приоритетом импортирования и с одинаковым назначенным приоритетом, XSLT-процессор имеет выбор: он может сообщить об ошибке или использовать то правило, которое в таблице стилей появляется последним. Различные процессоры в такой ситуации ведут себя по-разному, что может привести к некоторым проблемам с переносимостью. Это следует иметь в виду и позаботиться о том, чтобы неоднозначность не возникала.

Переменные, выражения и типы данных

Система типов данных лежит в основе любого языка, а способы, которыми выражения вычисляют значения и присваивают их переменным, близко связаны с системой типов. Давайте рассмотрим эти аспекты языка более подробно.

Переменные

XSLT позволяет определять глобальные переменные, которые доступны во всей таблице стилей, а также локальные переменные, которые доступны только в пределах конкретного тела шаблона. Имена и значения переменных определяются в элементе <xsl:variable>. Например:

```
<xsl:variable name="width" select="50"/>
```

Здесь определяется переменная с именем `width` и значением `50`. К переменной можно затем обращаться в выражениях XPath, записывая ее в виде `$width`. Если элемент `<xsl:variable>` находится на самом верхнем уровне таблицы стилей (то есть является непосредственным потомком элемента `<xsl:stylesheet>`), тогда это глобальная переменная; при появлении в пределах тела элемента `<xsl:template>` это – локальная переменная.

Точно так же XSLT позволяет задавать глобальные и локальные параметры, используя элемент `<xsl:param>`. Глобальные параметры задаются вне таблицы стилей (например, в командной строке или через API – конкретный механизм зависит от реализации). Локальные параметры для шаблона задаются с помощью элемента `<xsl:with-param>` при вызове шаблона.

Переменные и параметры не являются статически типизированными: они принимают значения любого присвоенного им типа. В XSLT и XPath определено пять типов данных:

- **Строковый тип (String)** – любая последовательность символов Unicode, разрешенных в XML.
- **Числовой тип (Number)** – число с плавающей точкой двойной точности, как определено в IEEE 754.
- **Логический тип (Boolean)** – принимает значение истина или ложь.
- **Набор узлов (Node-set)** – набор узлов в исходном дереве.
- **Внешний объект (External object)** – объект, например объект Java, возвращаемый функцией расширения, которая написана на языке, отличном от XSLT/XPath. Присваивая XSLT-переменным значения внешних объектов, можно передавать значения из одной внешней функции в другую.

В XSLT 1.0 существует дополнительный тип данных – **фрагмент конечного дерева (Result Tree Fragment)**. Это, по существу, дерево, соответствующее модели, описанной ранее в этой главе. В рабочем проекте спецификации XSLT 1.1 фрагмент конечного дерева уже не считается отдельным типом данных, но входит в тип данных – набор узлов. Дерево рассматривается как набор узлов, в составе которого есть один узел, являющийся корнем дерева.

Эти типы данных описаны более подробно далее в разделе «Типы данных» этой главы.

В общих чертах использование переменных очень похоже на их использование в традиционных языках программирования и языках подготовки сценариев. Для них даже существуют подобные правила области действия. Однако есть очень важное отличие: значение, присвоенное однажды переменной, не может быть изменено. Это отличие очень сильно влияет на способ написания программ, поэтому оно обсуждается подробно в разделе «Программирование без операторов присваивания» главы 9.

Выражения

Синтаксис выражений определен в рекомендации XPath и описан подробно в главе 5.

В таблице стилей XSLT выражения XPath используются в ряде контекстов. Они употребляются как значения атрибутов для многих элементов XSLT, например:

```
<xsl:value-of select="($x + $y) * 2"/>
```

В этом примере $x и $y – ссылки на переменные, а операторы «+» и «*» имеют свои обычные значения знаков сложения и умножения.

Многие XPath-выражения, подобные этому, следуют синтаксису, который похож на синтаксис других языков программирования. А отличиями в синтаксисе и своим названием XPath обязан синтаксису **выражения пути (Path Expression)**.

Выражение пути определяет навигацию по дереву документа. Начиная от некоторой стартовой точки – обычно или от текущего узла, или от корня – он производит последовательные шаги в определенных направлениях. На каждой стадии путь может разветвиться: так, например, можно найти все атрибуты всех непосредственных потомков стартового узла. Результатом всегда является набор узлов. Он может быть пустым или содержать только один узел, но все равно он считается набором.

Направления передвижения по дереву называются осями. Различные оси подробно определены в главе 5. Это следующие оси:

- Ось child, которая находит всех непосредственных потомков узла.
- Ось attribute, которая находит все атрибуты узла.
- Ось ancestor, которая находит всех предков узла.
- Ось following-siblings, которая находит узлы, встречающиеся после текущего и имеющие того же родителя.
- Ось preceding-siblings, которая находит узлы, встречавшиеся перед текущим и имеющие того же родителя.

Кроме указания направления передвижения по дереву выражения пути могут определять на каждом этапе, какие узлы должны быть выбраны. Это можно сделать несколькими различными способами:

- Указывая имена узлов (полностью или частично).
- Указывая тип узлов (например элементы или инструкции обработки).
- Определяя предикат, которому должны удовлетворять узлы, – произвольное логическое выражение.
- Определяя относительное положение узла по оси: например, возможно выбрать только самый близкий предшествующий одноуровневый узел.

Синтаксис выражений пути использует символ «/» в качестве оператора, разделяющего поочередные шаги. Символ «/» в начале выражения пути ука-

зывает, что начало отсчета – корневой узел; в противном случае это обычно текущий узел. В пределах каждого шага ось, написанная первой, отделяется от других условий разделителем «::». Однако ось child выбрана по умолчанию, поэтому ее можно опустить, а ось attribute можно обозначать символом «@».

Например, выражение:

```
child::item/attribute::category
```

является выражением из двух шагов: на первом шаге выбираются все непосредственные потомки элемента <item> текущего узла, а на втором – их атрибуты category. Это выражение можно упростить:

```
item/@category
```

Предикаты, которым должны удовлетворять узлы, записываются в квадратных скобках, например:

```
item[@code='T']/@category
```

Это выбирает атрибуты category тех прямых потомков элементов <item>, которые имеют атрибут code со значением 'T'.

Существует много способов сокращения выражений пути, делающих их проще для написания, но основная структура остается той же. Более детально это описано в главе 5.

Контекст

Способ вычисления выражений является в некоторой степени контекстнозависимым. Например, значение выражения $x зависит от текущего значения переменной x, а значение выражения «.» зависит от того, какой узел в исходном документе обрабатывается в данный момент.

Существует два контекстных аспекта: статический контекст, который зависит только от того, где именно в таблице стилей находится выражение, и динамический контекст, который зависит от состояния обработки во время вычисления выражения.

Статический контекст включает в себя:

- Набор объявлений пространств имен, действующих в той области таблицы стилей, где записано выражение. Он определяет действительность и значение всех префиксов пространств имен, использованных в выражении.
- Набор объявлений переменных (то есть элементов <xsl:variable> и <xsl:param>), действующих в точке, где записано выражение. Он определяет действительность всех ссылок на переменные, использованные в выражении.
- Набор функций, которые могут быть вызваны. Внешние библиотеки функций можно устанавливать через определенные поставщиком механизмы или с помощью элемента <xsl:script> в XSLT 1.1 (это описано в соответствующем разделе главы 4).

- Базовый URI элемента таблицы стилей, содержащего XPath-выражение. Он воздействует на результат, только если выражение использует функцию `document()`, указывая относительный URI, который интерпретируется относительно таблицы стилей. Функция `document()` описана в соответствующем разделе главы 7.

Динамический контекст включает в себя:

- Текущие значения всех переменных, в области действия которых находится выражение. Они могут быть различными при каждом вычислении выражения.
- Текущая локализация в исходном дереве. Текущая локализация включает:
  - **Текущий узел.** Это узел в исходном дереве, который обрабатывается в данный момент. Узел становится текущим узлом, когда он обрабатывается с помощью инструкции `<xsl:apply-templates>` или `<xsl:for-each>`. К текущему узлу можно обращаться через функцию `current()`.
  - **Узел контекста.** Это обычно тот же самый текущий узел, только указанный в предикате для уточнения шагов в рамках выражения пути, когда он является узлом, проверяемым в данный момент предикатом. На узел контекста можно ссылаться с помощью выражения «`.`» или более длинной его формы «`self::node()`». Например, выражение «`a[.='Мадрид']`» выбирает все элементы `<a>`, имеющие строковые значения `'Мадрид'`.
  - **Положение в контексте.** Это целое число (>=1), которое указывает положение узла контекста в текущем наборе узлов. На положение в контексте можно ссылаться с помощью функции `position()`. Когда инструкция `<xsl:apply-templates>` или `<xsl:for-each>` используется для обработки набора узлов, этот набор становится текущим набором узлов, и положение в контексте по мере обработки каждого из узлов набора принимает значения 1...n. Когда в пределах выражения пути используется предикат, положение в контексте – это положение узла в пределах набора проверяемых узлов. Так, например, выражение «`child::a[position() != 1]`» выбирает все дочерние элементы с именем `<a>`, кроме первого.
  - **Размер контекста.** Это целое число (> =1), которое указывает число узлов в текущем наборе узлов. Размер контекста определяется с помощью функции `last()`. Так, например, выражение «`child::a[position() != last()]`» выбирает все дочерние элементы с именем `<a>`, кроме последнего.

Спецификации XSLT и XPath используют различную терминологию для описания контекста. XSLT оперирует с концепциями текущего узла и текущего набора узлов, в то время как XPath – с концепциями узла контекста, положения в контексте и размера контекста. Когда выполняется инструкция `<xsl:apply-templates>` или `<xsl:for-each>`, текущим набором узлов становится набор обрабатываемых узлов, а текущим узлом становится каждый из этих узлов по очереди. Когда вычисляется XPath-выражение, узлом контекста становится текущий узел; положением в контексте становится поло-

жение текущего узла в рамках текущего набора узлов; а размером контекста является размер текущего набора узлов.

Некоторые системные функции, используемые в выражениях, имеют другие зависимости от контекста, например действие функции `key()` зависит от набора действующих объявлений `<xsl:key>`; но вышеприведенный набор охватывает всю контекстную информацию, доступную для пользовательских выражений напрямую.

Типы данных

XSLT – это динамически типизируемый язык, в котором типы связаны скорее со значениями, чем с переменными. В этом отношении он подобен языкам VBScript и JavaScript.

Не считая фрагментов конечного дерева, в настоящее время различают пять доступных типов данных, а преобразования между ними обычно выполняются неявно, если этого требует контекст. Кроме того, однако, для выполнения явных преобразований доступны функции `boolean()`, `number()` и `string()`. В таблице ниже приведены результаты преобразований между этими пятью типами данных:

из \ в	**логический тип**	**число**	**строка**	**набор узлов**	**внешний объект**
логический тип	**неприменимо**	ложь ⇒ 0 истина ⇒ 1	ложь ⇒ 'false' истина ⇒ 'true'	недопустимо	недопустимо
число	0 ⇒ ложь другое ⇒ истина	**неприменимо**	преобразует в десятичный формат	недопустимо	недопустимо
строка	пустая ⇒ ложь другое ⇒ истина	анализируется как десятичное число	**неприменимо**	недопустимо	недопустимо
набор узлов	пустой ⇒ ложь другое ⇒ истина	преобразуется через строковый тип данных	строковое значение первого по порядку узла в документе	**неприменимо**	недопустимо
внешний объект	недопустимо	недопустимо	недопустимо	недопустимо	**неприменимо**

Более подробное описание функций `boolean()`, `number()` и `string()` дано в главе 7.

В XSLT 1.1 временное дерево, создаваемое с использованием непустого элемента `<xsl:variable>`, относится к типу данных – набор узлов (набор узлов содержит единственный узел – корень дерева), а его преобразования в и из других типов данных определены в таблице выше. Однако в XSLT 1.0 временные деревья представляются дополнительным типом данных – фрагмен-

тами конечного дерева, – который ведет себя во многом подобно набору узлов, но может использоваться не во всех контекстах, в которых допустимо использование наборов узлов. Многие продукты XSLT 1.0 снабжены дополнительной функцией для преобразования фрагментов конечного дерева в набор узлов, когда это необходимо; в MSXML3, например, такой функцией является функция `msxml:node-set()`.

> *Подробнее о дополнительных функциях в продуктах конкретных производителей см. в соответствующих приложениях.*

Преобразование фрагмента конечного дерева в другие типы данных: строку, число или логический тип – происходит неявно и приводит к тому же результату, что и преобразование набора узлов, содержащего только корневой узел дерева, что отражено в таблице выше. Из этих правил следует:

из \ в	логический тип	число	строка
фрагмент конечного дерева	всегда истина	преобразуется в строку, которая затем преобразуется в число	сцепление всех текстовых узлов дерева

В следующих разделах описываются основные типы данных, а именно: логический тип, число, строка и набор узлов. Тип данных – внешний объект, – который используется функциями расширения, написанными на других языках программирования, обсуждается в главе 8.

Значения логического типа

Данные логического типа в XPath имеют значения истина и ложь. Для представления этих значений нет никаких констант, вместо этого они могут быть записаны через вызов функций `true()` и `false()`.

Значения данных логического типа можно получить при сравнении значений других типов данных с помощью таких операторов, как «=» и «!=», а также их можно объединять с помощью двух операторов – «and» и «or» – и функции `not()`.

В отличие от SQL, XPath не использует трехзначную логику. Значение логических данных – всегда или истина, или ложь; оно не может быть не определено или быть значением пустым. Ближайший эквивалент значению null SQL в XPath – пустой набор узлов, и при сравнении пустого набора узлов со строковым типом данных или числом результатом всегда будет ложь, независимо от используемого оператора сравнения. Например, если текущий элемент не имеет атрибута `name`, то выражения «@name='Boston'» и «@name!='Boston'» оба возвратят значение ложь. Однако выражение «not(@name='Boston')» возвратит значение истина.

Дополнительные сведения по поводу иногда странного поведения операторов равенства и неравенства в отношении наборов узлов см. в разделах «ВыражениеРавенства (EqualityExpr)» и «ВыражениеОтношения (RelationalExpr)» главы 5.

Числовые значения

Число в XPath – это всегда число с плавающей точкой двойной точности (64-разрядное), и его поведение должно отвечать стандарту IEEE 754. Этот стандарт – «Стандарт IEEE для двоичной арифметики с плавающей точкой» («IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std.» 754-1985) – в течение ряда лет был широко реализован многими микропроцессорами, но только его использование в языке Java принесло ему признание программистов на языках высокого уровня. Тем, кто понимает, как плавающая запятая ведет себя в Java, содержание этого раздела будет довольно знакомо; остальным оно может показаться странным.

В отличие от большинства других языков программирования, XPath не использует экспоненциальный формат для чисел с плавающей запятой ни на вводе, ни на выводе. Если требуется ввести число один триллион, следует записать его как 1000000000000, а не 1.0E12.

IEEE 754 определяет следующий диапазон значений для чисел с удвоенной точностью:

Значение	Описание
Конечные ненулевые значения	Это значения формата $s \times m \times 2^x$, где s (знак, sign) равно +1 или -1, m (мантисса, mantissa) – целое положительное число, меньше 2^{53}, а x (экспонента) – целое число в диапазоне от -1075 до 970, включительно.
Положительный нуль	Это результат вычитания числа из самого себя. Кроме того, это может быть результат деления любого положительного числа на бесконечность или деления очень малого числа на очень большое число того же знака.
Отрицательный нуль	Это результат деления любого отрицательного числа на бесконечность. Это может быть также результатом деления положительного числа на отрицательную бесконечность или деления очень малого отрицательного числа на очень большое положительное число и наоборот.
Положительная бесконечность	Это результат деления любого положительного числа на нуль. Также это может быть результатом перемножения двух очень больших чисел одного знака. Заметьте, что деление на нуль – не ошибка: это действие имеет четкий результат.
Отрицательная бесконечность	Это результат деления любого отрицательного числа на нуль. Это может быть также результатом перемножения двух очень больших чисел с разными знаками.
NaN (не-число)	Это результат попытки преобразовать нечисловое значение строкового типа данных в число. Кроме того, это может использоваться для обозначения «неизвестного» или «неприменимого», подобных значению null в SQL.

Эти значения не могут быть непосредственно написаны как константы XPath. Однако они могут быть выражены как результаты выражений, например:

Значение	XPath выражение
Отрицательный нуль	-0
Положительная бесконечность	1 div 0
Отрицательная бесконечность	-1 div 0
NaN (не-число)	number («NaN»)

Технически отрицательные числа не могут быть написаны непосредственно как константы: «-10» является скорее выражением, чем числом, – но на практике оно может использоваться везде, где могут использоваться числовые константы. Единственное, за чем нужно следить, – перед унарным отрицательным оператором может быть необходим пробел, если записывается выражение типа «$x div -1».

Значения чисел, кроме NaN, **упорядочиваются**. Ниже показан порядок от самого маленького до самого большого числа:

- отрицательная бесконечность
- отрицательные конечные ненулевые значения
- отрицательный нуль
- положительный нуль
- положительные конечные ненулевые значения
- положительная бесконечность

Этот порядок показывает результат сравнений «меньше чем» (less-than) и «больше чем» (greater-than), а также результат сортировки с использованием элементов <xsl:apply-templates> или <xsl:for-each> с ключом сортировки, указанным с помощью выражения <xsl:sort data-type="number">.

NaN является не упорядочиваемым числовым значением, поэтому операторы «<», «<=», «>» и «>=» возвращают ложь, если любой или оба операнда – NaN. Однако когда <xsl:sort> используется для сортировки последовательности числовых значений, которая включает одно или больше значений NaN, эти значения ставятся в начале последовательности (или в конце, если выбран порядок по убыванию).

> *Положение NaN в отсортированной последовательности не было определено в оригинале спецификации XSLT 1.0, и это упущение было исправлено позднее в списке опечаток. По этой причине можно встретить XSLT-процессоры, которые не поддерживают такой способ сортировки.*

Положительный нуль и отрицательный нуль при сравнении считаются равными. Это означает, что операторы «=», «<=» и «>=» возвратят истину, в то время как операторы «!=», «<» и «>» возвратят ложь. Однако другие действия могут различать положительный и отрицательный нули; например, выражение «1.0 div $x» имеет значение положительной бесконечности, если $x – положительный нуль, и отрицательной бесконечности, если $x – отрицательный нуль.

Оператор равенства «=» возвратит ложь, если любой или оба операнда – NaN, а оператор неравенства «!=» возвратит истину, если любой или оба операнда – NaN. Проявляйте осторожность с очевидными противоречиями, к которым это ведет; например выражение «$x=$x» может иметь значение истина, а «$x<$y» не обязательно дает тот же самый ответ, что и «$y>$x».

Самый простой способ проверить, является ли $x значением NaN, следующий:

```
<xsl:if test="$x!=$x">
```

Если это кажется слишком туманным на чей-то взгляд, то, зная что $x – числовое значение, можно записать:

```
<xsl:if test="string($x)='NaN'">
```

Тем, кто знаком со значениями null в SQL, эта логика покажется знакомой, но есть некоторые тонкие отличия. Например, в SQL условие «null=null» имеет значение null, поэтому «not(null=null)» также имеет значение null; в то время как в XPath условие «NaN=NaN» имеет значение ложь, поэтому «not(NaN=NaN)» является истиной.

XPath обеспечивает ряд операторов и функций, которые действуют на числовые значения:

- Числовые операторы сравнения «<», «<=», «>» и «>=». Имейте в виду, что в пределах таблицы стилей следует соблюдать соглашения экранирования в XML и писать, например «<» вместо «<».
- Числовые операторы равенства «=» и «!=».
- Унарный отрицательный оператор «-».
- Мультипликативные операторы «*», «div» и «mod».
- Аддитивные операторы «+» и «-».
- Функция number(), которая может преобразовать любое значение в число.
- Функции string() и format-number(), которые преобразуют числа в строковые типы данных.
- Функция boolean(), которая преобразует числа в логический тип данных.
- Функции round(), ceil() и floor(), которые преобразуют числа в целые числа.
- Функция sum(), которая суммирует числовые значения набора узлов.

Операторы ведут себя с числами точно в соответствии с IEEE 754. XPath не так строго, как Java, определяет используемые алгоритмы округления для

неточных результатов и последовательность выполняемых действий. Однако многие реализации придерживаются правил языка Java.

Числовые операторы и функции XPath никогда не выдают ошибку. Операция, которая вызывает переполнение, приводит к положительной или отрицательной бесконечности, операция, которая вызывает опустошение, приводит к положительному или отрицательному нулю, а операция, которая не имеет никакого другого заметного результата, приводит к `NaN`. Все числовые действия и функции с `NaN` как операндом приводят в результате к `NaN`. Например, если применить функцию `sum()` к набору узлов, тогда, если строковое значение любого из узлов не может быть преобразовано в число, результатом функции `sum()` будет `NaN`.

Строковые значения

Строковые значения в XPath – это любая последовательность, состоящая из нуля или большего количества символов, где набор возможных символов тот же, что и в XML: по существу, это символы, определенные в Unicode.

В выражениях XPath значения строк могут быть написаны в форме литералов с использованием одинарных или двойных кавычек, например `'Джон'` или `"Мери"`. Теоретически сам строковый набор может содержать противоположный символ кавычки как часть своего значения, например `"John's"`. Однако на практике выражения XPath записываются в рамках атрибутов XML, так что противоположный символ кавычки вообще будет использоваться как разделитель атрибутов. Более подробные сведения приведены в разделе «Литералы» главы 5.

В отличие от SQL, никакого специального значения null нет. В тех случаях, где никакое другое значение не подходит, используется строковый тип данных нулевой длины. Фактически термины «нулевой строковый тип» и «пустой строковый тип» используются взаимозаменяемо и подразумевают строковые данные нулевой длины.

Единственно разрешенными управляющими символами ASCII (коды ниже `#x20`) являются пробельные символы `#x9`, `#xA` и `#xD` (табуляция, возврат каретки, переход на новую строку).

Строковые данные можно сравнивать с помощью операторов «`=`» и «`!=`». Их сравнивают символ за символом (без дополнительного заполнения пробелами, как в SQL). Некоторые реализации могут нормализовывать строковые данные перед их сравнением, поддерживать различные представления Unicode для одного и того же символа с ударением, но это не обязательно. Нет никаких операторов или функций для сравнения двух образцов строковых данных без учета регистра: лучшее, чего можно достичь (если известно, что набор символов ограничен, например, только символами ASCII), – это преобразовать нижний регистр в верхний или наоборот, используя функцию `translate()`. Можно также воспользоваться внешней функцией, определяемой пользователем.

Нет также никаких операторов или функций, способных определить в алфавитной последовательности, является ли одна строка большей или меньшей, чем другая: операторы «<» и «>» всегда требуют числового сравнения. Единственный обходной способ (кроме создания внешней функции) состоит в том, чтобы произвести сортировку; записать оба образца строковых данных в виде элементов временного дерева, а затем обработать элементы дерева в отсортированном порядке.

При подсчете символов в строковых данных, например с помощью функции `string-length()`, значимым является число символов XML, а не число 16-разрядных кодов Unicode. Это означает, что псевдопары Unicode считаются как один символ. Псевдопары Unicode, которые используются для расширения Unicode за пределы 65 535 символов, очень редко встречаются на практике, хотя в будущем их использование может возрасти.

Наборы узлов

Набор узлов – это группа узлов в исходном дереве документа. Если есть несколько исходных деревьев документа, набор узлов может содержать узлы больше, чем от одного дерева. В набор узлов могут также входить узлы из временных деревьев или из деревьев, вошедших в таблицу стилей от внешних функций или в виде параметров. Набор узлов могут образовывать узлы любого типа, причем разные типы узлов могут быть смешаны в одном и том же наборе узлов. Это чисто математический набор; каждый узел может встречаться самое большее однажды, и не существует никакого обязательного порядка.

Для представления единственного узла не введено специального типа данных; вместо этого возможен набор узлов, состоящий только из одного узла. Например, при использовании выражения «`@name`» для поиска значения атрибута `name` текущего элемента результатом будет набор узлов, содержащий единственный узел атрибута, если данный элемент имеет атрибут `name`, или пустой набор, если не имеет.

Когда значение набора узлов преобразуется в логический тип, пустой набор узлов принимает значение ложь, а набор, содержащий один или больше узлов, принимает значение истина. Таким образом, можно использовать критерий:

```
<xsl:if test="@name">
```

чтобы выяснить, имеет ли текущий элемент атрибут `name`.

Узлы в наборе узлов могут иметь потомков, но эти потомки не считаются членами данного набора узлов. Например, выражение «/» возвращает набор узлов, содержащий единственный узел – корень. Другие узлы, подчиненные корню, могут быть достигнуты от этого узла, но сами они не являются членами набора узлов, и поэтому значением «`count(/)`» всегда будет 1.

Набор узлов внутренне не упорядочен, хотя во многих контекстах узлы обрабатываются в **порядке документа**. Когда два узла происходят из одного документа, их относительное положение в порядке документа основано на их по-

ложении в дереве документа: например, в порядке документа элемент предшествует своим потомкам, а узлы потомков одного уровня (узлы – потомки одного родителя) перечисляются в том порядке, в каком они встречаются в исходном документе. Когда два узла происходят из разных документов, их относительный порядок не определен. Упорядочение узлов атрибутов и пространств имен определено только частично: за узлом элемента следуют узлы его пространств имен, затем его атрибуты, а после этого – его потомки, но порядок среди узлов пространств имен и среди узлов атрибутов не уточняется.

Временные деревья

XSLT позволяет создавать новое дерево и обращаться к нему, используя переменную. В XSLT 1.0 принято считать такие деревья отдельным типом данных, называемым фрагментами конечного дерева. Фрагмент конечного дерева аналогичен набору узлов, содержащему единственный узел – корень дерева, но он не может использоваться во всех контекстах, где могут использоваться наборы узлов. В частности, с фрагментом конечного дерева можно делать только две вещи: можно копировать его в другое дерево, используя инструкцию `<xsl:copy-of>`, или можно извлекать его строковые значения (сцепление всех текстовых узлов в дереве). Однако многие поставщики обеспечивают дополнительную функцию, позволяющую конвертировать фрагмент конечного дерева в эквивалентный набор узлов.

В XSLT 1.1 временное дерево ведет себя точно как набор узлов и может использоваться везде, где может использоваться набор узлов. Преобразование в набор узлов происходит неявно всякий раз, когда переменная, значением которой является временное дерево, используется в контексте, который предполагает набор узлов. Таким образом, этот дополнительный тип данных исчез, а вместе с ним исчез и довольно неуклюжий термин «фрагмент конечного дерева».

> *Фактически в XSLT 1.1 вообще нет названия для этой концепции. В этой книге автор счел необходимым придумать название, чтобы было удобнее объяснять концепцию, поэтому возникли термины «временное дерево» и «переменная, значением которой является дерево» (tree-valued variable). Этих терминов нет в рекомендации по XSLT.*

Временное дерево всегда содержит корневой узел, а корневой узел может иметь потомков. Дерево не обязательно должно соответствовать правильно построенному XML-документу, например корневой узел сам может иметь текстовые узлы, а в числе его прямых потомков может быть более одного узла элемента. Однако оно должно соответствовать тем же правилам, каким соответствует в XML внешняя анализируемая сущность: например, все атрибуты, принадлежащие узлу элемента, должны иметь явные названия.

Пример: Временные деревья

Временное дерево создается с помощью тела объявления `<xsl:variable>`, например:

```
<xsl:variable name="tree"
   >AAA<xsl:element name="x">
   <xsl:attribute name="att">att-value</xsl:attribute>
      BBB</xsl:element>
   <xsl:element name="y"
   />CCC</xsl:variable>
```

(Столь странная компоновка должна гарантировать, что ни один пробельный символ не соседствует с текстовыми значениями, потому что это было бы трудно показать на схеме дерева. Влияние пробельных символов обсуждается в главе 3.)

Это создает дерево, показанное на схеме ниже. Каждый прямоугольный блок представляет узел; три его уровня – это, соответственно, тип узла, имя узла и строковое значение узла. Символ звездочки опять указывает, что данное строковое значение является сцеплением строковых значений дочерних узлов.

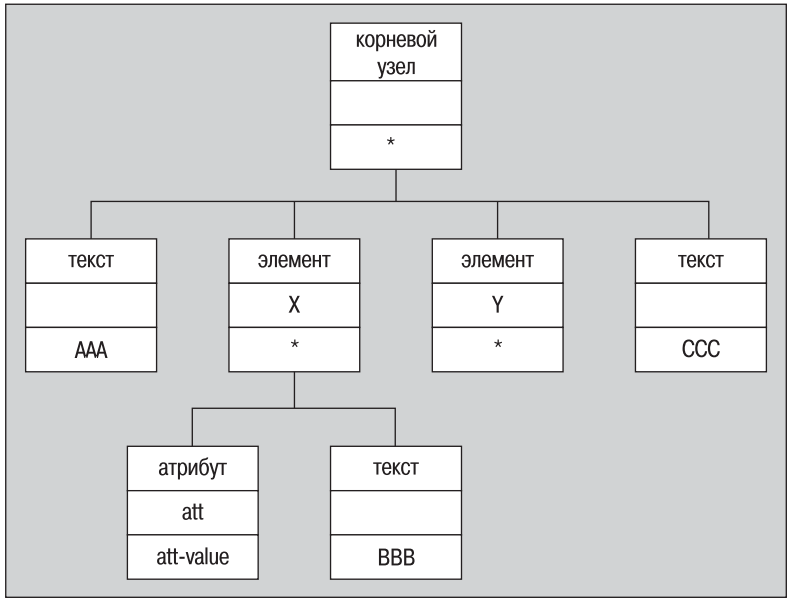


Рис. 2.9. *Временное дерево*

Как уже говорилось, в XSLT 1.0, сформировав дерево, с ним можно делать только две вещи: скопировать его в текущее выходное дерево (которое может быть конечным деревом или другим временным деревом) с помощью инструкции `<xsl:copy-of>` или преобразовывать его значение в строковый тип данных. Преобразование его в строковые данные дает сцепление всех текстовых узлов в дереве: в вышеупомянутом примере это даст «AAABBBCCC».

В XSLT 1.1 можно манипулировать деревом множеством других способов. Например:

- Можно применять шаблоны для обработки узлов дерева, используя инструкцию <xsl:apply-templates>. Удобно использовать различные режимы, чтобы было понятно, какие шаблонные правила предполагается применять к тому или иному дереву.
- Можно обращаться к его узлам явным образом, используя выражения пути, например <xsl:value-of select=$tree/x/@att¨/> даст на вывод текст «att-value».
- Можно использовать функции sum() и count(), например «count($tree//*)» возвратит 2 (число узлов элементов на дереве).

Это означает, к примеру, что можно использовать временное дерево как вспомогательную таблицу для последующего поиска. Следующий фрагмент таблицы стилей использует данные, сохраняющиеся во временном дереве, для получения названия месяца, номер которого содержится в переменной $mm:

```
<xsl:variable name=¨months¨>
  <Январь/><Февраль/><Март/><Апрель/><Май/><Июнь/>
  <Июль/><Август/><Сентябрь/><Октябрь/><Ноябрь/><Декабрь/>
</xsl:variable>
. . .
  <xsl:value-of select=¨name($months/*[position()=$mm])¨/>
```

В XSLT 1.0 можно достичь того же эффекта следующим образом:

```
<xsl:value-of select=¨name(xx:node-set($months)/*[position()=$mm])¨/>
```

Но детали зависят от используемого XSLT-процессора.

Резюме

В этой главе обсуждались важные концепции, необходимые для понимания того, что делает XSLT-процессор.

- Исследована полная архитектура системы, в которой таблица стилей управляет преобразованием исходного дерева в конечное дерево.
- В некоторых подробностях показана древовидная модель, используемая в XSLT, и ее соотношение с XML-стандартами; кроме того, отмечен ряд ее отличий от модели DOM.
- Рассмотрено, как используются шаблонные правила для определения действий, которые будут предприняты XSLT-процессором, когда он сталкивается со специфическими типами узлов дерева.
- Показан способ использования в языке XSLT-выражений, типов данных и переменных для вычисления значений.

В следующей главе будет более подробно рассмотрена структура таблицы стилей XSLT.

3

Структура таблицы стилей

В этой главе описана общая структура таблицы стилей. Предназначение этой главы – объяснить некоторые концепции, используемые при создании таблиц стилей, прежде чем перейти к основному справочному разделу книги в главах с 4 по 8. Некоторые из этих концепций сложны, они часто вызывают затруднения, поэтому здесь они будут обсуждаться более детально. Однако для создания первой собственной таблицы стилей вовсе не обязательно полностью овладеть всем материалом этой главы, ее можно использовать как справочное пособие, возвращаясь к необходимым темам, когда потребуется их более глубокое понимание.

В этой главе рассматриваются следующие темы:

- **Модули таблиц стилей**: будет показано, как можно сформировать таблицу стилей из одного или нескольких модулей таблиц стилей, связываемых вместе элементами `<xsl:import>` и `<xsl:include>`.
- Элемент `<xsl:stylesheet>` (или `<xsl:transform>`), являющийся самым внешним элементом большинства модулей таблиц стилей.
- Инструкция обработки `<?xml-stylesheet?>`, используемая для связывания исходного документа с соответствующей таблицей стилей и позволяющая также внедрять таблицы стилей прямо в исходный документ, стиль которого они определяют.
- Краткое описание **элементов верхнего уровня**, находящихся в таблице стилей, то есть непосредственных потомков элементов `<xsl:stylesheet>` или `<xsl:transform>`; полные спецификации приводятся в главе 4.
- Упрощенные таблицы стилей, в которых элементы `<xsl:stylesheet>` и `<xsl:template match="/">` опущены, чтобы таблицы стилей XSLT больше походили на таблицы обычных языков шаблонов, с которыми могут быть знакомы некоторые пользователи.

- Идея **тела шаблона**, последовательности текстовых узлов и конечных литеральных элементов, копируемых в конечное дерево, а также инструкций и дополнительных элементов, которые подлежат выполнению.
- **Шаблоны значений атрибутов**, которые используются для определения изменяющихся атрибутов не только для конечных литеральных элементов, но также и некоторых элементов XSLT.
- Средства, позволяющие расширить спецификацию как компаниями-разработчиками, так и самим консорциумом W3C, без неблагоприятного воздействия на переносимость таблиц стилей.
- Обработка **пробельных символов** в исходном документе, непосредственно в таблице стилей и в конечном дереве.

Модульное строение таблицы стилей

В предыдущей главе обсуждалась модель обработки данных в XSLT, основанная на том, что таблица стилей определяет правила, по которым исходное дерево преобразуется в конечное дерево.

Таблицы стилей подобно программам на других языках могут стать довольно длинными и сложными, поэтому имеет смысл допускать разделение их на отдельные модули. Это позволит многократно использовать модули и объединять их по-разному для различных целей: например, может потребоваться использовать две различные таблицы стилей для отображения пресс-релизов на экране и на бумаге, но некоторые компоненты обеих таблиц могли бы быть общими для них. Такие общие компоненты можно вынести в отдельный модуль, который используется в обоих случаях.

Можно рассматривать весь набор модулей как **программу таблицы стилей**, а его компоненты – как **модули таблицы стилей**.

> *Стандарт XSLT не использует эту терминологию. Фактически в нем под термином «таблица стилей» иногда подразумевается программа таблицы стилей, а иногда – модуль таблицы стилей, как они определены здесь.*

Один из модулей является **основным модулем таблицы стилей**. Это, фактически, основная программа, модуль, который задается либо при помощи инструкции обработки `<?xml-stylesheet?>` в исходном документе, либо с помощью каких-нибудь параметров командной строки, либо через API, предоставляемый компанией-разработчиком. Основной модуль таблицы стилей может с помощью элементов `<xsl:include>` и `<xsl:import>` подключать другие модули таблицы стилей. Последние, в свою очередь, могут также подключать дополнительные модули и так далее.

Пример: Использование элемента <xsl:include>

Исходный документ

Входной документ, пример.xml, выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<документ>
    <автор>Майкл Кэй</автор>
    <название>Справочник по XSLT</название>
    <авторское-право/>
    <дата/>
    <аннотация>Всеобъемлющее руководство по спецификациям XSLT 1.1 и XPath 1.0,
     опубликованным консорциумом World Wide Web
       </аннотация>
</документ>
```

Таблицы стилей

Таблица стилей использует инструкцию <xsl:include>. Задача этой таблицы стилей – скопировать исходный документ в выходной документ без изменений, за исключением того, что во все элементы <дата> вставляется текущая дата, а во все элементы <авторское-право> вставляются строковые данные, идентифицирующие обладателя авторского права.

Тремя модулями этого проекта таблицы стилей являются файлы: основной.xsl, дата.xsl и авторское-право.xsl. Модуль дата.xsl использует функцию расширения; он написан так, чтобы мог работать с любым процессором XSLT 1.1, который поддерживает функции расширения, написанные на Java.

В качестве имени таблицы стилей нужно указать имя основного модуля таблицы стилей, другие модули будут подключены автоматически. Данная таблица стилей написана так, что все модули должны находиться в одном и том же каталоге.

основной.xsl

Первый модуль, основной.xsl, содержит основную логику таблицы стилей:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0"
>
<xsl:include href="дата.xsl"/>
<xsl:include href="авторское-право.xsl"/>

<xsl:output method="xml" encoding="utf-8" indent="yes"/>
<xsl:strip-space elements="*"/>

<xsl:template match="дата">
    <дата><xsl:value-of select="$date"/></дата>
```

```
</xsl:template>

<xsl:template match="авторское-право">
    <авторское-право><xsl:call-template name="copyright"/></авторское-право>
</xsl:template>

<xsl:template match="*">
    <xsl:copy>
        <xsl:copy-of select="@*"/>
        <xsl:apply-templates/>
    </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Он начинается с двух элементов <xsl:include>, подключающих другие модули. Элемент <xsl:output> указывает, что вывод должен быть в формате XML с использованием набора символов Unicode и с отступами, показывающими структуру XML. Элемент <xsl:strip-space> указывает, что узлы пробельных символов в исходном документе должны игнорироваться; позже в этой главе будет рассказано подробнее об обработке пробельных символов. Далее, есть три шаблонных правила: одно для элементов <дата>, второе – для элементов <авторское-право> и третье – для всех остальных.

Шаблонное правило для элементов <дата> выводит значение переменной, названной $date. Эта переменная не определена в данном модуле таблицы стилей, но она есть в модуле дата.xsl, поэтому она доступна и здесь.

Шаблонное правило для элементов <авторское-право> аналогично вызывает шаблон, названный copyright. Снова в данном модуле нет шаблона с таким именем, но он есть в модуле авторское-право.xsl, поэтому может вызываться отсюда.

Наконец, шаблонное правило, соответствующее всем остальным элементам («match="*"»), заставляет копировать элемент из исходного документа в конечный, не изменяя его. Инструкции <xsl:copy> и <xsl:copy-of> обсуждаются в главе 4.

дата.xsl

Следующий модуль, дата.xsl, объявляет глобальную переменную, содержащую сегодняшнюю дату. Для получения даты он вызывает Java-класс «java.util.Date», обращаясь к нему как к внешней функции. Внешние функции обсуждаются позже в этой главе. Этот модуль таблицы стилей использует элемент <xsl:script>, определенный в XSLT версии 1.1.

```
<xsl:stylesheet
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.1"
      xmlns:Date="java:java.util.Date"
>
<xsl:script implements-prefix="Date"
       language="java" src="java:java.util.Date"/>
```

```
<xsl:variable name="date"
      select="Date:to-string(Date:new())"/>

</xsl:stylesheet>
```

В настоящее время немногие XSLT-процессоры поддерживают элемент <xsl:script>, так как он введен только в спецификации XSLT 1.1, которая во время написания книги принята только как рабочий проект. Однако эквивалентные специализированные средства есть во всех Java-основанных процессорах: см. подробнее в соответствующих приложениях.

авторское-право.xsl

Наконец, модуль авторское-право.xsl содержит шаблон copyright, который выводит информацию об авторском праве. Этот шаблон вызывается инструкцией <xsl:call-template> в главной таблице стилей. Для формирования информации об авторском праве шаблон использует переменную $owner; позже будет показано, насколько это полезно.

Довольно странное написание начального тега <xsl:template> помогает избежать в выводе перевода на новую строку перед текстом информации об авторском праве. Позже в этой главе будут описаны другие способы достижения этого.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0"
>
<xsl:variable name="owner">Wrox Press</xsl:variable>

<xsl:template name="copyright"
>Copyright © <xsl:value-of select="$owner"/> 2001</xsl:template>

</xsl:stylesheet>
```

Причина разделения этой программы таблицы стилей на три модуля в том, что модули дата.xsl и авторское-право.xsl многократно используются в других таблицах стилей. Функционально таблица стилей производила бы те же действия, если бы переменная $date и шаблон под названием copyright были определены прямо в главной таблице стилей.

Результат

```
<?xml version="1.0" encoding="utf-8" ?>
<документ>
   <автор>Майкл Кэй</автор>
   <название>Справочник по XSLT</название>
   <авторское-право>Copyright © Wrox Press 2001</авторское-право>
   <дата>Thu Jan 18 10:27:44 GMT+00:00 2001</дата>
   <аннотация>Всеобъемлющее руководство по спецификациям XSLT 1.1 и XPath 1.0,
    опубликованным консорциумом World Wide Web</аннотация>
</документ>
```

Нет никакой синтаксической разницы между основным модулем таблицы стилей и любым другим модулем; фактически любой модуль может использоваться как основной модуль.

Это означает, что элементы <xsl:include> и <xsl:import> могут использоваться в любом модуле, не только в основном. Таким образом, программа таблицы стилей, фактически, представляет собой дерево модулей таблицы стилей, корнем которого является основной модуль.

Вообще, модуль таблицы стилей – законченный XML-документ (исключение, **встроенная таблица стилей**, будет описано позже в этой главе, см. стр. 126). Элементом документа (самым внешним элементом XML-документа) является в этом случае элемент <xsl:stylesheet> или элемент <xsl:transform>: эти названия – синонимы. Элементы, являющиеся непосредственными потомками элемента <xsl:stylesheet> или <xsl:transform>, называются **элементами верхнего уровня** (правильнее было бы назвать их элементами второго уровня, но термин «элементы верхнего уровня» введен в стандарте). Элементы верхнего уровня, определенные для XSLT, перечислены позже в этой главе (см. стр. 129).

Элементы <xsl:include> и <xsl:import> – это всегда элементы верхнего уровня. Обычно элементы верхнего уровня могут появляться в любом порядке, но <xsl:import> – исключение: он должен появляться раньше всех других элементов верхнего уровня. Оба элемента имеют атрибут href, значение которого – URI. Обычно это будет относительный URL, определяющий положение включаемого или импортируемого модуля таблицы стилей относительно родительского модуля. Например, элемент <xsl:include href="mod1.xsl"/> подключает модуль под названием mod1.xsl, расположенный в том же самом каталоге, где и исходный модуль.

Разница между элементами <xsl:include> и <xsl:import> в том, что противоречивые определения разрешаются по-разному:

- <xsl:include> осуществляет дословное включение указанного модуля таблицы стилей, за исключением содержащегося в нем элемента <xsl:stylesheet>, в то место, где написан элемент <xsl:include>. Включенный модуль обрабатывается так, словно его элементы верхнего уровня, с их содержанием, находятся в исходном модуле непосредственно на месте элемента <xsl:include>.
- <xsl:import> также включает элементы верхнего уровня из указанного модуля таблицы стилей, но в этом случае определения из импортированного модуля имеют более низкое **преимущество импортирования**, чем определения в исходном модуле. Если встречаются противоречивые определения, то обычно верх берет то, которое имеет более высокое преимущество. Подробные правила, фактически, зависят от типа определения и приведены в спецификации элемента <xsl:import> в главе 4. Импортирование модуля, таким образом, довольно похоже на определение подклассов: исходный модуль также может использовать некоторые определения из импортированного модуля без изменений, а другие заменять своими собственными.

Наиболее очевидный вид определения – определение шаблонного правила с помощью элемента <xsl:template> с атрибутом match. Как было показано в предыдущей главе, если есть несколько шаблонных правил, соответствующих некоторому узлу в исходном дереве, выбор из них, в первую очередь, определяется преимуществом импортирования, а правила с преимуществом ниже самого высокого отбрасываются. Таким образом, шаблонное правило, определенное в конкретном модуле таблицы стилей, будет автоматически иметь приоритет над другим соответствующим правилом, имеющимся в импортируемом модуле.

Когда один модуль A импортирует два других модуля, B и C, как показано на рис. 3.1, то преимущество импортирования у A выше, чем у B и C, а у C выше, чем у B, если элемент <xsl:import>, который вводит B, предшествует элементу <xsl:import>, который вводит C.

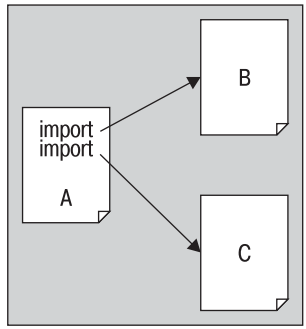


***Рис. 3.1.** Импортирование модулей таблиц стилей*

Когда таблица стилей с помощью <xsl:include> включает другую таблицу, определения включаемой таблицы стилей имеют такое же преимущество импортирования, как и определения из исходной таблицы.

Когда два определения имеют одинаковое преимущество импортирования (потому что они находятся в одном и том же модуле таблицы стилей или потому что каждый был в модуле, включенном в другую таблицу с помощью <xsl:include>), правила для разрешения конфликта зависят от типа определения. В случае, например, определений именованных шаблонов или переменных два определения с одним и тем же именем всегда рассматриваются как ошибка. В случаях определений шаблонных правил могут быть два решения: сообщение об ошибке или выбор определения, которое в таблице стилей встречается позже. В некоторых реализациях выбор между этими двумя решениями предоставляется пользователю. Подробные правила для каждого типа элементов верхнего уровня даются в главе 4, кроме того, они резюмируются в разделе <xsl:import>.

Пример: Использование элемента <xsl:import>

Этот пример дополняет предыдущий пример – использование элемента <xsl:include>, показывая, как использовать элемент <xsl:import> для включения определений в другой модуль таблицы стилей с отменой некоторых из них.

Исходный документ

Исходный документ для этого примера – пример.xml.

Таблица стилей

Вспомните, что модуль авторское-право.xsl использовал переменную $owner, содержащую имя владельца авторского права. Предположим, что требуется использовать шаблон copyright, но с другим владельцем авторского права. Этого можно добиться, написав основную таблицу стилей, измененную следующим образом (в загружаемых файлах примеров она называется principal2.xsl).

Эта таблица стилей для включения модуля авторское-право.xsl использует элемент <xsl:import>, вместо <xsl:include>, а также содержит новое объявление переменной $owner, которое отменит объявление в импортируемом модуле. Заметьте, что элемент <xsl:import> должен появляться первым.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0"
>
<xsl:import href="авторское-право.xsl"/>
<xsl:variable name="owner">Wrox Press Ltd</xsl:variable>
<xsl:include href="дата.xsl"/>

<xsl:output method="xml" encoding="utf-8" indent="yes"/>
<xsl:strip-space elements="*"/>

<xsl:template match="дата">
   <дата><xsl:value-of select="$date"/></дата>
</xsl:template>
<xsl:template match="авторское-право">
   <авторское-право><xsl:call-template name="copyright"/></авторское-право>
</xsl:template>

<xsl:template match="*">
   <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Результат

```
<?xml version="1.0" encoding="utf-8" ?>
<документ>
   <автор>Майкл Кэй</автор>
   <название>Справочник по XSLT</название>
   <авторское-право>Copyright © Wrox Press Ltd 2001</авторское право>
   <дата> Thu Jan 18 10:27:44 GMT+00:00 2001</дата>
   <аннотация>Всеобъемлющее руководство по спецификациям XSLT 1.1 и XPath 1.0,
опубликованным консорциумом World Wide Web</аннотация>
</документ>
```

Этот пример не работал бы, если бы был использован элемент `<xsl:include>`, а не `<xsl:import>`. Появилось бы сообщение, что переменная `$owner` была объявлена дважды. Причина в том, что с `<xsl:include>` эти два объявления имеют одинаковое преимущество импортирования, поэтому ни одно из них не может отменить другое.

Для модуля таблицы стилей будет ошибкой импортирование или включение самого себя, прямо или косвенно, потому что это вызвало бы бесконечный цикл.

Однако не будет ошибкой импортирование или включение модуля таблицы стилей в несколько мест в программе таблицы стилей. Следующая запись не ошибочна:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:import href="дата.xsl"/>
   <xsl:import href="дата.xsl"/>
</xsl:stylesheet>
```

Это может показаться довольно бессмысленным, но в сложной модульной структуре это может иногда получиться случайно, и это безопасно. Например, несколько модулей таблицы стилей могут независимо ссылаться на часто используемый модуль типа `дата.xsl`. В результате просто будет загружено несколько копий всех определений из `дата.xsl`, как будто было импортировано несколько идентичных файлов с различными названиями.

Если один и тот же модуль подключается дважды с помощью инструкции `<xsl:include>`, включенные определения будут иметь одинаковое преимущество импортирования, что, вероятно, вызовет ошибку. Если, например, включенный модуль определяет глобальную переменную или именованный шаблон, появится сообщение о повторяющихся определениях. В других случаях, например, когда файл использует элемент `<xsl:attribute-set>`, чтобы определить именованные наборы атрибутов, повторяющиеся определения неопасны (элемент `<xsl:attribute-set>` описан в главе 4). Однако если есть риск загрузки одного и того же модуля дважды, имеет смысл использовать `<xsl:import>`, а не `<xsl:include>`.

Заметьте, что у элементов `<xsl:include>` и `<xsl:import>` атрибут `href` не изменяется: компилятор таблицы стилей может скомпоновать все модули в таб-

лице стилей задолго до того, как исходный документ поступит для фактического выполнения преобразования. Многие часто спрашивают о средстве, которое может загружать модули таблицы стилей динамически, основываясь на решении, принятом во время выполнения преобразования. Но этого нельзя сделать.

Иногда такая потребность возникает, когда люди пытаются использовать <xsl:import> «задом наперед». Довольно естественно захотеть написать универсальную таблицу стилей G, которая при некоторых обстоятельствах (скажем, если пользователь – француз) импортирует таблицу стилей A, а при других обстоятельствах (если пользователь – испанец) импортирует таблицу стилей B. Но делать так не следует: необходимо выбрать A или B в качестве основного модуля таблицы стилей, который импортирует универсальный модуль G. Всегда специализированный модуль таблицы стилей должен импортировать универсальный модуль, а не наоборот.

Элемент <xsl:stylesheet>

Элемент <xsl:stylesheet> (или <xsl:transform>, что является синонимом) – самый внешний элемент любого модуля таблицы стилей.

Название <xsl:stylesheet> – общепринятое. Первая часть, xsl, является префиксом, задающим пространство имен, которому принадлежит имя элемента. Может использоваться любой префикс, если он связан с помощью объявления пространства имен с URI http://www.w3.org/1999/XSL/Transform. Кроме того, есть также обязательный атрибут version. Таким образом, открывающий тег элемента <xsl:stylesheet> обычно выглядит следующим образом:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0"
>
```

> *Если встретится таблица стилей, которая использует URI пространства имен* http://www.w3.org/TR/WD-xsl*, то эта таблица написана на диалекте Microsoft, основанном на раннем рабочем проекте стандарта XSLT. Эта версия была выпущена с Internet Explorer 5. Есть много различий между диалектом IE5 (который здесь упоминается как* **WD-xsl***, но Microsoft обычно ссылается на него просто как на «XSL») и окончательной спецификацией XSLT, описываемой в этой книге. Подробнее о WD-xsl см. в книге издательства Wrox Press «Справочник программиста XML IE5» («XML IE5 Programmer's Reference», ISBN 1-861001-57-6). Microsoft после этого выпустил модифицированную версию процессора MSXML, полностью поддерживающего спецификацию XSLT 1.0, подробности о котором можно найти в приложении A.*

Другие атрибуты, которые могут встретиться с этим элементом, описаны в разделе <xsl:stylesheet> главы 4. В частности это:

- id – для идентификации таблицы стилей, если она находится как встроенная таблица стилей внутри другого документа. Встроенные таблицы стилей описаны в следующем разделе.

- extension-element-prefixes список префиксов пространств имен, которые обозначают элементы, используемые для определенных компанией-разработчиком или определяемых пользователем расширений языка XSLT.
- exclude-result-prefixes список пространств имен, используемых в таблице стилей, которые не должны копироваться в конечное дерево, если они не нужны фактически. Как это работает, будет объяснено в разделе «Конечные литеральные элементы» этой главы.

Эти атрибуты воздействуют только на модуль таблицы стилей, в котором находится данный элемент <xsl:stylesheet>; они не влияют на то, что происходит во включенных или импортированных модулях таблицы стилей.

Элемент <xsl:stylesheet> часто содержит дополнительные объявления пространств имен. Действительно, если используются атрибуты extension-element-prefixes или exclude-result-prefixes, то все префиксы пространств имен, которые они упоминают, должны быть объявлены через объявления пространств имен в элементе <xsl:stylesheet>. Например, если требуется объявить «saxon» как префикс элемента расширения, то открывающий тег элемента <xsl:stylesheet> будет выглядеть следующим образом:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:saxon="http://icl.com/saxon"
   version="1.0"
   extension-element-prefixes="saxon"
>
```

Объявления пространств имен в элементе <xsl:stylesheet> и еще где-нибудь в таблице стилей имеют силу только в модуле таблицы стилей, в котором они находятся. Они не наследуются включенными или импортированными модулями.

Инструкция обработки <?xml-stylesheet?>

Эта инструкция обработки не является частью стандарта XSLT или XPath, точнее, отдельно по ней есть краткая рекомендация W3C, которую можно найти по адресу *http://www.w3.org/TR/xml-stylesheet*. В спецификации XSLT она упоминается только в примере, поэтому не подразумевается, что XSLT-процессоры должны ее поддерживать, тем не менее, все больше процессоров учитывает ее.

Инструкция обработки <?xml-stylesheet?> используется в исходном XML-документе для задания таблицы стилей, которая должна использоваться для его обработки. Может присутствовать несколько инструкций обработки <?xml-stylesheet?>, определяющих различные таблицы стилей, которые нужно использовать при разных обстоятельствах.

Инструкция обработки имеет атрибут href, значение которого – URI таблицы стилей (то есть основного модуля таблицы стилей), и атрибут type, ука-

зывающий язык, на котором написана таблица стилей. Это не обязательно должна быть таблица стилей XSLT, это может быть и CSS. Рекомендация W3C определяет, что для XSLT тип среды должен быть или `text/xml`, или `application/xml`; однако для реализации Microsoft в Internet Explorer 5 это должен быть `text/xsl`.

> *Формально инструкции обработки XML не содержат атрибутов, они содержат имя (здесь* `xml-stylesheet`*), сопровождаемое символьными данными. Однако многим нравится формировать символьные данные как последовательность пар* `имя="значение"`*, подобно атрибутам в открывающем теге элемента, и рекомендации по xml-stylesheet поддерживают эту практику. Они рассматривают пары* `имя="значение"` *как псевдоатрибуты.*

Полный список псевдоатрибутов для инструкции обработки `<?xml-stylesheet?>` следующий:

Имя атрибута	Значение	Смысл
`href` обязательный	URI	URI таблицы стилей. Это может быть абсолютный или относительный URL документа XML, содержащего таблицу стилей, или он может содержать идентификатор фрагмента (например `#styleB`), используемый для определения местоположения таблицы стилей в пределах большего файла. Подробности в разделе «Встроенные таблицы стилей» этой главы.
`type` обязательный	MIME-тип	Задает язык, на котором написана таблица стилей; обычно `text/xml` или `text/xsl` (см. обсуждение выше).
`title` необязательный	Строка	Если есть несколько инструкций обработки `<?xml-stylesheet?>`, каждой должно быть дано название, чтобы различать их. Тогда пользователь сможет выбирать, которая из таблиц стилей требуется. Например, могут быть специальные таблицы стилей для многотиражной печати или озвучивания документа.
`media` необязательный	Строка	Описание назначения вывода, например `"print"`, `"projection"` или `"aural"`. Список возможных значений определен в спецификации HTML 4.0. Это значение может использоваться для выбора таблицы стилей из имеющихся.
`charset` необязательный	Название кодировки символов, например `iso-8859-1`	Данный атрибут в таблицах стилей XSLT не представляет пользы, поскольку они, как и XML-документы, сами определяют кодировку символов.
`alternate` необязательный	`"yes"` или `"no"`	Если указано `"no"`, то это предпочтительная таблица стилей. Если указано `"yes"`, то это альтернативная таблица стилей.

Инструкция обработки <?xml-stylesheet?>, если она есть вообще, должна быть частью пролога документа, то есть находиться перед открывающим тегом элемента документа. Атрибут href задает местоположение таблицы стилей при помощи абсолютного или относительного URL. Например:

```
<?xml-stylesheet type="text/xsl" href="../style.xsl"?>
```

Как и в случае с CSS, можно иметь несколько инструкций обработки <?xml-stylesheet?>, соответствующих требуемым критериям. Хотя в этом случае могут обнаружиться вариации в различных программных продуктах. В некоторых из них это имеет такой же эффект, как при наличии одной таблицы стилей, импортирующей по порядку выбранные модули таблицы стилей при помощи элементов <xsl:import>. Стандартный способ использования этого состоит в том, чтобы иметь одну таблицу стилей, содержащую общие шаблонные правила, и более специфичные модули таблицы стилей, различающиеся с помощью псевдоатрибутов title или media. Однако некоторые процессоры, особенно MSXML3 от Microsoft, просто используют первую таблицу стилей и игнорируют остальные.

Использовать инструкцию обработки <?xml-stylesheet?> не обязательно, и большинство программ предлагает другие способы указать, какую таблицу стилей нужно применить к данному документу. Это особенно полезно, когда нужно применить таблицу стилей к XML-документу в броузере; указание этой инструкции обработки означает, что броузер может применять заданную по умолчанию таблицу стилей к документу, не требуя создания дополнительного сценария.

Очевидно, одна из причин отделения таблицы стилей от исходного XML-документа заключается в том, что одну и ту же информацию можно преобразовать или представить различными способами в зависимости от пользователя, его оборудования или особенностей режима доступа. Различные атрибуты инструкции обработки <?xml-stylesheet?> предназначены для определения правил, управляющих выбором соответствующей таблицы стилей. Данный механизм ориентирован на таблицы стилей, которые используются для отображения информации пользователям: это имеет меньше отношения к применению XSLT для выполнения преобразования данных.

Встроенные таблицы стилей

Есть одно исключение из правила, согласно которому модуль таблицы стилей должен быть XML-документом. Основной модуль таблицы стилей может быть **встроен** в другой XML-документ, обычно в тот документ, стиль которого он определяет.

Способность встраивать таблицы стилей внутрь исходного документа можно считать наследием CSS. Эта возможность может пригодиться, если есть отдельный документ, который вы хотите использовать независимо, но в большинстве ситуаций лучше применять внешнюю таблицу стилей, которая может пригодиться для многих исходных документов. Некоторым нравится встраивать

таблицы стилей, чтобы сократить время загрузки, но это может негативно отразиться на общей производительности, так как это означает, что броузер не сможет определить, что эта таблица стилей уже находится в его кэше.

> ***Не все XSLT-процессоры поддерживают встроенные таблицы стилей. Прежде чем применять их, сверьтесь с документацией используемого продукта.***

Самым внешним элементом таблицы стилей все еще является элемент `<xsl:stylesheet>` или `<xsl:transform>`, но он уже не будет самым внешним элементом XML-документа (то есть элементом документа). Элемент `<xsl:stylesheet>` для его идентификации обычно имеет атрибут `id`, и из содержащего его документа на него ссылаются с помощью инструкции обработки `<?xml-stylesheet?>`, например:

Пример: Встроенная таблица стилей

В данном примере показана таблица стилей, встроенная в исходный XML-документ, содержащий список книг.

Исходный документ

Файл данных `embedded.xsl` содержит и исходный документ, и таблицу стилей, как показано ниже:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="#style1"?>
<книги>
   <книга категория="справочник">
      <автор>Найджел Рис</автор>
      <название>Поговорки века</название>
      <цена>8.95</цена>
   </книга>
   <книга категория="беллетристика">
      <автор>Ивлин Во</автор>
      <название>Меч чести</название>
      <цена>12.99</цена>
   </книга>
   <книга категория="беллетристика">
      <автор>Герман Мелвилл</автор>
      <название>Моби Дик</название>
      <цена>8.99</цена>
   </книга>
   <книга категория="беллетристика">
      <автор>Дж. Р. Р. Толкиен</автор>
      <название>Властелин колец</название>
      <цена>22.99</цена>
   </книга>

   <xsl:stylesheet id="style1" version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="xsl:stylesheet"/>
```

```
      <xsl:template match="книги">
      <html><body>
         <h1>Список книг</h1>
         <table border="1">
            <xsl:apply-templates/>
         </table>
      </body></html>
   </xsl:template>

   <xsl:template match="книга">
      <tr><xsl:apply-templates/></tr>
   </xsl:template>

   <xsl:template match="автор | название | цена">
      <td><xsl:value-of select="."/></td>
   </xsl:template>

   </xsl:stylesheet>
</книги>
```

Можно запустить эту таблицу стилей в Saxon, применяя следующую команду (на момент написания книги она не работала в Saxon 6):

```
saxon -a embedded.xsl
```

Параметр `-a` указывает процессору Saxon искать инструкцию обработки `<?xml-stylesheet?>` в данном исходном документе и обрабатывать его, используя эту таблицу стилей. Saxon не позволяет (при использовании командной строки) определять критерии выбора конкретной таблицы стилей, поэтому, если имеется несколько таблиц, он использует составную таблицу стилей, в которую импортированы все имеющиеся.

Конечный документ

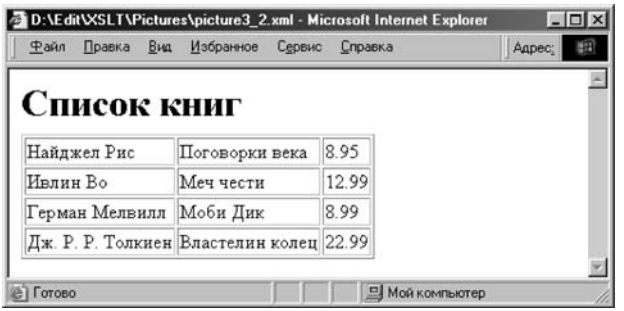


Рис. 3.2. Вывод списка книг в окне броузера

Обратите внимание на пустое шаблонное правило, которое соответствует элементу `<xsl:stylesheet>`. Оно необходимо, поскольку без него таблица стилей пыталась бы обработать саму себя, наряду с остальной частью документа. Пустое шаблонное правило гарантирует, что когда встречается элемент

<xsl:stylesheet>, вывод не генерируется, а его потомки не обрабатываются. Нужно соблюдать осторожность, чтобы также избежать совпадений с другими элементами в таблице стилей. Например, если таблица стилей ищет названия книг при помощи выражения типа «//название», она случайно может спутать его с конечным литеральным элементом <название> во встроенной таблице стилей.

Встроенный модуль таблицы стилей обычно используется как основной модуль таблицы стилей. Стандарт не указывает явно, может ли встроенная таблица стилей быть включена или импортирована в другую. На практике поддержка таких возможностей зависит от используемых программ, однако в документации к немногим из них говорится о встроенных таблицах стилей.

Если объявления пространства имен находятся вне встроенной таблицы стилей, их область действия распространяется и на вложенную таблицу стилей, что может привести к копированию в конечное дерево дополнительных узлов пространств имен. Можно избежать этого, используя атрибут exclude-result-prefixes для элемента <xsl:stylesheet>.

Элементы верхнего уровня

Термин **элемент верхнего уровня** употребляется для элемента, который является непосредственным потомком элемента <xsl:stylesheet> или <xsl:transform>. Точнее было бы называть его элементом второго уровня, но в стандарте принят термин **элемент верхнего уровня**.

Текстовые узлы не разрешается иметь в качестве непосредственных потомков элемента <xsl:stylesheet> или <xsl:transform>, если только они не состоят из одних пробельных символов. Инструкции обработки и комментарии могут встречаться – XSLT-процессор игнорирует их.

Элементы верхнего уровня могут появляться в таблице стилей в любом порядке, за исключением того, что любые элементы <xsl:import>, если они есть, должны предшествовать остальным. В большинстве случаев порядок элементов не имеет никакого значения; однако, если есть противоречивые определения, XSLT-процессоры либо сообщают об ошибке, либо выбирают последнее из этих определений. Когда создается переносимая таблица стилей, нельзя полагаться на какое-то определенное поведение XSLT-процессора в подобной ситуации, поэтому следует удостовериться, что противоречивых определений нет.

Элементы, которые могут встретиться в верхнем уровне, делятся на три категории:

- Элементы верхнего уровня, определенные в спецификации XSLT
- Элементы верхнего уровня, определяемые производителем
- Элементы верхнего уровня, определяемые пользователем

Рассмотрим подробнее эти три категории.

Элементы верхнего уровня, определенные в спецификации XSLT

К элементам верхнего уровня, определенным в спецификации XSLT, относятся следующие элементы:

```
<xsl:attribute-set>          <xsl:namespace-alias>        <xsl:script>
<xsl:decimal-format>         <xsl:output>                 <xsl:strip-space>
<xsl:import>                 <xsl:param>                  <xsl:template>
<xsl:include>                <xsl:preserve-space>         <xsl:variable>
<xsl:key>
```

Значение этих элементов объясняется в главе 4. Никакой другой элемент XSLT (то есть никакой другой элемент с URI пространства имен *http://www.w3.org/1999/XSL/Transform*) не может использоваться на верхнем уровне.

Элементы верхнего уровня, определяемые производителем

Элемент верхнего уровня, определяемый производителем, должен принадлежать пространству имен с непустым URI, отличному от пространства имен XSLT. В общем случае это будет пространство имен, определенное компанией-разработчиком: например, для продукта Saxon соответствующим URI пространства имен является `http://icl.com/saxon`. Значение элементов в этой категории полностью зависит от компании-разработчика, хотя спецификация требует, чтобы такие элементы не изменяли значения стандартных конструкций XSLT. Имейте в виду, что эти элементы верхнего уровня формально не являются элементами расширения, и для того, чтобы они действовали, не требуется объявлять их пространства имен в атрибуте `extension-element-prefixes`.

Некоторые компании-разработчики вводят элементы верхнего уровня, позволяющие определять внешние функции, которые могут быть вызваны из выражений XPath в таблице стилей (но поскольку теперь XSLT 1.1 поддерживает элемент `<xsl:script>`, такие расширения должны постепенно исчезать). Другие могут использовать дополнительные элементы для указания параметров отладки или трассировки. Подобные расширения описаны в соответствующих приложениях по конкретным программным продуктам.

Элементы верхнего уровня, определяемые пользователем

Элемент верхнего уровня, определяемый пользователем, также должен принадлежать пространству имен с непустым URI, отличному от пространства имен XSLT. Также желательно, чтобы URI отличался от URI пространства имен, используемого какой-либо компанией-разработчиком. Эти элементы игнорируются XSLT-процессором.

В XSLT 1.0 элементы верхнего уровня, определяемые пользователем, удобны для вставки подставляемых данных, сообщений об ошибках и т. п. Мож-

но обращаться к этим элементам изнутри таблицы стилей, обрабатывая таблицу стилей как дополнительный исходный документ и загружая ее с помощью функции document(), которая описана в главе 7. Если первый параметр этой функции – пустая строка, это интерпретируется как ссылка на модуль таблицы стилей, в котором находится функция document().

Так, например, если таблица стилей содержит следующий элемент верхнего уровня, определяемый пользователем:

```
<user:data xmlns:user="http://acme.com/">
   <message nr="1">Исходный документ пуст</message>
   <message nr="2">Неверная дата</message>
   <message nr="3">Значение переменной sales не является числом</message>
</user:data>
```

тогда та же самая таблица стилей может содержать именованный шаблон, чтобы отображать нумерованные сообщения следующим образом:

```
<xsl:template name="display-message">
   <xsl:param name="message-nr"/>
   <xsl:message xmlns:user="http://acme.com/">
      <xsl:value-of
      select="document('')/*/user:data/message[@nr=$message-nr]"/>
   </xsl:message>
</xsl:template>
```

Элемент <xsl:value-of> вычисляет строковое значение выражения XPath в своем атрибуте select и записывает его в конечное дерево. В данном случае выражение XPath – это выражение пути, начинающееся с вызова функции «document('')», которое выбирает корневой узел модуля таблицы стилей, затем идет символ «*», который выбирает его первого потомка (элемент <xsl:stylesheet>), затем идет «user:data» – выбирает элемент <user:data>, затем «message[@nr=$message-nr]» – выбирает элемент <message>, атрибут nr которого равен значению параметра $message-nr в таблице стилей.

Этот именованный шаблон можно вызвать из любого места таблицы стилей, используя такую последовательность:

```
<xsl:if test="string(number(@sales))='NaN'">
   <xsl:call-template name="display-message">
       <xsl:with-param name="message-nr" select="3"/>
    </xsl:call-template>
</xsl:if>
```

Элемент <xsl:if> проверяет, является ли атрибут sales текущего исходного элемента числом: если нет, то результатом преобразования его в число, а затем в строковые данные будет строка NaN, что означает «не-число». В этом случае программа вызовет шаблон, который был определен ранее для отображения сообщения: «Значение переменной sales не является числом». (В стандарте не определено, куда должны выводиться сообщения, сгенерированные при помощи элемента <xsl:message>. Это может быть всплывающее окно или просто сообщение в регистрационном журнале веб-сервера.)

Преимущество такой методики в том, что все сообщения при этом собираются в одном месте, что облегчает поддержку. Кроме того, ее можно легко расширить, чтобы использовать различные наборы сообщений в зависимости от предпочитаемого пользователем языка.

В XSLT 1.1 нужда в этой методике отпадет, так как станет удобнее указывать постоянные данные в определении глобальной переменной. Вместо записи:

```
<user:data xmlns:user="http://acme.com/">
   <message nr="1">Исходный документ пуст</message>
   <message nr="2">Неверная дата</message>
   <message nr="3">Значение переменной sales не является числом</message>
</user:data>
```

можно будет написать:

```
<xsl:variable name="data">
   <message nr="1">Исходный документ пуст</message>
   <message nr="2">Неверная дата</message>
   <message nr="3">Значение переменной sales не является числом</message>
</xsl:variable>
```

а вместо:

```
<xsl:value-of select="document('')/*/user:data/message[@nr=$message-nr]"/>
```

можно написать:

```
<xsl:value-of select="$data/message[@nr=$message-nr]"/>
```

Упрощенные таблицы стилей

Упрощенная таблица стилей использует упрощенный синтаксис, в котором опускаются элемент `<xsl:stylesheet>` и все элементы верхнего уровня.

XSLT-спецификация называет такой прием **конечным литеральным элементом в роли таблицы стилей**. Его задача – позволить людям с навыками создания HTML, но без опыта программирования писать простые таблицы стилей с минимальным обучением этому. Упрощенная таблица стилей имеет каркас, который выглядит как целевой документ (обычно HTML, но не обязательно), и использует инструкции XSLT для заполнения изменяющихся фрагментов.

Модуль таблицы стилей интерпретируется как упрощенная таблица стилей, если самый внешний элемент – не `<xsl:stylesheet>` или `<xsl:transform>`. Самый внешний элемент может иметь любое название при условии, что оно не принадлежит пространству имен XSLT. Он все равно должен содержать объявление пространства имен XSLT и должен иметь атрибут `xsl:version`. Для XSLT 1.1 его значение должно быть «1.0» или «1.1» («1.1» используется, если в таблице стилей задействованы особенности, определенные в XSLT 1.1, а «1.0» используется, если нужно, чтобы она работала с процессорами XSLT 1.0). Когда атрибут `xsl:version` не равен «1.0» или «1.1», включается режим совместимости с последующими версиями. Он обсуждается позже в этой главе (на стр. 159).

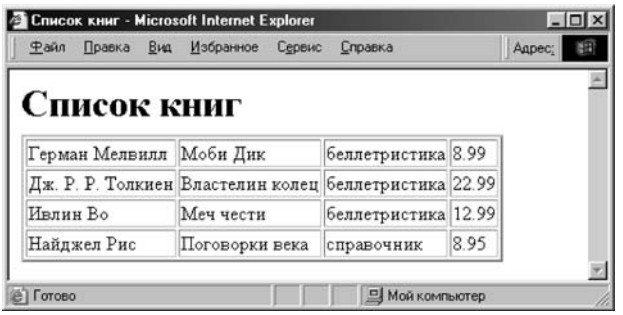
Пример: Упрощенная таблица стилей

В этом примере показана таблица стилей в форме каркаса HTML-страницы с вложенными в нее инструкциями XSLT для извлечения данных из исходного документа. Английская версия этой таблицы стилей (как и многих других примеров) может быть загружена с *http://www.wrox.com*. Имя файла – `simplified.xsl`, и его можно использовать вместе с файлом данных `books.xml`, который был создан во второй главе.

Мы же рассмотрим здесь таблицу стилей `упрощенная.xsl` для файла `книги.xml`, которая выглядит следующим образом:

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0">
<head>
<title>Список книг</title></head>
<body>
<h1>Список книг</h1>
   <table border="2">
   <xsl:for-each select="//книга">
      <xsl:sort select="автор"/>
      <tr>
        <td><xsl:value-of select="автор"/></td>
        <td><xsl:value-of select="название"/></td>
        <td><xsl:value-of select="@категория"/></td>
        <td><xsl:value-of select="цена"/></td>
      </tr>
   </xsl:for-each>
   </table>
</body>
</html>
```

Если применить ее к файлу `книги.xml` (о котором говорится в примере из главы 2), выводом будет отсортированная таблица, показывающая книги следующим образом:



Герман Мелвилл	Моби Дик	беллетристика	8.99
Дж. Р. Р. Толкиен	Властелин колец	беллетристика	22.99
Ивлин Во	Меч чести	беллетристика	12.99
Найджел Рис	Поговорки века	справочник	8.95

Рис. 3.3. *Вывод отсортированного списка книг*

Упрощенная таблица стилей эквивалентна таблице стилей, в которой самый внешний элемент (обычно элемент <html>) входит сначала в элемент <xsl:template> с атрибутом «match="/"», а тот, в свою очередь – в элемент <xsl:stylesheet>. Атрибут xsl:version самого внешнего элемента становится атрибутом version элемента <xsl:stylesheet>. Таким образом, расширенная форма вышеупомянутого примера выглядела бы следующим образом:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
<html>
<head><title>Список книг</title></head>
<body>
<h1>Список книг</h1>
   <table border="2">
   <xsl:for-each select="//книга">
      <xsl:sort select="автор"/>
      <tr>
        <td><xsl:value-of select="автор"/></td>
        <td><xsl:value-of select="название"/></td>
        <td><xsl:value-of select="@категория"/></td>
        <td><xsl:value-of select="цена"/></td>
      </tr>
   </xsl:for-each>
   </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Задача атрибута «match="/"» – задать шаблонное правило, которое будет задействовано первым, как только будет активизирована таблица стилей. Как было показано в главе 2, обработка всегда начинается с корневого узла исходного дерева документа, и любое шаблонное правило, соответствующее корневому узлу, будет вызвано первым. Образец соответствия «/» обозначает корневой узел. В упрощенной таблице стилей это будет единственным применяемым шаблонным правилом.

Упрощенная таблица стилей не может делать многих вещей, потому что она не может содержать никаких XSLT-элементов верхнего уровня. Например, упрощенная таблица стилей не может включать или импортировать другую таблицу стилей, не может иметь глобальных переменных или параметров и не может определять ключи. Однако когда эти дополнительные возможности необходимы, всегда можно усложнить таблицу стилей, окружив ее элементами <xsl:stylesheet> и <xsl:template>.

Теоретически возможно, чтобы таблица стилей включала или импортировала упрощенную таблицу стилей, которая была бы расширена, как описано выше, но обычно это не делается.

Тела шаблонов

Содержимое элемента <xsl:template> после всех определений параметров, содержащихся в элементах <xsl:param>, является **телом шаблона**. В спецификации XSLT это называется просто шаблоном, но так как этот термин часто используется нестрого для обозначения элемента <xsl:template>, в данной книге предложен термин **тело шаблона**.

Для многих других элементов XSLT также принято считать их содержимое телом шаблона. Это просто имеет отношение к правилам, определяющим, какое содержимое может встретиться в данном элементе, но вовсе не означает, что обязательно должен присутствовать элемент <xsl:template>. Например, содержимое элементов <xsl:variable> или <xsl:if> следует тем же самым правилам, что и содержимое <xsl:template> (игнорирование элементов <xsl:param>), и оно также является телом шаблона. Из этого следует, что одно тело шаблона может содержаться внутри другого. Например, рассмотрим следующее шаблонное правило:

```
<xsl:template match="para">
   <xsl:if test="position()=1">
      <hr/>- o - 0 - o -<hr/>
   </xsl:if>
   <xsl:apply-templates/>
   <xsl:if test="position()=last()">
      <hr/>- o - 0 - o -<hr/>
   </xsl:if>
</xsl:template>
```

Изображенное в виде дерева с использованием системы обозначений, введенной в главе 2, это правило имеет структуру, показанную ниже на схеме. Пунктирными рамками обведены три тела шаблонов. В телах шаблонов в этом дереве есть три вида узлов: текстовые узлы, инструкции XSLT (типа <xsl:if>) и конечные литеральные элементы (например, <hr>), являющиеся элементами, которые будут записаны в конечное дерево (рис. 3.4).

В XSLT тело шаблона – самый близкий эквивалент блочным или составным операторам в языках программирования с блочной структурой, таких как C или Java; и, подобно блокам в C или Java, оно определяет область действия любых локальных переменных, объявленных в рамках блока.

Тело шаблона – это последовательность одноуровневых узлов дерева в таблице стилей. Узлы комментариев и инструкций обработки допустимы, но XSLT-процессор игнорирует их. Интерес представляют только текстовые узлы и узлы элементов.

Когда тело шаблона применяется, текстовые узлы, находящиеся в нем, копируются в конечное дерево. Однако текстовые узлы в теле шаблона, состоящие только из пробельных символов, будут игнорироваться, если только включающий их элемент не содержит атрибут xml:space, который определяет такие узлы как существенные.

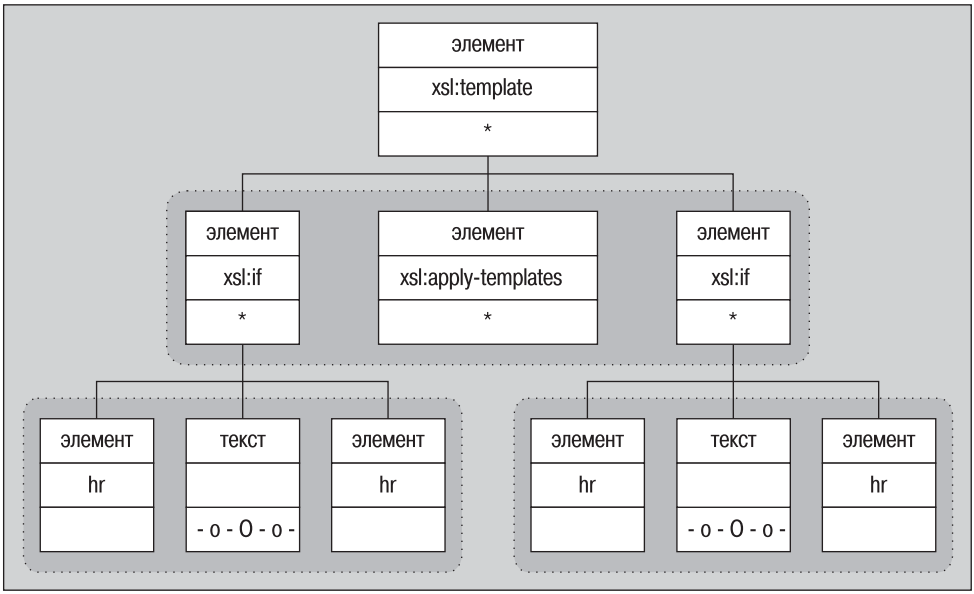


***Рис. 3.4.** Схема тела шаблона в виде дерева*

Текстовые узлы, содержащие только пробельные символы, существенны также, если они являются содержимым элемента `<xsl:text>`*, но в этом случае они не входят в тело шаблона.*

Узлы элементов в теле шаблона могут быть трех типов: инструкции XSLT, элементы расширения и конечные литеральные элементы. Они обсуждаются в следующих трех разделах.

Инструкции XSLT

Инструкциями XSLT являются следующие элементы:

```
<xsl:apply-imports>          <xsl:copy>                  <xsl:message>
<xsl:apply-templates>        <xsl:copy-of>               <xsl:number>
<xsl:attribute>              <xsl:document>*             <xsl:processing-instruction>
<xsl:call-template>          <xsl:element>               <xsl:text>
<xsl:choose>                 <xsl:fallback>              <xsl:value-of>
<xsl:comment>                <xsl:for-each>              <xsl:variable>
                             <xsl:if>
```

* Элемент `<xsl:document>` введен в XSLT 1.1; все остальные определены и в XSLT 1.0.

Никакой другой элемент из пространства имен XSLT не может находиться в теле шаблона. Другие элементы XSLT, например `<xsl:with-param>`, `<xsl:sort>` и `<xsl:otherwise>`, не считаются инструкциями, потому что они не могут находиться прямо в теле шаблона – они могут встречаться только в определенных контекстах. Элемент `<xsl:param>` – несколько аномальный, поскольку

он может быть потомком элемента <xsl:template>, но он должен располагаться перед другими элементами и, следовательно, не входит в тело шаблона. Таким образом, он не классифицируется как инструкция.

Значение всех этих инструкций XSLT будет раскрыто в главе 4.

Если в теле шаблона встречается неизвестный элемент из пространства имен XSLT, предпринимаемое действие зависит от того, включен ли **режим опережающей совместимости**. Он обсуждается позже в этой главе (на стр. 159).

Элементы расширения

Элемент расширения – это инструкция, определяемая компанией-разработчиком или пользователем, в отличие от инструкций, определенных в стандарте XSLT. В обоих случаях она считается элементом расширения, потому что принадлежит пространству имен, указанному в атрибуте extension-element-prefixes содержащего ее элемента <xsl:stylesheet> или в атрибуте xsl:extension-element-prefixes самого элемента, или содержащего ее конечного литерального элемента, или элемента расширения.

На практике элементы расширения компаниями-разработчиками определяются чаще, чем пользователями. Для XSLT 1.0 некоторые компании-разработчики ввели элементы расширения для направления вывода таблицы стилей в несколько выходных файлов (в XSLT 1.1 это обеспечивается стандартным средством – инструкцией <xsl:document>). Другой пример: в процессоре Saxon введен элемент расширения <saxon:group> для выполнения группировки (подобно конструкции GROUP BY в SQL) – возможность, которая стандартными средствами XSLT производится неуклюже и неэффективно.

Не все продукты позволяют пользователям определять собственные элементы расширения, а в тех случаях, когда это возможно, требуется порой довольно сложное программирование на системном уровне. Обычно проще вызывать пользовательский код, используя функции расширения, которые намного легче писать. В XSLT 1.1 станет возможно даже писать функции расширения, переносимые между XSLT-процессорами; во всяком случае между теми, которые поддерживают Java или JavaScript.

Следующий пример показывает элемент <acme:instruction>, который считался бы конечным литеральным элементом, если бы не атрибут xsl:extension-element-prefixes, который превращает его в элемент расширения:

```
<acme:instruction
   xmlns:acme="http://acme.co.jp/xslt"
   xsl:extension-element-prefixes="acme"/>
```

Способы реализации новых элементов расширения не определены в спецификации XSLT и, очевидно, различны у всех компаний-разработчиков. Фактически, от XSLT-процессоров не требуется обеспечивать механизм для определения новых элементов расширения. Единственное требование – они должны распознать элемент расширения, когда он встречается, и отличать его от конечного литерального элемента.

Что случится, если таблица стилей, которая использует элемент расширения, определенный, например, в процессоре Xalan, запускается с другим процессором, к примеру, от Microsoft? Если процессор сталкивается с элементом расширения, который он не может обработать (обычно потому, что он был введен другой компанией-разработчиком), он должен предпринять действие, четко определенное в стандарте XSLT: если автор таблицы стилей определил действие `<xsl:fallback>`, он должен выполнить его, если нет – должен сообщить об ошибке. Что он не должен делать – это обрабатывать элемент расширения как конечный литеральный элемент и копировать его в конечное дерево.

> *Инструкция* `<xsl:fallback>` *позволяет указать, как XSLT-процессор должен поступать с элементами расширения, которые он не опознает. Это описано более подробно далее в этой главе на стр. 159, а полные спецификации находятся в главе 4.*

Любой элемент, находящийся в теле шаблона и не являющийся инструкцией XSLT или элементом расширения, интерпретируется как **конечный литеральный элемент** (например, элементы `<hr/>` в предыдущем примере тел шаблона). Когда применяется тело шаблона, конечный литеральный элемент копируется в конечное дерево.

Таким образом, в действительности в теле шаблона есть два вида узлов: инструкции и данные. Инструкции подчиняются правилам для конкретных инструкций, а узлы данных (текстовые узлы и конечные литеральные элементы) копируются в конечное дерево.

Конечные литеральные элементы играют важную роль в структуре таблицы стилей, поэтому в следующем разделе они обсуждаются более подробно.

Конечные литеральные элементы

Конечный литеральный элемент – это элемент в теле шаблона в таблице стилей, который не может интерпретироваться как инструкция и поэтому обрабатывается как данные, то есть копируется в текущий вывод.

Конечные литеральные элементы описываются здесь по той же схеме, которая используется в главе 4 для описания элементов XSLT.

Формат

Расположение

Фиксированный конечный элемент всегда находится прямо в теле шаблона.

Атрибуты

Имя	Значение	Смысл
`xsl:exclude-result-prefixes` необязательный	Список префиксов пространств имен, разделенных пробельными символами (см. замечание ниже)	Каждый префикс из списка должен задавать пространство имен, которое действует в данной точке модуля таблицы стилей. Заданное пространство имен не должно копироваться в конечное дерево.

Имя	Значение	Смысл
xsl:extension-element-prefixes необязательный	Список префиксов пространств имен, разделенных пробельными символами (см. замечание ниже)	Каждый префикс из списка должен задавать пространство имен, которое действует в данной точке модуля таблицы стилей. Элементы, которые являются потомками этого конечного литерального элемента и имена которых находятся в одном из заданных пространств имен, обрабатываются как элементы расширения, а не как конечные литеральные элементы.
xsl:version необязательный	Число	Для процессора XSLT 1.0: если значение «1.0», то любой элемент XSLT, который является потомком данного конечного литерального элемента, должен быть элементом XSLT, определенным в Рекомендации версии 1.0. Если значение – любое другое, становится возможным режим совместимости с последующими версиями (см. ниже). Для процессора XSLT 1.1: значения «1.0» и «1.1» – синонимы; оба позволяют любой элемент XSLT, определенный в спецификации XSLT 1.1.
xsl:use-attribute-sets необязательный	Список полных имен, задающих именованные элементы <xsl:attribute-set>, разделенные пробельными символами (см. замечание ниже)	Атрибуты, определенные в именованных наборах атрибутов, применяются и копируются в вывод как атрибуты данного конечного литерального элемента.
Другие атрибуты необязательные	Шаблон значения атрибутов	Любые выражения XPath в значении, находящиеся между фигурными скобками, вычисляются, а результирующие строковые данные образуют значение атрибута, копируемое в текущий вывод. Шаблоны значений атрибутов описаны далее в этой главе на стр. 148.

Примечание

Некоторые атрибуты имеют форму списков, разделенных пробельными символами. Это просто списки имен (или префиксов), в которых разные имена разделены любым из определенных в XML пробельных символов: табуляции, возврата каретки, перехода на новую строку или пробела. Например, можно написать:

```
<TD xsl:use-attribute-sets="blue italic centered"/>
```

Здесь имена blue, italic и centered должны соответствовать именам элементов <xsl:attribute-set> в других местах таблицы стилей.

Содержимое

Содержимое конечного литерального элемента – тело шаблона. Таким образом, сюда могут входить инструкции XSLT, элементы расширения, конечные литеральные элементы и/или текстовые узлы.

Использование

Конечный литеральный элемент копируется в конечное дерево, а его содержимое подвергается обработке.

Рассмотрим тело шаблона, содержащее только конечный литеральный элемент:

```
<TD>Код товара</TD>
```

В этом случае элемент <TD> будет записан в конечное дерево с дочерним текстовым узлом, содержимое которого – «Код товара». Когда конечное дерево выводится в XML- или HTML-файл, текст регенерируется в той форме, в какой он содержался в таблице стилей, или в эквивалентной. Нет гарантии, что он будет идентичен вплоть до каждого символа: в частности, процессор может добавлять или удалять пробельные символы в пределах тегов, или он может представлять символы, используя ссылки на символы или сущности.

Если конечный литеральный элемент имеет содержимое, то содержимым должно быть другое тело шаблона, и это тело шаблона тоже обрабатывается: любые узлы, сформированные в ходе этого процесса в конечном дереве, станут потомками элемента, созданного из конечного литерального элемента.

Например, если тело шаблона следующее:

```
<TD><xsl:value-of select="."/></TD>
```

тогда при обработке элемента <TD> его содержимое также будет подвергнуто обработке. В этом случае содержимым является тело шаблона, состоящее из одной инструкции XSLT, причем эта инструкция применяется, создавая текстовый узел, который будет потомком элемента <TD> в конечном дереве. Инструкция <xsl:value-of select="."> выводит строковое значение текущего узла в исходном дереве. Так, если бы этим значением было «$83.99», то результат был бы:

```
<TD>$83.99</TD>
```

Соблазнительно рассматривать это как последовательность из трех шагов:

- Открывающий тег <TD> заставляет записать в вывод открывающий тег <TD>
- Элемент <xsl:value-of> вычисляется, а результат («$83.99») записывается в вывод
- Закрывающий тег </TD> заставляет записать в вывод закрывающий тег </TD>

Однако это – не истинная картина того, что происходит, и лучше не думать об этом таким образом, иначе появятся вопросы: например, как задержать запись конечного тега до того момента, пока во вводе не встретится некоторое условие.

Процесс преобразования приводит к записи узлов в конечное дерево, а не тегов в последовательный файл. Элемент `<TD>` ***в таблице стилей заставляет записать в конечное дерево элемент*** `<TD>`***. Нельзя записать в дерево половину узла: запись открывающего и закрывающего тегов не происходит как две отдельные операции. Теги*** `<TD>` ***и*** `</TD>` ***генерируются, только когда конечное дерево сериализуется как XML или HTML.***

Следующая схема поможет пояснить это.

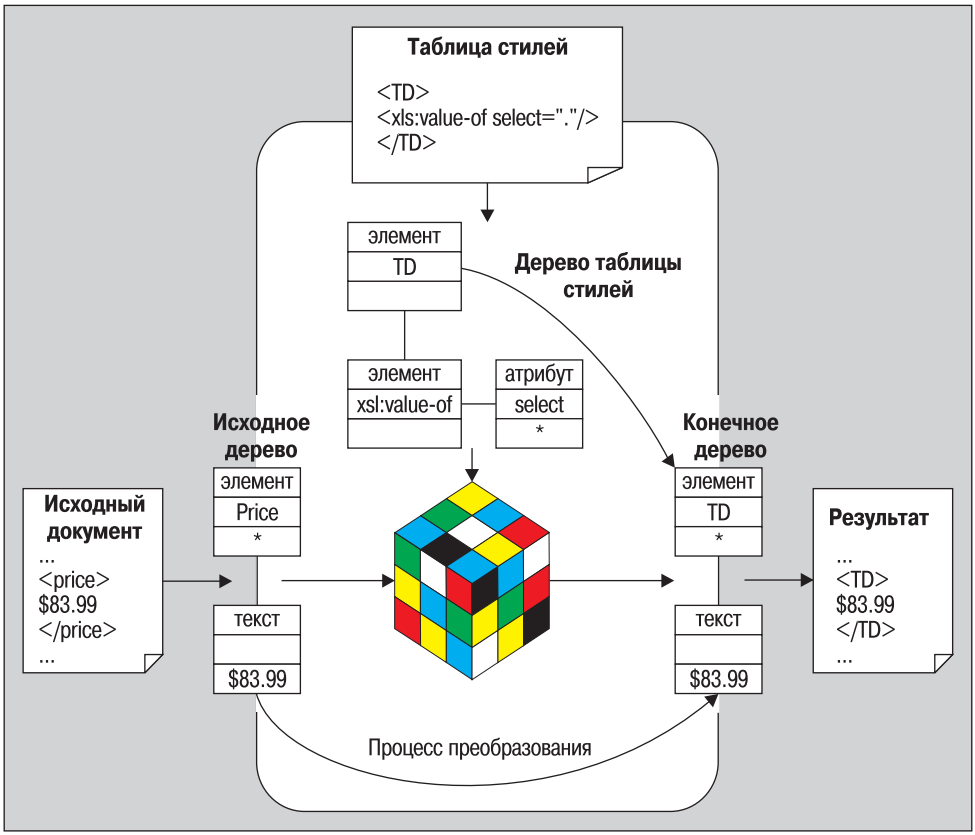


Рис. 3.5. *Схема обработки элемента* `<TD>` *при XSLT-преобразовании*

Если имеет значение, в каком месте вывода должны появиться теги, стоит сделать набросок требуемой формы конечного дерева, а затем продумать, как написать таблицу стилей, чтобы произвести на дереве требуемые узлы. Так как элемент в конечном дереве будет производиться всегда в результате обработки одного тела шаблона в таблице стилей, это все равно, что найти ответ на вопрос, какое условие в исходном дереве должно генерировать данный конечный элемент.

Предположим, например, что нужно сформировать HTML-таблицу с пятью столбцами, выстраивая элементы `<item>` из исходного XML по пять в строке.

Тогда условием в исходном XML, которое заставит генерировать строку вывода, будет элемент <item>, занимающий позиции 1, 6, 11 и т. д. Логическая схема может быть написана следующим образом:

```
<xsl:template match="item[position() mod 5 = 1]">
<tr>
  <xsl:for-each select=". | following-sibling::item[position() &lt; 5]">
    <td><xsl:value-of select="."/></td>
  </xsl:for-each>
</tr>
</xsl:template>

<xsl:template match="item"/>
```

Первое шаблонное правило соответствует **элементам** <item>, которые должны находиться в начале новой строки; оно выводит элемент <tr> и пять элементов <td>, соответствующих этому <item> и еще четырем следующим одноуровневым элементам. Второе правило соответствует **элементам** <item>, которые **не** должны находиться в начале новой строки, и оно не делает ничего, потому что эти элементы уже обработаны в соответствии с первым шаблонным правилом.

Атрибуты конечного литерального элемента

Если конечный литеральный элемент имеет другие атрибуты, отличающиеся от специальных xsl-атрибутов из списка, приведенного выше, то эти атрибуты также копируются в текущий вывод. Так, если тело шаблона содержит:

```
<TD><IMG src="picture1.gif"/></TD>
```

то вывод будет содержать копию всей этой конструкции. Внешний элемент <TD>, как и прежде, копируется в конечное дерево, но на этот раз его содержимое состоит из другого конечного литерального элемента, <IMG>, который копируется в конечное дерево как потомок элемента <TD> вместе со своим атрибутом src. В этот раз и дерево таблицы стилей, и конечное дерево имеют вид, показанный ниже:

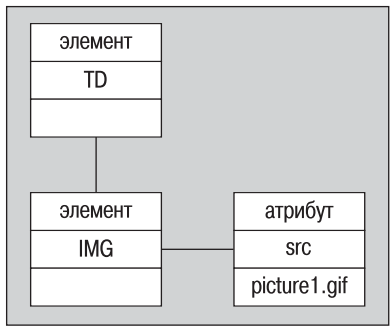


***Рис. 3.6.** Копирование конечного литерального элемента в конечное дерево вместе с атрибутами*

Если в значении атрибута конечного литерального элемента есть фигурные скобки («{» и «}»), то он считается **шаблоном значения атрибута** (подробнее обсуждается в следующем разделе). Текст между фигурными скобками рассматривается как выражение XPath, и вычисляется его строковое значение; атрибут, записываемый в конечное дерево, содержит вместо выражения это строковое значение. Например, предположим, что к файлу `книги.xml`, используемому ранее, применяется следующий шаблон:

```
<xsl:template match="/">
<xsl:for-each select="//книга">
<div id="div{position()}">
            <xsl:value-of select="название"/>
</div>
</xsl:for-each>
</xsl:template>
```

Поскольку функция `position()` по мере перемещения по набору книг принимает значения 1, 2, 3 и 4, вывод будет иметь следующий вид:

```
<div id="div1">Поговорки века</div>
<div id="div2">Меч чести</div>
<div id="div3">Моби Дик</div>
<div id="div4">Властелин Колец</div>
```

Можно также генерировать атрибуты для конечного литерального элемента двумя другими способами:

- Атрибут может быть сгенерирован инструкцией `<xsl:attribute>`. Эта инструкция необязательно должна присутствовать текстуально в пределах содержимого конечного литерального элемента в таблице стилей, но она должна быть применена до генерирования любых дочерних узлов (элементов или других потомков).

  Необходимость этого правила заключается в том, что оно позволяет XSLT-процессору избежать формирования конечного дерева в памяти. Многие процессоры сериализуют синтаксис XML прямо в выходной файл по мере формирования узлов, а правило, требующее, чтобы атрибуты формировались раньше дочерних элементов или текстовых узлов, гарантирует такую возможность. Правило гласит, что выполнение таблицы стилей должно быть последовательным, но, конечно, приемлема любая стратегия выполнения, дающая такой же эффект.

- Набор атрибутов может быть сгенерирован при помощи именованного набора атрибутов. Конечный литеральный элемент должен содержать атрибут `xsl:use-attribute-sets`, называющий наборы атрибутов, которые будут встроены: эти названия должны соответствовать элементам `<xsl:attribute-set>` на верхнем уровне таблицы стилей. Каждый именованный набор атрибутов содержит последовательность инструкций `<xsl:attribute>`, и они заставляют добавлять атрибут к сгенерированному элементу, как если бы они находились прямо в содержимом конечного литерального элемента. Именованные наборы атрибутов полезны для создания специализированных коллекций атрибутов, например назва-

ний, цветов и размеров шрифтов, которые вместе определяют стиль, неоднократно используемый в выходных документах; это прямые аналоги стилей, имеющихся в более простых языках типа CSS.

Атрибуты добавляются к сформированному узлу элемента в определенном порядке: сначала атрибуты, включенные с помощью `xsl:use-attribute-sets`, затем атрибуты, имеющиеся непосредственно у конечного литерального элемента и, наконец, атрибуты, добавленные с помощью инструкции `<xsl:attribute>`. Смысл такого порядка в том, что если добавляются два или более атрибутов с одним и тем же названием, то учитывается последний из них. Это не означает, что атрибуты обязательно сохранят тот же порядок и при сериализации конечного дерева.

Пространства имен для конечного литерального элемента

Узлы пространства имен конечного литерального элемента также копируются в текущий вывод. Часто это вносит некоторую путаницу. Конечный литеральный элемент в таблице стилей будет иметь один узел пространства имен для каждого объявления пространства имен, в области действия которого он находится, то есть для каждого атрибута `xmlns` или атрибута `xmlns:*` конечного литерального элемента непосредственно или любого из его родительских элементов в таблице стилей. Единственное исключение – атрибут `«xmlns=""»`, он не действует как объявление пространства имен, а отменяет любое более раннее объявление пространства имен по умолчанию.

Элемент конечного дерева, созданный из конечного литерального элемента, гарантированно имеет узел пространства имен для каждого узла пространств имен, имевшегося у конечного литерального элемента в таблице стилей, исключая следующее:

- узел пространства имен для URI пространства имен XSLT `http://www.w3.org/1999/XSL/Transform` не копируется.
- узел пространства имен для пространства имен, объявленного как пространство имен элементов расширения, не копируется. Пространство имен объявляется пространством имен элементов расширения путем включения его префикса в значение атрибута `extension-element-prefixes` элемента `<xsl:stylesheet>` или в значение атрибута `xsl:extension-element-prefixes` данного конечного литерального элемента, любого родительского конечного литерального элемента или элемента расширения.
- узел пространства имен для исключенного пространства имен не копируется. Пространство имен объявляется исключенным пространством имен путем включения его префикса в значение атрибута `exclude-result-prefixes` элемента `<xsl:stylesheet>` или в значение атрибута `xsl:exclude-result-prefixes` данного конечного литерального элемента или любого родительского конечного литерального элемента.

Рассмотрим, например, следующую таблицу стилей:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
   version="1.1"
   xmlns:Date="java:java.util.Date"
>

<xsl:script implements-prefix="Date" language="java"
    src="java:java.util.Date"/>

<xsl:template match="/"
      xmlns = "urn:acme-com:gregorian">
    <date><xsl:value-of select="$today"/></date>
</xsl:template>

<xsl:param name="today" select="Date:toString(Date:new())"/>

</xsl:stylesheet>
```

Элемент <date> находится в области действия трех пространств имен, а именно: пространство имен XSLT, пространство имен «java:java.util.Date» и пространство имен по умолчанию «urn:acme-com:gregorian». Пространство имен XSLT не копируется в конечное дерево, а другие два – копируются. Таким образом, элемент <date>, добавляемый в конечное дерево, гарантированно будет иметь эти два пространства имен – «java:java.util.Date» и «urn:acme-com:gregorian».

> *Приведенная выше таблица стилей использует две функции расширения:* «Date:new()» *и* «Date.toString()». *Она была написана с применением элемента* <xsl:script>*, введенного в рабочий проект XSLT 1.1. Для ее работы с процессором XSLT 1.0 потребуется внести в нее незначительные изменения: подробнее см. в соответствующем приложении по продукту.*

Если бы параметр $today был равен «2000-13-18», вывод был бы следующим (независимо от исходного документа):

```
<date xmlns="urn:acme-com:gregorian"
      xmlns:Date="java:java.util.Date">2000-13-18</date>
```

Первое объявление пространства имен необходимо, так как оно определяет пространство имен для названия элемента <date>. Однако объявление xmlns:Date здесь можно и не использовать. Оно не наносит вреда, но и не дает никакой пользы. Оно есть здесь, потому что XSLT-процессор не может знать, что оно ненужно. Для того чтобы это объявление было опущено, можно использовать атрибут xsl:exclude-result-prefixes следующим образом:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.1"
   xmlns:Date="java:java.util.Date"
>

<xsl:script implements-prefix="Date" language="java"
    src="java:java.util.Date"/>

<xsl:template match="/"
      xmlns = "urn:acme-com:gregorian">
```

```
        <date xsl:exclude-result-prefixes="Date">
           <xsl:value-of select="$today"/>
        </date>
    </xsl:template>

    <xsl:param name="today" select="Date:toString(Date:new())"/>

    </xsl:stylesheet>
```

Факт, что элемент в конечном дереве имеет узел пространства имен, необязательно подразумевает, что когда конечное дерево будет записано как XML-документ, соответствующий элемент будет иметь объявление пространства имен для данного пространства имен. XSLT-процессор, с большой долей вероятности, опустит объявление пространства имен, если оно избыточно, другими словами, если оно дублирует объявление пространства имен для содержащего его элемента. Однако оно не может быть опущено просто по причине неиспользования из-за того, что объявления пространств имен могут воздействовать на данные в выходном документе неизвестным для XSLT-процессора способом. Приложения имеют полное право использовать объявления пространств имен для определения областей действия идентификаторов и имен, имеющихся в значениях атрибутов или в тексте.

Элемент в конечном дереве, сгенерированный из конечного литерального элемента, может иметь также дополнительные узлы пространств имен, кроме описанных выше. Например, он может унаследовать узлы пространств имен, которые были сгенерированы для содержащего его элемента. Кроме того, поскольку XSLT-процессор должен генерировать вывод, который соответствует рекомендации по пространствам имен XML, сгенерированный элемент будет также иметь узлы пространств имен, соответствующие любым префиксам пространств имен, использованным в имени элемента или в имени любого атрибута, даже если они были объявлены исключенными пространствами имен.

> *Атрибут* `xsl:exclude-result-prefixes` *употребляется для удаления объявлений пространств имен, которые не используются и не нужны. С его помощью нельзя удалить объявление пространства имен префиксов, которые используются в конечном дереве. Кроме того, его не применяют для удаления дубликатов объявлений пространств имен, поскольку большинство процессоров делают это автоматически.*

Префиксы пространств имен

Когда конечный литеральный элемент копируется в конечное дерево, расширенные имена (то есть локальное имя и URI пространства имен) новых узлов элемента и атрибутов в конечном дереве будут такими же, как у соответствующих узлов в таблице стилей. Обычно имена, которые выводятся в итоге, также используют тот же самый префикс пространства имен.

Случаются нетипичные ситуации, когда XSLT-процессор должен изменить префикс пространства имен. Например, можно создать два атрибута, которые используют одинаковый префикс пространства имен, относящийся к разным URI пространств имен, как в следующем примере:

```
<output>
   <xsl:attribute name="out:file"
      xmlns:out="http://domain-a.com/">a</xsl:attribute>
   <xsl:attribute name="out:dir"
      xmlns:out="http://domain-b.com/">b</xsl:attribute>
</output>
```

Генерируемый вывод в этом случае будет иметь такой вид:

```
<output
   out:file="a"
   ns1:dir="b"
   xmlns:out="http://domain-a.com/"
   xmlns:ns1="http://domain-b.com/"/>
```

У XSLT-процессора нет другого выбора, кроме как изобрести префикс для одного из пространств имен, потому что предложенный префикс уже используется с другим значением. Однако на выходном файле это никак не отразится, поскольку префиксы пространств имен, по существу, произвольны (только URI имеет какое-то реальное значение).

> *Когда узлы пространств имен копируются из исходного документа или дерева таблицы стилей в конечное дерево, префикс пространства имен и URI пространства имен копируются без изменений. Когда копируются узлы элемента или атрибута, расширенное имя элемента или атрибута (то есть его локальное имя и URI пространства имен) всегда сохраняется, но префикс пространства имен иногда может быть изменен. Однако если это случается, в конечное дерево добавляется дополнительный узел пространства имен, чтобы сопоставить новый префикс пространства имен с правильным URI пространства имен.*

Использование псевдонимов для пространств имен

При копировании конечного литерального элемента в конечное дерево в некоторых ситуациях вместо изменения префикса пространства имен необходимо изменить URI пространства имен.

Наиболее очевидная ситуация, в которой это требуется, – когда выходной документ сам является таблицей стилей. Это требование не столь экзотично, как может показаться: создание таблицы стилей подобным образом – очень удобный прием. Например, если компания изменяет фирменный оформительский стиль (меняет шрифты и цвета), можно написать XSLT-преобразование, чтобы привести все имеющиеся таблицы стилей к новому стандарту.

При генерировании таблицы стилей может понадобиться генерировать в конечном дереве XSLT-элементы типа `<xsl:template>`; но такие элементы нельзя включать как конечные литеральные элементы таблицы стилей, потому что они были бы спутаны с инструкциями. Тогда приемлемым решением является включение их в таблицу стилей с другим пространством имен и объявление в элементе `<xsl:namespace-alias>`, что при копировании конечного литерального элемента в конечное дерево URI должен быть изменен.

Более подробно см. `<xsl:namespace-alias>` в главе 4.

Шаблоны значений атрибутов

Как уже было показано, шаблон значения атрибута – это особый вид параметризованного значения атрибута. Его можно использовать двумя способами:

- Для конечного литерального элемента шаблон значения атрибута обеспечивает способ генерирования атрибута, значение которого вычисляется во время выполнения, а не берется всегда одно и то же, например `<TD WIDTH="{$width}">`. Того же эффекта можно достичь и с помощью инструкции `<xsl:attribute>`, но шаблоны значений атрибутов проще писать и они более понятны.
- Для ряда элементов XSLT некоторые атрибуты могут вычисляться во время выполнения. Например, при сортировке вместо записи `«order="ascending"»` или `«order="descending"»` можно написать `«order="{$order}"»`, чтобы порядок изменялся в зависимости от параметра времени выполнения. Заметьте, что эта возможность доступна для очень немногих атрибутов. Они перечисляются позже в этом разделе.

Термин **шаблон** здесь не имеет никакого отношения к шаблонным правилам XSLT, телам шаблонов или элементам `<xsl:template>`. Шаблоны значения атрибута – просто условное обозначение встраивания переменных компонентов в постоянное значение атрибута.

Шаблон значения атрибута – это строка, в которую могут быть встроены выражения XPath, заключенные в фигурные скобки («{» и «}»). Выражение XPath вычисляется, результат преобразуется в строку с помощью правил преобразования, описанных в разделе «Типы данных» главы 2, и полученная строка подставляется в значение атрибута на место выражения.

Предположим, например, что имеется набор изображений, представляющих буквы алфавита, как показано ниже, и нужно использовать их, чтобы отобразить первую букву в абзаце.

***Рис. 3.7.** Буквы используемого шрифта*

Для этого можно написать шаблонное правило следующим образом (игнорируя практические детали о том, как поступать с абзацами, которые не начинаются с заглавной буквы). Данное правило использует функцию `substring()`, которая описана в главе 7.

```
<xsl:template match="para">
   <p><img src="fancy{substring(.,1,1)}.gif"/>
   <xsl:value-of select="substring(.,2)" /></p>
</xsl:template>
```

Абзац, который начинается с буквы A (как текущий абзац), приведет к вычислению атрибута src элемента <img> как «img src="fancyA.gif"» и будет отображен в броузере следующим образом:

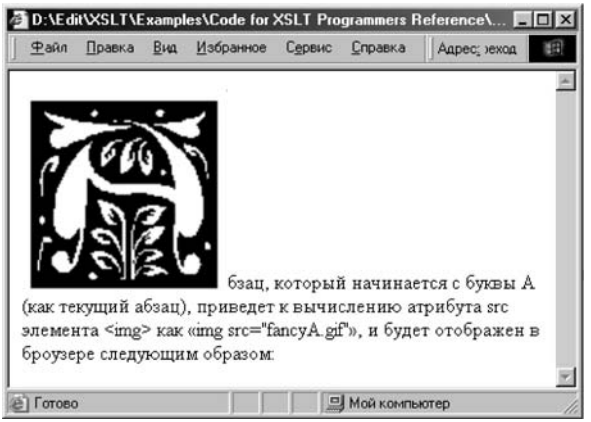


***Рис. 3.8.** Отображение полученного вывода в окне броузера*

Если нужно включить в значение атрибута символы «{» или «}» в их обычном значении (иногда это требуется при создании динамических HTML-страниц), то их нужно удваивать: «{{» или «}}». Однако это следует делать только в атрибуте, который интерпретируется как шаблон значения атрибута. В других атрибутах фигурные скобки не имеют специального значения.

Фигурные скобки никогда не могут быть вложенными. Их можно использовать только для включения выражения XPath в текст атрибута таблицы стилей; они не могут использоваться непосредственно в выражениях XPath. Всегда можно достичь требуемого эффекта другим способом; например, вместо:

```
<a href="#{id( 'A{@nr}' )}">  <!-- НЕВЕРНО -->
```

можно написать:

```
<a href="#{id( concat('A', @nr) )}">
```

Функция concat(), описанная в главе 7, выполняет конкатенацию строк.

Шаблоны значений атрибутов не могут использоваться где угодно в таблице стилей. Их можно применять лишь для тех атрибутов, которые определены в рекомендации XSLT как шаблоны значений атрибутов. В следующей таблице дан полный список всех мест, где можно применять их:

Элемент	Атрибуты, интерпретируемые как шаблоны значений атрибутов
Конечные литеральные элементы	Все атрибуты, кроме атрибутов из пространства имен XSLT
Элементы расширения	Как определено в спецификации каждого элемента расширения

Элемент	Атрибуты, интерпретируемые как шаблоны значений атрибутов
<xsl:attribute>	name, namespace
<xsl:document>	Все атрибуты (замечание: этот элемент введен в XSLT 1.1)
<xsl:element>	name, namespace
<xsl:number>	format, lang, letter-value, grouping-separator, grouping-size
<xsl:output>	Все атрибуты (в XSLT 1.1: в XSLT 1.0, ни один из атрибутов не интерпретируется как шаблон значения атрибута)
<xsl:processing-instruction>	name
<xsl:sort>	lang, data-type, order, case-order

Во всех других контекстах не стоит даже пытаться их использовать, так как фигурные скобки будут или игнорироваться, или вызовут сообщение об ошибке. Конечно, очень соблазнительно, например, воспользоваться элементом <xsl:call-template>, а имя вызываемого шаблона записать в переменную и написать:

```
<!-- НЕВЕРНО -->
<xsl:param name="tname"/>
<xsl:call-template name="{$tname}"/>
<!-- НЕВЕРНО -->
```

Однако так нельзя делать, потому что атрибута name (или любого другого атрибута <xsl:call-template>) элемента <xsl:call-template> нет в вышеупомянутом списке возможных применений шаблонов значений атрибутов.

Почему шаблоны значений атрибутов так строго ограничиваются? Некоторые ограничения кажутся взятыми с потолка, но большая их часть введена намеренно, чтобы упростить задачи XSLT-процессора:

- Шаблоны значений атрибутов обычно недопустимы в атрибутах элементов верхнего уровня. Это гарантирует, что их значения известны еще до прочтения исходного документа и не меняются между запусками таблицы стилей. Однако в рабочем проекте XSLT 1.1 это ограничение снято для элемента <xsl:output> и позволено определять все его атрибуты как шаблоны значений атрибутов.
- Шаблоны значений атрибутов недопустимы в атрибутах, значение которых – выражение XPath или образец (pattern). Это гарантирует, что выражения и образцы скомпилируются при прочтении таблицы стилей, и не нужно будет заново анализировать их при каждом их вычислении.
- Шаблоны значений атрибутов обычно недопустимы в атрибутах, значение которых – имя другого объекта таблицы стилей, например именованного шаблона или именованного набора атрибутов. Это гарантирует, что ссылки из одного объекта таблицы стилей на другой объект могут

быть разрешены раз и навсегда при первом прочтении таблицы стилей. Однако они допустимы в именах объектов, записываемых в конечное дерево.

- Шаблоны значений атрибутов недопустимы в атрибутах, интерпретируемых синтаксическим анализатором XML, в частности, для `xml:space`, `xml:lang` и объявлений пространств имен (`xmlns` и `xmlns:prefix`). Причина в том, что синтаксический анализатор XML читает значение раньше, чем XSLT-процессор получает возможность раскрыть его. Атрибут `xml:base` при использовании с конечным литеральным элементом ведет себя аномально: значением этого атрибута, как его видит синтаксический анализатор XML, является его значение до раскрытия шаблона значения атрибута, в то время как значением, копируемым в конечное дерево, является значение после раскрытия шаблона значения атрибута. Поэтому в случае атрибута `xml:base` лучше всего избегать фигурных скобок в его значении. (Атрибут `xml:base`, значение которого определено в отдельной рекомендации W3C под названием «XML Base», полностью поддерживается только в XSLT 1.1.)

Остальные ограничения можно считать взятыми с потолка. Например, трудно найти разумный довод, почему атрибут `terminate` элемента `<xsl:message>` не может быть шаблоном значения атрибута, тем не менее, спецификация не позволяет это.

Когда вычисляется выражение XPath внутри шаблона значения атрибута, его контекст такой же, как и в любом другом выражении в таблице стилей. Понятие контекста выражения сформулировано в соответствующем разделе главы 2: оно определяет значение конструкций типа «`.`», ссылающееся на контекстный узел, и «`position()`», возвращающее положение в контексте. Переменные и префиксы пространств имен могут использоваться в выражениях, только если их область действия распространяется на данное место таблицы стилей. Контекстный узел, положение в контексте и размер контекста определяются по текущему узлу и текущему списку узлов, обрабатываемых самым последним вызовом `<xsl:apply-templates>` или `<xsl:for-each>`; если еще не было такого вызова (что может случиться во время вычисления значения глобальной переменной или во время вычисления значений атрибутов элемента `<xsl:output>`), то текущий узел и текущий список узлов содержат только корневой узел.

Расширяемость

Наученный горьким опытом общения с «фирменными» расширениями HTML от компаний-разработчиков, комитет W3C, ответственный за XSLT, очень осторожно подошел к разрешению компаниям-разработчикам расширений – только строго контролируемым путем.

Механизмы расширяемости в XSLT регулируются несколькими негласными принципами проектирования:

- Для гарантии, что расширения от компании-разработчика не будут конфликтовать со стандартными средствами (включая средства, вводимые в будущих версиях) или с расширениями, вводимыми другой компанией-разработчиком, используются пространства имен.
- XSLT-процессор должен быть способен распознавать используемые расширения, включая расширения, введенные другой компанией-разработчиком, и когда он не может обрабатывать эти расширения, отказ должен происходить определенным образом.
- Автор таблицы стилей должен быть способен выяснить, доступны ли специфические расширения, и определить действия при их отсутствии. Например, таблица стилей могла бы обеспечить достижение того же эффекта другим способом, или она могла бы обойтись без какого-то специального эффекта в выводе.

Основными механизмами расширений, описываемыми ниже, являются функции расширения и элементы расширения. Однако компании-разработчики могут также вводить другие виды расширений или обеспечивать механизмы их введения пользователями или третьими лицами. Это могут быть следующие расширения:

- Элементы, определенные в XSLT, могут иметь дополнительные, определяемые компанией-разработчиком атрибуты, при условии, что они используют непустой URI пространства имен и что они не изменяют поведения стандартных элементов и атрибутов. Например, компания-разработчик может добавить к элементу `<xsl:template>` атрибут типа `acme:debug`, задача которого – приостановить выполнение, когда шаблон подвергается обработке. Но добавление атрибута `acme:repeat="2"`, задача которого – выполнить шаблон дважды, противоречило бы правилам соответствия спецификации. (Конечно, принимают ли компании-разработчики во внимание правила соответствия спецификации – это отдельный вопрос.)
- Компании-разработчики могут определять дополнительные элементы верхнего уровня; снова при условии, что они используют непустой URI пространства имен и что они не изменяют поведения стандартных элементов и атрибутов. Пример такого элемента – элемент `<msxsl:script>` от Microsoft для определения внешних функций VBScript или JScript (это предшественник стандартного элемента `<xsl:script>` в XSLT 1.1). Любой процессор, который не распознает данный URI пространства имен, игнорирует такие элементы верхнего уровня.
- Некоторые определенные в XSLT атрибуты имеют допускающий изменения набор значений, где компании-разработчики свободны в выборе поддерживаемых значений. Примерами являются атрибут `lang` элементов `<xsl:number>` и `<xsl:sort>`, который обеспечивает зависимую от языка нумерацию и сортировку, атрибут `method` элемента `<xsl:output>`, который определяет, как конечное дерево выводится в файл, и атрибут `format` элемента `<xsl:number>`, который позволяет компании-разработчику обеспечивать дополнительные способы нумерации, кроме тех, что определены в стандарте. Список свойств системы, указываемый в первом параметре

функции system-property(), тоже имеет допускающий изменения набор значений, так же как и набор названий языков, разрешенных в атрибуте language элемента <xsl:script> XSLT 1.1.

Функции расширения

Функции расширения обеспечивают механизм для расширения возможностей XSLT путем обращения к другим языкам, таким как Java или JavaScript. Чаще всего это нужно для того, чтобы:

- Улучшить производительность (например, при выполнении сложных строковых манипуляций)
- Полнее использовать системные возможности и сервисы
- Использовать готовый код на другом языке
- Работать было удобнее, поскольку сложные алгоритмы и вычисления могут быть весьма громоздкими, когда они пишутся на языке XSLT

Термин **функция расширения** используется как для функций, предлагаемых компаниями-разработчиками в дополнение к основным функциям, определенным в стандартах XSLT и XPath (описанным в главе 7), так и для функций, написанных пользователями или третьими лицами.

Рекомендация XSLT 1.0 позволяет вызывать функции расширения, но не определяет, как они должны быть написаны, как должны быть связаны с таблицей стилей или какие языки должны поддерживаться. Это означает, что для XSLT 1.0 довольно трудно написать функцию расширения, которая работала бы с XSLT-процессорами разных компаний-разработчиков, даже несмотря на то, что существует ряд соглашений, принятых несколькими компаниями-разработчиками.

В рабочем проекте XSLT 1.1 эта ситуация значительно улучшена. Рабочий проект XSLT 1.1 определяет стандартный интерфейс для вызова функций расширения, написанных на Java или JavaScript (формально ECMAScript). Он не требует, чтобы процессоры поддерживали оба эти языка, и позволяет им поддерживать, если нужно, дополнительные языки, но функции расширения для Java теперь должны быть переносимы между всеми XSLT-процессорами, которые поддерживают интерфейс Java, а функции расширения для JavaScript должны быть совместимыми для тех, которые поддерживают интерфейс JavaScript. Существует несколько незначительных ограничений этой совместимости в плане соответствия моделей XPath и DOM. Привязки языков и эти ограничения описаны в главе 8.

Именем функции в синтаксисе выражений XPath (см. главу 5) является полное имя, то есть имя с необязательным префиксом пространства имен. Если префикса нет, то это должно быть одно из имен основных функций, определенных в стандарте: например основная функция not() может быть вызвана как:

```
<xsl:if test="not( @name = 'Моцарт' )">
```

Если префикс есть, функция считается функцией расширения. URI пространства имен, соответствующий префиксу, используется для поиска в таблице стилей элемента <xsl:script>, который, в свою очередь, определяет, где может быть найдена реализация функции. Например, следующая таблица стилей объявляет глобальную переменную и задает ее значение как случайное число между нулем и единицей, используя подходящий метод класса java.lang.Math из стандартной библиотеки Java:

```
<xsl:stylesheet version="1.1"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:Math="java:java.lang.Math">
<xsl:script implements-prefix="Math" language="java" src="java:java.lang.Math"/>
<xsl:variable name="random-number" select="Math:random()"/>
```

Когда вызывается функция, загружается соответствующий класс Java и ищется (используя Java-механизмы интроспекции) метод, имя и параметры которого соответствуют имени и параметрам, указанным в вызове XSLT, после чего вызывается требуемый метод.

До введения элемента <xsl:script> в XSLT 1.1 несколько процессоров XSLT 1.0 обеспечивали механизм, в котором URI пространства имен заканчивается полностью заданным именем Java-класса. Этот механизм предполагает, что вся информация, необходимая для идентификации и вызова функции, содержится непосредственно в URI пространства имен. Вышеупомянутый пример может быть переписан для процессора xt следующим образом:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:variable name="random-number" select="Math:random()"
    xmlns:Math="http://www.jclark.com/xt/java/java.util.Date"/>
```

Подобные механизмы обеспечивают и другие процессоры, включая Saxon и Xalan, но в каждом случае правила для формирования URI пространства имен слегка различны, поэтому лучше использовать элемент <xsl:script>, если он поддерживается.

Можно передать результат одной внешней функции в другую, даже если это объект, который не может быть прямо представлен как один из типов данных XPath. Его называют **внешним объектом**.

Как показывает следующий пример, это позволяет выводить текущую дату и время, используя стандартный Java-класс «java.util.Date». Заданный по умолчанию конструктор для этого класса создает объект Date, значение которого – текущая дата, а метод «toString()», вызванный для данного объекта, возвращает строковое представление даты. В таблице стилей можно записать так:

```
<xsl:stylesheet version="1.1"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:Date="java:java.util.Date">

<xsl:script implements-prefix="Date" language="java" src="java:java.util.Date"/>
```

```
<xsl:variable name="today" select="Date:new()"/>

<xsl:template name="show-todays-date">
    <xsl:value-of select="Date:toString($today)"/>
</xsl:template>
. . .
```

В выражении «Date:new()» слово Date является префиксом пространства имен, соответствующим URI пространства имен, которое относится к классу «java.util.Date», поэтому процессор знает, что его задача – создать экземпляр этого Java-класса. В результате переменная «$today» будет содержать внешний объект, который является Java-объектом типа «java.util.Date». Для выведения даты с использованием <xsl:value-of> нужно преобразовать этот объект в строку, поэтому нужно передать его в метод «toString()» этого же класса (потому что он тоже имеет префикс пространства имен «Date»). По правилам целевой объект метода записывается как его первый параметр; таким образом, запись «Date:toString($today)» в выражении XPath эквивалентна записи «today.toString()» в Java. Результат ее действия в том, что названный шаблон записывает текущую дату в конечное дерево.

Рабочий проект XSLT 1.1 обеспечивает также стандартный интерфейс для функций JavaScript: вместо загрузки из отдельного файла, содержащего скомпилированный код, код на JavaScript можно встраивать непосредственно в элемент <xsl:script>. Процессор MSXML3 от Microsoft является (на момент написания книги) реализацией XSLT 1.0, но он обеспечивает собственный, очень похожий механизм, используя элемент <msxsl:script> вместо <xsl:script>.

Ниже приведен пример простой функции расширения, реализованной с использованием этого механизма XSLT-процессора Microsoft, и выражения, которое вызывает данную функцию:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:js="javascript:my-extensions">

<msxsl:script
   xmlns:msxsl="urn:schemas-microsoft-com:xslt"
   language="VBScript"
   implements-prefix="ms"
>
Function ToMillimetres(inches)
   ToMillimetres = inches * 25.4
End Function
</msxsl:script>

<xsl:template match="/" >
   <xsl:variable name="test" select="12"/>
   <size><xsl:value-of select="ms:ToMillimetres($test)"/></size>
</xsl:template>
</xsl:stylesheet>
```

С помощью функции `function-available()` можно проверить, доступна ли определенная функция расширения. Например:

```
<xsl:choose xmlns:acme="http://acme.co.jp/xslt">
   <xsl:when test="function-available('acme:moonshine')">
      <xsl:value-of select="acme:moonshine($x)"/>
   </xsl:when>
   <xsl:otherwise>
      <xsl:text>*** Извините, функция moonshine сегодня недоступна ***</xsl:text>
   </xsl:otherwise>
</xsl:choose>
```

В спецификации говорится, что XSLT-процессор, который не распознает функцию расширения `acme:moonshine()`, не должен аварийно завершать работу только потому, что таблица стилей ссылается на нее: он должен это сделать, только если эта функция вызывается. В данном случае аварийное завершение не должно случиться, потому что сначала проверяется доступность функции при помощи функции `function-available()`, возвращающей ложь, если функция недоступна.

Функции расширения могут иметь побочные эффекты, потому что они пишутся на универсальных языках программирования. Например, они могут производить изменения в базе данных, могут просить пользователя ввести значение или могут использовать счетчики. В процессор Xalan входит пример приложения, которое реализует счетчик с помощью функций расширения, эффективно обходя запрещение изменения значений переменных XSLT. Однако функции расширения с побочными эффектами следует применять с большой осторожностью, потому что в спецификации XSLT не говорится, в каком порядке могут происходить вещи. Например, там не говорится, когда вычисляется переменная: когда впервые встречается ее объявление или когда впервые используется ее значение. В частности, в одной из программ, `xt`, выбрана стратегия отложенных вычислений: переменные никогда не вычисляются, пока не потребуется. Если для вычисления таких переменных используются функции расширения с побочными эффектами, результаты могут быть очень удивительными, потому что порядок, в котором вызываются функции расширения, становится весьма непредсказуемым. Например, если одна функция записывает в регистрационный журнал, а другая в это время закрывает его, может случиться, что журнал будет закрыт раньше произведения записи.

О создании функций расширения подробнее говорится в главе 8.

Элементы расширения

Элемент расширения – это элемент, находящийся внутри тела шаблона, который принадлежит пространству имен, обозначенному как пространство имен расширений. Пространство имен обозначается как пространство имен расширений путем включения его префикса пространства имен в атрибут `extension-element-prefixes` элемента `<xsl:stylesheet>` или в атрибут `xsl:exten-`

sion-element-prefixes непосредственно самого элемента или содержащего его элемента расширения, или конечного литерального элемента.

Например, Saxon предоставляет элемент расширения <saxon:while> для выполнения циклов, пока условие остается истинным. Не существует аналогичной стандартной конструкции XSLT, потому что без побочных эффектов истинное условие никогда не может стать ложным. Но при использовании вместе с функциями расширения <saxon:while> может быть полезным дополнением.

Следующий именованный шаблон принимает строку в качестве параметра и разбивает ее на слова, выводя каждое слово как отдельный элемент. В качестве функции расширения он использует стандартный Java-класс java.util.StringTokenizer:

```
<xsl:stylesheet version="1.1"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:Tokenizer="java:java.util.StringTokenizer">

<xsl:script implements-prefix="Tokenizer"
   language="java" src="java:java.util.StringTokenizer"/>

<xsl:template name="tokenize">
   <xsl:param name="sentence"/>
   <xsl:variable name="tok" select="Tokenizer:new($sentence)"/>
   <saxon:while test="Tokenizer:hasMoreTokens($tok)"
          xsl:extension-element-prefixes="saxon"
          xsl:exclude-result-prefixes="Tokenizer"
          xmlns:saxon="http://icl.com/saxon">
       <слово>
         <xsl:value-of select="Tokenizer:nextToken($tok)"/>
       </слово>
   </saxon:while>
</xsl:template>
</xsl:stylesheet>
```

Заметьте, для того, чтобы это работало, слово «saxon» должно быть объявлено как префикс элемента расширения, иначе элемент <saxon:while> интерпретировался бы как конечный литеральный элемент и копировался бы в вывод. Атрибут xsl:exclude-result-prefixes не является строго необходимым, но он предотвращает загромождение вывода ненужными объявлениями пространств имен.

Если этот шаблон вызывается со значением параметра «Кошка сидела на циновке» следующим образом:

```
<xsl:template match="/">
  <xsl:call-template name="tokenize">
    <xsl:with-param name="sentence">Кошка сидела на циновке</xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

это приведет к следующему выводу:

```
<слово>Кошка</слово><слово>сидела</слово><слово>на</слово><слово>циновке</слово>
```

Если в элементе `<xsl:stylesheet>` *объявляются дополнительные пространства имен, они будут копироваться в элементы* `<слово>`. *Чтобы избавиться от них, нужно добавить префикс нежелательного пространства имен к атрибуту* `xsl:exclude-result-prefixes`.

Как и в случае функций расширения, термин **элемент расширения** охватывает нестандартные элементы, предоставляемые как компанией-разработчиком, так и пользователями или третьей стороной. Нет специального требования, что реализация XSLT должна давать пользователям возможность определять новые элементы расширения; говорится только, что она должна реагировать особым образом, когда сталкивается с элементами расширения, которые не может обработать.

В тех случаях, когда программный пакет позволяет пользователям определять элементы расширения (например, такая возможность есть у пользователей Saxon и Xalan), используемые механизмы и API, вероятно, сложнее, чем для функций расширения, и использование их – непростая задача. Однако элементы расширения могут предоставить возможности, которые было бы очень трудно обеспечить одними только функциями расширения.

Если в таблице стилей есть элемент расширения, то все XSLT-процессоры опознают его как таковой, но в общем случае некоторые смогут обрабатывать его, а некоторые – нет (потому что он определен другой компанией-разработчиком). Как и с функциями расширения, процессор не должен аварийно завершаться только потому, что присутствует элемент расширения, завершение разрешается, если только делается попытка подвергнуть его обработке.

Существует два механизма, позволяющих авторам таблиц стилей проверять, доступен ли определенный элемент расширения: функция `element-available()` и инструкция `<xsl:fallback>`.

Механизм действия функции `element-available()` очень похож на механизм функции `function-available()`. Например:

```
<xsl:choose xmlns:acme="http://acme.co.jp/xslt">
   <xsl:when test="element-available('acme:moonshine')">
     <acme:moonshine select="$x" xsl:extension-element-prefixes="acme"/>
   </xsl:when>
   <xsl:otherwise>
      <xsl:text>*** Извините, элемент moonshine сегодня недоступен ***</xsl:text>
   </xsl:otherwise>
</xsl:choose>
```

Заметьте, что во время вызова функции `element-available()` префикс для элемента расширения (в данном примере «`acme`») должен быть объявлен в объявлении пространства имен, но не требуется, чтобы он обозначал элементы расширения.

Инструкция `<xsl:fallback>` (которая подробно описана в главе 4) обеспечивает альтернативный способ, позволяющий указать, что должно произойти, если элемент расширения недоступен. Следующий пример эквивалентен предыдущему:

```
<acme:moonshine select="$x"
   xmlns:acme="http://acme.co.jp/xslt" xsl:extension-element-prefixes="acme">
   <xsl:fallback>
      <xsl:text>*** Извините, элемент moonshine сегодня недоступен ***</xsl:text>
   </xsl:fallback>
</acme:moonshine>
```

Когда подвергается обработке элемент расширения, а XSLT-процессор не знает, что с ним делать, он должен подвергнуть обработке любой дочерний элемент <xsl:fallback>. Если есть несколько дочерних элементов <xsl:fallback>, он должен применить каждый из них. Только в том случае, когда нет ни одного элемента <xsl:fallback>, он должен сообщить об ошибке. Наоборот, если XSLT-процессор способен обработать элемент, он должен игнорировать любой дочерний элемент <xsl:fallback>.

> *В спецификации не говорится фактически, что элемент расширения должен допускать наличие дочернего элемента* <xsl:fallback>. *Есть множество инструкций XSLT, которые не допускают* <xsl:fallback> *в качестве потомка, например* <xsl:copy-of> *и* <xsl:value-of>. *Однако элемент расширения, который не допускает* <xsl:fallback>, *определенно противоречил бы стандарту.*

Определяемые компанией-разработчиком или пользователем элементы верхнего уровня таблицы стилей формально не являются элементами расширения, потому что они не находятся в теле шаблона; поэтому пространство имен, в котором они находятся, не должно обозначаться как пространство имен расширений.

Совместимость с последующими версиями

В то время как расширяемость, как показано в предыдущем разделе, касается способа написания таблиц стилей, которые гибки по отношению к расширениям для языка XSLT, введенным компанией-разработчиком или пользователем, совместимость с последующими версиями заключается в достижении гибкости по отношению к различиям между версиями стандарта XSLT. Эти два понятия, безусловно, тесно связаны.

В настоящее время действуют две версии рекомендации по XSLT: версия 1.0 и рабочий проект для версии 1.1 (не считая шести рабочих проектов, которые предшествовали версии 1.0). Так что совместимость между версиями скоро станет проблемой. Однако проектировщики языка предвидели эти трудности, и даже версия 1.0 позволяет писать переносимые таблицы стилей.

В таблице стилей должен указываться номер версии (обычно «version=1.0» или «version=1.1») как атрибут элемента <xsl:stylesheet>. Атрибут «version=1.0» подразумевает, что таблица стилей использует только средства, определенные в XSLT версии 1.0, а определение «version=1.1» указывает, что могут потребоваться средства, определенные в версии 1.1.

Если используются только средства, определенные в XSLT версии 1.0, то следует указать «version=1.0», и тогда каждый совместимый XSLT-процес-

сор правильно обработает таблицу стилей. Эта стратегия поддерживается во всех примерах этой книги.

При использовании возможностей, которые впервые введены лишь в версии 1.1, нужно указать «version=1.1». В этом случае процессор, соответствующий XSLT 1.1, правильно обработает таблицу стилей, как определено в спецификации XSLT 1.1, но процессор XSLT 1.0 обработает ее в **режиме совместимости с последующими версиями**. Аналогично процессор XSLT 1.1 переключится в режим совместимости с последующими версиями, если в таблице стилей указана «version=2.0» или любая другая версия, которую процессор не опознает.

В режиме опережающей совместимости XSLT-процессор должен учитывать, что таблица стилей использует средства XSLT, определенные в версии стандарта, который был опубликован уже после выпуска программы. Процессор, конечно, не будет знать, что делать с этими средствами, но он должен учитывать, что автор таблицы стилей использует их намеренно. Процессор обрабатывает их таким же образом, как расширения от компании-разработчика, которые он не понимает:

- Он должен сообщать об ошибке для XSLT-элементов, которых он не понимает, только если их необходимо подвергнуть обработке и если нет ни одной дочерней инструкции <xsl:fallback>.
- Он должен игнорировать атрибуты, которые он не опознает, и нераспознанные значения для известных атрибутов.
- Он должен сообщать об ошибке для функций, которые он не опознает или которые содержат неверное число аргументов, только в том случае, когда функция вызывается фактически. Можно избежать этой сбойной ситуации, используя функцию function-available() для проверки существования функции перед ее вызовом.
- Он должен сообщать о синтаксических ошибках в выражениях XPath, которые используют синтаксис, не разрешенный в XPath версии 1.0, только в том случае, когда выражение вычисляется фактически. (Обе версии – XSLT 1.0 и XSLT 1.1 – используют синтаксис XPath 1.0; то есть синтаксис XPath для них одинаков.)

Такое поведение поддерживается, только когда включен режим совместимости с последующими версиями, а это случается, только если элемент <xsl:stylesheet> определяет версию, отличную от «1.0» или (только для XSLT 1.1) «1.1». Режим совместимости с последующими версиями можно указать также для части таблицы стилей, если определить атрибут xsl:version для любого конечного литерального элемента, опять же со значением, отличным от «1.0» или «1.1». Если режим совместимости с последующими версиями не включен, то есть если версия определена как «1.0» или «1.1», то любое использование элемента, атрибута или функции, которые не определены в стандарте XSLT версии 1.1, или XPath синтаксиса, который не определен в стандарте XPath 1.0, является ошибкой, и должно сообщаться, выполняется это фактически или нет.

Если указать «version=1.0», а затем использовать средства XSLT 1.1 типа <xsl:document>, то процессор XSLT 1.0 отклонит это как ошибку. В то же время, процессор XSLT 1.1 успешно обработает эту таблицу стилей. XSLT-процессоры не обязаны знать, какие средства вводились в той или иной версии стандарта.

Обработка в режиме совместимости с последующими версиями введена для того, чтобы позволить создавать таблицу стилей, которая использует средства версии 1.1, но, тем не менее, вполне хорошо может выполняться с XSLT-процессором, поддерживающим только версию 1.0, или с какого-то момента в будущем – использует средства версии 2.0, но, тем не менее, вполне хорошо может выполняться с процессором XSLT 1.1. Для достижения этого можно использовать те же самые механизмы, которые используются для обработки расширений от компании-разработчика: функции function-available() и element-available() и элемент <xsl:fallback>.

Существует еще один механизм, который тоже можно использовать: это функция system-property() (описанная в главе 7) для выяснения, какую версию XSLT реализует процессор или какой процессор используется. Например, можно записать такой код:

```
<xsl:if test="system-property('xsl:version')=2.0 or
      starts-with(system-property('xsl:vendor', 'xalan'))">
   <xsl:новая-возможность/>
</xsl:if>
```

Полагаться на номер версии, получаемый таким образом, не очень надежно: все еще существует множество процессоров, которые возвращают «1.0» как значение «system-property('xsl:version')», но которые не реализуют стандарт XSLT полностью, и такое положение, конечно, сохранится. Однако проверка, какой именно процессор используется, удобна для переносимости, особенно, когда компании-разработчики не строго придерживаются правил соответствия.

Формально процессор, который реализует некоторые, но не все новшества XSLT 1.1, не соответствует ни XSLT 1.0, ни XSLT 1.1. Но поскольку многие XSLT-процессоры разрабатываются поступательно, с выходом новых версий каждые несколько недель, вполне можно попасть как раз на такую программу, занимающую «промежуточное положение». Возможно, программа будет возвращать «1.1» как значение «system-property('xsl-version')», если компания-разработчик уверена, что его продукт является «почти» совместимым: опыт показывает, что каждая компания-разработчик понимает это по-своему!

Пробельные символы

Обработка пробельных символов может быть довольно запутанной. Когда вывод – HTML, на этот счет можно не слишком волноваться, потому что, кроме некоторых очень специфичных контекстов, HTML обычно считает любую последовательность пробелов и переходов на новую строку одним

пробелом. Но для других выходных форматов получение пробелов и новых строк в нужных местах и избежание их там, где они не нужны, может быть существенным.

Налицо две проблемы:

- Управление тем, какие из пробельных символов в исходном документе являются существенными, и поэтому должны быть видимы для таблицы стилей.
- Управление тем, какие из пробельных символов в таблице стилей являются значащими, потому что значащие пробельные символы таблицы стилей, вероятно, будут копироваться в вывод.

Пробельным символом является любая последовательность из следующих четырех символов:

Символ	Символ Unicode
табуляция	`#x9`
переход на новую строку	`#xA`
возврат каретки	`#xD`
пробел	`#x20`

Определение их в XSLT точно такое же, как и непосредственно в XML. Другие символы, такие как неразрывный пробел (`#xA0`), который знаком авторам HTML как ссылка на сущность «` `», возможно, вызывают такой же расход черных чернил, как эти четыре, но они не включены в определение.

В определении есть и некоторые другие недоразумения. Ссылка на символ «`&#x20;`» во многих случаях равносильна нажатию клавиши «пробел» на клавиатуре, но в некоторых обстоятельствах это не так. Ссылка на символ «`&#x20;`» будет обработана как пробельный символ XSLT-процессором, но не синтаксическим анализатором XML, поэтому необходимо разбираться, какие правила применяются на той или иной стадии обработки.

В стандарте XML предприняты некоторые попытки различать значащие и незначащие пробельные символы. Пробельные символы в элементах с содержимым, представленным только элементами, считаются незначащими, тогда как пробельные символы в элементах, позволяющих #PCDATA-содержимое, являются значащими. Однако различие зависит от того, используется или нет проверяющий анализатор, и в любом случае стандарт требует уведомлять приложение об обоих типах пробельных символов. Создатели спецификации XSLT решили, что обработка пробельных символов не должна зависеть ни от DTD, ни от того, какой анализатор используется – проверяющий или нет. Вместо этого обработка пробельных символов контролируется полностью из исходного документа (с помощью атрибута `xml:space`) или из таблицы стилей (с помощью инструкций `<xsl:strip-space>` и `<xsl:preserve-space>`), которые подробно описаны в главе 4.

Первые стадии обработки пробельных символов – задача анализатора XML, и она выполняется задолго до того, как XSLT-процессор может увидеть данные:

- Концы строк, встречающиеся в текстовом содержимом элемента, всегда нормализуются к единственному символу новой строки «
». Это ликвидирует различия между окончаниями строк в системах Unix, Windows и Macintosh.
- Анализатор XML нормализует значения атрибутов. Символы табуляции или конца строки всегда замещаются одним пробелом, если они не записаны как ссылки на символ типа «	» или «&#A;»; для некоторых типов атрибутов (любых, кроме типа CDATA) анализатор XML удалит также начальные и концевые пробельные символы и нормализует другие последовательности пробельных символов к одному пробелу.

Такая нормализация атрибутов может быть существенной, если рассматриваемый атрибут – выражение XPath в таблице стилей. Например, предположим, что нужно проверить, содержит ли строковое значение символ конца строки. Это можно записать так:

```
<xsl:if test="contains(address, '&#xA')">
```

Важно использовать здесь ссылку на символ «
», а не реальный символ конца строки, потому что символ конца строки был бы преобразован синтаксическим анализатором XML в пробел, а выражение тогда фактически проверяло бы, содержит ли эта строка пробел.

Практически это означает, что если нужен конкретный пробельный символ, то нужно записывать его как ссылку на символ; если же пробельные символы используются только как разделители и заполнители, тогда замена на ссылки не нужна.

> ***Спецификация XSLT предполагает, что анализатор XML передаст все текстовые узлы, состоящие из пробельных символов, XSLT-процессору. Однако исходными данными для XSLT-процессора служит дерево, а спецификация XSLT не накладывает ограничений на его формирование. При использовании процессора MSXML3 от Microsoft ему предоставляется дерево в форме DOM, а по умолчанию при формировании модели DOM в MSXML3 текстовые узлы, состоящие из пробельных символов, удаляются. Если нужно, чтобы анализатор вел себя соответственно требованиям спецификации XSLT, следует перед загрузкой документа установить свойство*** `preserveWhitespace` ***объекта*** `Document` ***в значение*** `true`.

Как только анализатор XML выполнит свою работу, XSLT-процессор производит некоторую собственную обработку. К этому времени ссылки на сущности и символы раскрываются, поэтому не остается никакой разницы между пробелом, записанным как пробел или как « »:

- Смежные текстовые узлы объединяются в один текстовый узел (**нормализуются**, по терминологии DOM).
- Затем, если текстовый узел состоит полностью из пробельных символов, они удаляются из дерева при условии, что содержащий их элемент пере-

числен в определении <xsl:strip-space> таблицы стилей. Детальные правила на самом деле сложнее и принимают во внимание также наличие атрибута xml:space в исходном документе; подробнее см. элемент <xsl:text> в главе 4.

В этом процессе никогда не удаляются пробельные символы, смежные с непробельными. Например, рассмотрим:

```
<статья>
   <заголовок>Abelard and Heloise</заголовок>
   <подзаголовок>Some notes towards a family tree</подзаголовок>
   <автор>Brenda M Cook</автор>
   <аннотация>
      The story of Abelard and Heloise is best recalled
nowadays from the stage drama of 1970 and it is perhaps inevitable that Diana Rigg
stripping off for Keith Mitchell should be the most enduring image of this historic
couple in some people's minds.
   </аннотация>
</статья>
```

В этом фрагменте пять текстовых узлов содержат только пробельные символы: по одному перед каждым из дочерних элементов <заголовок>, <подзаголовок>, <автор> и <аннотация> и еще один между закрывающими тегами элементов <аннотация> и <статья>. Анализатор XML передает пробельные символы этих узлов XSLT-процессору, и дальнейшее зависит уже от таблицы стилей – принимать их во внимание или нет. Обычно в такой ситуации эти пробельные символы не имеют особого значения и могут быть удалены из дерева указанием <xsl:strip-space elements="статья"/>.

Пробельный символ в пределах элемента <аннотация> не может быть удален такой операцией. Символы перехода в новую строку в начале и в конце аннотации, а также в конце каждой строки являются частью текста, передаваемого синтаксическим анализатором приложению, а в таблице стилей невозможно объявить их незначащими. Они всегда будут присутствовать в модели дерева исходного документа. Что можно сделать – при обработке этих узлов в исходном дереве вызвать функцию normalize-space(), которая удалит начальные и концевые пробельные символы и заменит все другие последовательности из одного или нескольких пробельных символов на один пробел. Функция normalize-space() описана в главе 7.

Чтобы подчеркнуть это, в XSLT очень четко различаются текстовые узлы, содержащие только пробельные символы, и узлы, содержащие еще что-то кроме пробельных символов. Текстовый узел, состоящий пробельных символов, может существовать только там, где между двумя частями разметки нет ничего, кроме пробельных символов.

Рассмотрим следующий документ в качестве еще одного примера.

```
<человек>
   <имя>Пруденс Флауэрс</имя>
   <работодатель>Банк Lloyds</работодатель>
```

```
    <место-работы>
      71 Lombard Street
      London, UK
      <почтовый-индекс>EC3P 3BS</почтовый-индекс>
    </место-работы>
</человек>
```

Где здесь узлы пробельных символов? Рассмотрим этот пример снова, сделав на этот раз пробельные символы видимыми:

```
<человек>↵
→<имя>Пруденс Флауэрс</имя>↵
→<работодатель>Банк Lloyds</работодатель>↵
→<место-работы>↵
→♦♦♦71 Lombard Street↵
→♦♦♦London, UK↵
→♦♦♦<почтовый-индекс>EC3P 3BS</почтовый-индекс>♦↵
→</место-работы>↵
</человек>
```

Символ перехода на новую строку и символ табуляции между тегами <человек> и <имя> не соседствуют ни с какими непробельными символами, поэтому они представляют собой узел, состоящий из пробельных символов. То же относится и к символам между тегами </имя> и <работодатель> и между </работодатель> и <место-работы>. Однако большинство пробельных символов внутри элемента <место-работы> находится в одном текстовом узле с непробельными символами, поэтому они не составляют отдельный узел, состоящий из пробельных символов. Чтобы стало еще понятнее, выделим символы в узлах пробельных символов белым шрифтом на черном фоне:

```
<человек>■
■→<имя>Пруденс Флауэрс</имя>■
■→<работодатель>Банк Lloyds</работодатель>■
■→<место-работы>↵
→♦♦♦71 Lombard Street↵
→♦♦♦London, UK↵
→♦♦♦<почтовый-индекс>EC3P 3BS</почтовый-индекс>♦↵
■→</место-работы>■
</человек>
```

Почему это так важно? Как уже было показано, элемент <xsl:strip-space> позволяет контролировать, что будет происходить с узлами, состоящими из пробельных символов (выделенными выше белым шрифтом на черном фоне), но он не позволяет делать что-то особое с пробельными символами (изображенными черным шрифтом на белом фоне), которые встречаются в обычных текстовых узлах.

Все узлы, состоящие из пробельных символов, в этом примере – непосредственные потомки элемента <человек>, поэтому они могут быть удалены с помощью инструкции:

```
<xsl:strip-space elements="человек"/>
```

Если не произвести удаление, узлы, состоящие из пробельных символов, в исходном дереве сохранятся.

Узлы, состоящие из пробельных символов, в таблице стилей

В самой таблице стилей удаляются все узлы, состоящие из пробельных символов, за исключением пробельных символов в пределах элемента <xsl:text>. Таким образом, при желании скопировать текстовый узел, состоящий из пробельных символов, из таблицы стилей в конечное дерево следует записать его внутри элемента <xsl:text>, как показано ниже:

```
<xsl:value-of select="строка-адреса[1]"/>
<xsl:text>&#xA;</xsl:text>
<xsl:value-of select="строка-адреса[2]"/>
```

Единственная причина использования здесь ссылки на символ «
», а не самого символа перехода на новую строку в том, что так понятнее для читателей; кроме того, снижается вероятность случайного добавления к этому символу дополнительного пробела или символа табуляции. Написание пробельного символа в виде ссылки на символ не предотвратит в XSLT его обработку как пробельного символа, так как ссылки на символы будут раскрыты анализатором XML еще до того, как они попадут в XSLT-процессор.

Роль удаления узлов, состоящих из пробельных символов

Удаление узлов, состоящих из пробельных символов, приводит к двум основным эффектам. Скажем, для элемента <человек> в примере выше оно приведет к следующим результатам:

- При использовании элемента <xsl:apply-templates/> для обработки всех потомков элемента <человек> там не оказывается узлов, состоящих из пробельных символов, поэтому они не выбираются. Если бы их оставили в исходном дереве, то по умолчанию они были бы скопированы в конечное дерево.
- При использовании элемента <xsl:number> или функции position() или count() для подсчета узлов там не оказывается узлов, состоящих из пробельных символов, поэтому они не учитываются. Если бы их оставили в исходном дереве, то элементы <человек>, <работодатель> и <место-работы> стали бы узлами 2, 4 и 6, а не 1, 2 и 3.

Бывают случаи, когда важно сохранить узлы, состоящие из пробельных символов. Рассмотрим следующее:

```
<para>
Под редакцией <name>Джеймса Кларка</name>♦<email>jjc@jclark.com</email>
</para>
```

Ромб здесь обозначает пробел, который должен быть сохранен, но поскольку он не соседствует ни с каким другим текстом, он подлежал бы удалению. На практике в элементах, имеющих смешанное содержимое (то есть в элементах, которые имеют в качестве потомков и элементы, и текстовые узлы), пробельные символы почти всегда значимы.

Если нужно удалить в исходном дереве все узлы, состоящие из пробельных символов, можно написать:

```
<xsl:strip-space elements="*"/>
```

Если нужно удалить все узлы, состоящие из пробельных символов, кроме узлов в пределах некоторых указанных элементов, можно написать:

```
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="para h1 h2 h3 h4"/>
```

Если для любого элемента в документе (в исходном документе или в таблице стилей) использован определенный в XML атрибут «`xml:space="preserve"`», он имеет приоритет над этими правилами: узлы, состоящие из пробельных символов, в том элементе и во всех его потомках будут оставлены в дереве, если действие данного атрибута не отменяется указанием «`xml:space="default"`» для какого-либо потомка. Это позволяет «поштучно» регулировать для элементов, сохранять или нет пробельные символы, тогда как `<xsl:strip-space>` регулирует это по типам элементов.

Решение проблем с пробельными символами

С пробельными символами в выводе связаны две типичные проблемы: их оказывается либо слишком много, либо слишком мало.

При генерировании HTML несколько лишних пробельных символов обычно не играют роли, хотя в некоторых случаях это может слегка деформировать размещение на странице. Однако в случае некоторых текстовых форматов (классический пример – значения, разделяемые запятыми) следует быть очень аккуратными, чтобы вывести пробельные символы в правильных местах.

Слишком много пробельных символов

Если в выводе оказывается слишком много пробельных символов, причиной могут быть три компонента процесса:

- Исходный документ
- Таблица стилей
- Отступы в выводе

Прежде всего, следует удостовериться, что для элемента `<xsl:output>` установлено «`indent="no"`», чтобы исключить последнюю из возможных причин.

Если вывод пробельных символов соседствует с текстом, то он, вероятно, исходит из того же места, откуда и сам текст.

- Если этот текст исходит из таблицы стилей, используйте элемент <xsl:text> для более точного контроля над выводом. Например, следующий код выводит запятую между пунктами списка, но он может также вводить после запятой новую строку, так как символ перехода на новую строку – часть того же текстового узла, что и запятая:

```
<xsl:for-each select="item">
   <xsl:value-of select="."/>,
</xsl:for-each>
```

Если нужна запятая, но не новая строка, измените это так, чтобы символ перехода на новую строку оказался в отдельном текстовом узле и поэтому подлежал удалению:

```
<xsl:for-each select="item">
   <xsl:value-of select="."/>,<xsl:text/>
</xsl:for-each>
```

- Если текст приходит из исходного документа, воспользуйтесь функцией normalize-space(), чтобы перед выводом текста вырезать из него начальные и концевые пробелы.

Если ненужные пробельные символы в выводе находятся между тегами, то их источниками, вероятно, являются узлы, состоящие из пробельных символов в исходном дереве, которые не были удалены, и для решения этой проблемы нужно добавить в таблицу стилей элемент <xsl:strip-space>.

Слишком мало пробельных символов

Если пробельные символы в выводе нужны, а их там не оказывается, воспользуйтесь элементом <xsl:text>, чтобы генерировать их в нужном месте. Например, следующий код выведет строки поэмы в HTML, начиная каждую строчку поэмы с новой строки:

```
<xsl:for-each select="line">
      <xsl:value-of select="."/><br/>
</xsl:for-each>
```

В окне броузера текст будет отображаться совершенно правильно, но просматривать этот HTML в текстовом редакторе будет трудно, потому что все сольется в одну строку. Было бы полезно после каждого элемента
 начинать новую строку – это можно сделать следующим образом:

```
<xsl:for-each select="line">
   <xsl:value-of select="."/><br/><xsl:text>&#xa;</xsl:text>
</xsl:for-each>
```

Этого же можно достичь и с помощью другого приема: воспользоваться тем, что символ неразрывного пробела (#xa0), хоть он и невидимый, не классифицируется как пробельный символ. Поэтому можно написать:

```
<xsl:for-each select="line">
```

```
    <xsl:value-of select=".¨/><br/> 
</xsl:for-each>
```

Это работает, потому что символ новой строки после « » является теперь частью узла, состоящего не только из пробельных символов.

Резюме

Целью этой главы было изучить полное строение таблицы стилей перед переходом в следующей главе к детальным спецификациям каждого элемента.

- Сначала объяснялось, как можно сформировать программу таблицы стилей из одного или нескольких готовых модулей таблиц стилей, соединенных элементами <xsl:import> и <xsl:include>. Было показано, как концепция преимущества импортирования позволяет одной таблице стилей отменить определения другой, в которую она импортируется.
- Рассказано об элементе <xsl:stylesheet> (или <xsl:transform>), который является самым внешним элементом большинства модулей таблиц стилей.
- Описана инструкция обработки <?xml-stylesheet?>, которую можно использовать для связи исходного документа с его таблицами стилей, и объяснялось, как с помощью этого создать таблицу стилей, вложенную прямо в исходный документ, стиль которого она определяет.
- Дан обзор элементов верхнего уровня, встречающихся в таблице стилей, то есть непосредственных потомков элемента <xsl:stylesheet> или <xsl:transform>, включая возможность вводить здесь элементы, определяемые пользователем или определенные компанией-разработчиком.
- Очень простые таблицы стилей могут быть написаны как единый шаблон, поэтому было показано, как можно опускать элементы <xsl:stylesheet> и <xsl:template match="/">, чтобы таблица стилей XSLT больше походила на таблицы простых языков шаблонов, с которыми знакомы некоторые пользователи.
- Основа структуры таблиц стилей – идея тела шаблона, последовательности текстовых узлов и конечных литеральных элементов, которые копируются в конечное дерево, а также инструкций и элементов расширения, которые подлежат выполнению. Это естественным образом привело к обсуждению конечных литеральных элементов и шаблонов значений атрибутов, которые применяются для определения переменных атрибутов не только для конечных литеральных элементов, но и для некоторых элементов XSLT.
- Обсуждено, как комитет стандартов W3C пытается гарантировать возможность расширения спецификаций и компаниями-разработчиками, и самим W3C, без неблагоприятных последствий для переносимости таблиц стилей. Показано, как работает таблица стилей, даже если она использует «фирменные» функции расширения и элементы расширения, которые не могут быть доступными во всех реализациях.

- Наконец, довольно подробно обсуждено, как таблицы стилей XSLT обрабатывают пробельные символы в исходном документе, в самой таблице стилей и в конечном дереве.

На этом преамбула заканчивается. В следующих четырех главах приведены подробные спецификации элементов XSLT, которые можно использовать в таблице стилей; выражений XPath, которые можно использовать в атрибутах некоторых из этих элементов; образцов соответствия для узлов, которыми можно воспользоваться для установления шаблонных правил; а также стандартных функций, которые доступны для использования в выражениях XPath.

4

Элементы XSLT

В этой главе в алфавитном порядке приведены сведения по каждому из элементов XSLT. Эти сведения включают:

- Краткое описание назначения элемента
- Указание, где в спецификации XSLT описан данный элемент
- Примерный обзор формата элемента, указывающий, где в таблице стилей он может находиться, какие атрибуты для него допустимы и каким может быть его содержимое (дочерние элементы)
- Задание формальных правил, определяющих поведение элемента
- Раздел, в котором приведены советы по использованию данного элемента XSLT
- Наконец, примеры кода, использующего данный элемент, демонстрирующие контекст, в котором он может использоваться

Раздел «Формат» для каждого элемента содержит структуру его синтаксиса, предназначенную для быстрого напоминания имен и типов атрибутов и других ограничений на контекст. Здесь выбрана интуитивная форма изложения: дается только обзор правил, а полностью они излагаются в последующих разделах «Расположение», «Атрибуты» и «Содержимое».

В этой главе использован ряд терминов, с которыми стоит ознакомиться перед чтением главы. Более полные объяснения приведены в главах 2 и 3, а следующие описания должны служить просто краткими памятками.

Более глубокое толкование терминов дано в глоссарии.

Термин	Описание
шаблон значения атрибута	Атрибут, значение которого может содержать выражения, заключенные в фигурные скобки, например «url="../{$href}"». Термин **шаблон** здесь не имеет никакого отношения к шаблонам XSLT. Встроенные выражения могут использоваться в значении атрибута (или считаются таковыми), только если атрибут явно обозначен как шаблон значения атрибута. Шаблоны значений атрибутов описаны более подробно в главе 3 в соответствующем разделе.
порядок появления в документе	Порядок узлов в исходном дереве, который соответствует последовательности появления соответствующих конструкций в исходном XML-документе: элемент предшествует своим непосредственным потомкам, а они упорядочиваются в соответствии с их положением в исходном документе.
выражение	Многие элементы XSLT имеют атрибуты, значение которых – выражение. Это всегда означает выражение XPath: его полное определение дается в главе 5. Выражение возвращает значение, которое может быть строкой, числом, логическим значением, набором узлов или внешним объектом. Эти типы данных описаны подробно в главе 2 в разделе «Типы данных».
подвергать обработке	Инструкции и тела шаблонов могут подвергаться обработке. Если говорить, что инструкции исполняются, это будет звучать довольно помпезно, но в этом есть некоторая логика: можно рассматривать тело шаблона в таблице стилей как образец множества фрагментов выходного документа, каждый из которых формируется путем создания нового экземпляра образца.
инструкция	Любой элемент, используемый в теле шаблона и не являющийся конечным литеральным элементом: в частности, инструкция XSLT или элемент расширения. Элемент <xsl:if> – инструкция, но элемент <xsl:strip-space> – нет, потому что <xsl:if> находится в теле шаблона, а <xsl:strip-space> – нет. Элементы расширения описаны в главе 3 в соответствующем разделе.
конечный литеральный элемент	Элемент таблицы стилей, используемый в *теле шаблона*, который копируется в выходной документ: например (при генерировании HTML) <p> или <td>. Конечные литеральные элементы описаны в главе 3 в соответствующем разделе.
образец	Некоторые элементы XSLT имеют атрибуты, значение которых должно служить образцом. Синтаксис образцов определен в главе 6. Образец – это проверочный критерий, который может быть применен к узлам для выяснения, соответствуют ли они ему. Например, образцу «title» соответствуют все элементы «title», а образцу «text()» соответствуют все текстовые узлы.

Термин	Описание
полное имя	XML-имя, не обязательно уточненное префиксом пространства имен. Примеры полных имен без префикса – «color» и «date-due». Примеры имен с префиксом – «xsl:choose» и «html:table». Когда имя имеет префикс, он всегда должен соответствовать объявлению пространства имен, область действия которого в таблице стилей распространяется на место, где используется полное имя. Дополнительные сведения о пространствах имен можно найти в соответствующем разделе главы 2.
таблица стилей	Вообще, под **таблицей стилей** подразумевается основной модуль таблицы стилей вместе со всеми модулями таблицы стилей, вложенными в него с помощью элементов <xsl:include> и <xsl:import>. Когда речь идет об одном из этих компонентов отдельно, он называется здесь **модулем таблицы стилей**.
тело шаблона	Последовательность инструкций и конечных литеральных элементов, содержащихся внутри (то есть являющихся непосредственными потомками) другого XSLT-элемента. Содержащий их элемент не обязательно должен быть элементом <xsl:template>; многие другие элементы в XSLT, например <xsl:if> и <xsl:variable>, также имеют в качестве содержимого тело шаблона. В спецификации XSLT это называется просто **шаблоном**, но в книге это названо **телом шаблона**, чтобы избежать путаницы с **шаблонным правилом** (элементом <xsl:template> с атрибутом match) и **именованным шаблоном** (элементом <xsl:template> с атрибутом name).
шаблонное правило	Элемент <xsl:template>, который имеет атрибут match.
временное дерево	Значение переменной, устанавливаемое с помощью элемента <xsl:variable>, который содержит тело шаблона, например <xsl:variable>Очень хороший язык</xsl:variable> Эта конструкция называется в спецификации XSLT 1.0 **фрагментом конечного дерева**, и она рассматривается как отдельный тип данных, отличающийся от других типов. Однако в XSLT 1.1 термин «фрагмент конечного дерева» больше не используется, а тип данных этого значения – просто набор узлов (фактически он состоит из единственного узла – корня дерева).
элемент XSLT	Любой из стандартных элементов пространства имен XSLT, перечисленных в этой главе, например <xsl:template> или <xsl:if>.

Для облегчения поиска XSLT-элементы перечисляются здесь в алфавитном порядке, так как я по собственному опыту знаю, насколько трудно найти нужный элемент в спецификации XSLT, где они упорядочены по функциональности. Алфавитный порядок хорош, если знаете, что ищете, но если эта книга для читателя – его первое знакомство с XSLT, возникает проблема, что связанные вещи не собраны вместе. И если пытаться читать последова-

тельно, то начать придется с элемента <xsl:apply-imports>, с которым на кратком обучающем курсе стоило бы знакомить слушателей к концу недели.

По этой причине сделана попытка некоторого упорядочения и группировки, чтобы те, кто плохо знаком с предметом, знали, с чего лучше начать:

Группировка	Элементы
Элементы, используемые для определения шаблонных правил и управления способом их вызова	<xsl:template> <xsl:apply-templates> <xsl:call-template>
Элементы, определяющие структуру таблицы стилей	<xsl:stylesheet> <xsl:include> <xsl:import>
Элементы, используемые для генерирования вывода	<xsl:value-of> <xsl:element> <xsl:attribute> <xsl:comment> <xsl:processing-instruction> <xsl:text>
Элементы, используемые для определения переменных и параметров	<xsl:variable> <xsl:param> <xsl:with-param>
Элементы, используемые для копирования информации из исходного документа в вывод	<xsl:copy> <xsl:copy-of>
Элементы, используемые для условной обработки и для организации циклов	<xsl:if> <xsl:choose> <xsl:when> <xsl:otherwise> <xsl:for-each>
Элементы для управления сортировкой и нумерацией	<xsl:sort> <xsl:number>
Элементы, используемые для управления форматом окончательного вывода	<xsl:output> <xsl:document>

Это охватывает все обычно используемые элементы; остальное можно классифицировать только как «разное».

xsl:apply-imports

Инструкция <xsl:apply-imports> применяется при использовании импортируемых таблиц стилей. Шаблонное правило одного модуля таблицы стилей может переопределять шаблонное правило в импортированном модуле таблицы стилей. Иногда требуется дополнить функциональность правила в импортированном модуле, а не заменять его полностью. Инструкция <xsl:apply-imports> предусмотрена для того, чтобы переопределяющее шаб-

лонное правило могло вызвать переопределяемое шаблонное правило из импортированного модуля.

Здесь просматривается явная аналогия с объектно-ориентированным программированием. Создание модуля таблицы стилей, который импортирует другой модуль, подобно созданию подкласса, методы которого переопределяют методы суперкласса. Инструкция <xsl:apply-imports> ведет себя аналогично функции super() в объектно-ориентированных языках, позволяющей добавить функциональность суперкласса в функциональные возможности подкласса.

Определен в

XSLT, раздел 5.6

Формат

Формат XSLT 1.0:

```
<xsl:apply-imports/>
```

Формат XSLT 1.1:

```
<xsl:apply-imports>
    <xsl:with-param> *
</xsl:apply-imports>
```

Расположение

<xsl:apply-imports> — инструкция, и всегда используется в пределах тела шаблона.

Атрибуты

Нет.

Содержимое

Элемент должен быть либо пустым, либо (только для XSLT 1.1) может содержать один или несколько элементов <xsl:with-param>.

Действие

Инструкция <xsl:apply-imports> зависит от концепции *текущего шаблонного правила*. Шаблонное правило становится текущим шаблонным правилом, когда оно вызывается с помощью <xsl:apply-templates>. Использование <xsl:call-template> не изменяет текущего шаблонного правила. Однако использование <xsl:for-each> делает текущее шаблонное правило пустым на время, пока не завершится выполнение инструкции <xsl:for-each>, после чего восстанавливается предыдущее значение. Текущее шаблонное правило становится пустым также во время вычисления значений глобальных переменных.

<xsl:apply-imports> ищет шаблонное правило, которое соответствует текущему узлу, используя те же правила поиска, как и <xsl:apply-templates>, но рассматривая только те шаблонные правила, которые (а) имеют такой же режим, как текущее шаблонное правило, и (б) определены в таблице стилей, которая была импортирована в таблицу стилей, содержащую текущее шаблонное правило. Подробнее о преимуществе импортирования см. <xsl:import> на стр. 256.

> *В спецификации не говорится точно, что означает* **импортированная в**. *Приемлемо интерпретировать это как модуль таблицы стилей, который является потомком текущей таблицы стилей в дереве импорта. Модуль таблицы стилей S становится непосредственным потомком другого модуля таблицы стилей T в дереве импорта, если он импортируется при помощи элемента <xsl:import> в T или в модуле, который включен в T прямо или косвенно через элемент <xsl:include>. Если модуль S импортирован в модуль T, то шаблоны, определенные в S, всегда будут иметь более низкое преимущество импортирования, чем шаблоны, определенные в T.*

Рабочий проект спецификации XSLT 1.1 позволяет с помощью элементов <xsl:with-param>, содержащихся внутри элемента <xsl:apply-imports>, определять параметры, передаваемые вызванному шаблону. Они действуют так же, как параметры для инструкций <xsl:call-template> и <xsl:apply-templates>: если имя передаваемого параметра соответствует имени элемента <xsl:param> в вызванном шаблоне, параметр примет это значение, в противном случае он примет значение по умолчанию, установленное для элемента <xsl:param>. Указание параметров, которые не соответствуют никакому элементу <xsl:param> в вызванном шаблонном правиле, не вызовет ошибку, – эти параметры просто будут игнорироваться.

Использование и примеры

Образцом возможного использования <xsl:apply-imports> может служить следующий пример.

Первая таблица стилей, a.xsl, содержит универсальные правила для отображения элементов. Например, она могла бы содержать следующее универсальное шаблонное правило для отображения дат:

```
<xsl:template match="дата">
   <xsl:value-of select="день"/>
   <xsl:text>/</xsl:text>
   <xsl:value-of select="месяц"/>
   <xsl:text>/</xsl:text>
   <xsl:value-of select="год"/>
</xsl:template>
```

Вторая таблица стилей, b.xsl, содержит правила специального назначения для отображения элементов. Например, требуется, чтобы она отображала даты, которые встречаются в определенном контексте, таким же образом, но полужирным шрифтом. Это можно записать так:

```
<xsl:template match="заголовок/дата">
   <b>
   <xsl:value-of select="день"/>
   <xsl:text>/</xsl:text>
   <xsl:value-of select="месяц"/>
   <xsl:text>/</xsl:text>
   <xsl:value-of select="год"/>
   </b>
</xsl:template>
```

Однако это дублирует большую часть тела шаблона, что не очень хорошо с точки зрения поддержки. Поэтому можно импортировать `a.xsl` в `b.xsl` и записать вместо этого:

```
<xsl:import href="a.xsl"/>
<xsl:template match="заголовок/дата">
   <b>
   <xsl:apply-imports/>
   </b>
</xsl:template>
```

Заметьте, что данная возможность позволяет вызывать только правила с более низким **преимуществом импортирования**, а не с более низким **приоритетом**. Как объясняется для `<xsl:import>` на стр. 256, преимущество импортирования зависит от того, как был загружен модуль таблицы стилей. Приоритет же может быть определен индивидуально для каждого шаблонного правила, как объясняется для `<xsl:template>` на стр. 349. Код, приведенный выше, будет работать, если только шаблонное правило «заголовок/дата» находится в таблице стилей, которая прямо или косвенно импортирует шаблонное правило «дата». Он не будет работать, например, если они находятся в одной и той же таблице стилей, но имеют разные приоритеты.

Во многих ситуациях того же эффекта можно достичь с не меньшим успехом, давая универсальному шаблону имя и вызывая его из шаблонного правила специального назначения посредством инструкции `<xsl:call-template>` (см. стр. 204). Эта методика также работает при переопределении шаблонного правила с более низким приоритетом (и равным преимуществом импортирования). Только в одном случае нельзя ее использовать – когда универсальное шаблонное правило написано кем-то другим и не может быть изменено. Например, эта ситуация могла бы возникнуть, если бы пользователям веб-страниц разрешалось создавать таблицы стилей XSLT, которые изменили бы функционирование авторской таблицы стилей.

Однако этот подход не работает, если нужно, чтобы одно правило переопределяло или дополняло множество других. Автор книги, например, столкнулся с такой ситуацией: один разработчик имел работающую таблицу стилей, но хотел добавить к ней правило: «выводить HTML-тег `<a>` для любого исходного элемента, который имеет атрибут `anchor`». Чтобы не изменять каждое правило в существующей таблице стилей, этого можно достичь, определив новый модуль таблицы стилей, который импортирует первоначальный модуль и содержит единственное правило:

```
<xsl:template match="*[@anchor]">
  <a name="{@anchor}"/>
  <xsl:apply-imports/>
</xsl:template>
```

В разделе <xsl:import> приведен более полный пример использования инструкции <xsl:apply-imports>.

См. также

<xsl:import> на стр. 256

<xsl:param> на стр. 316

<xsl:with-param> на стр. 383

xsl:apply-templates

Инструкция <xsl:apply-templates> определяет набор узлов, которые подлежат обработке, и заставляет систему обрабатывать их, выбирая для каждого соответствующее шаблонное правило.

Определен в

XSLT, раздел 5.4

Формат

```
<xsl:apply-templates select=Выражение mode=ПолноеИмя >
   ( <xsl:with-param> | <xsl:sort> ) *
</xsl:apply-templates>
```

Расположение

<xsl:apply-templates> – инструкция, и всегда используется внутри тела шаблона.

Атрибуты

Имя	Значение	Назначение
select необязательный	Выражение	Набор узлов, которые подлежат обработке. Если атрибут опущен, то обрабатываются все непосредственные потомки текущего узла.
mode необязательный	Полное имя	Режим обработки. Шаблонные правила, используемые для обработки выбранных узлов, должны иметь соответствующий режим.

Определение конструкций выражение и полное имя дано в начале этой главы, а более формально – в главе 5 в разделах «Выражение (Expression)» и «ПолноеИмя (QName)» соответственно.

Содержимое

Нуль или больше элементов <xsl:sort>.

Нуль или больше элементов <xsl:with-param>.

Действие

Элемент <xsl:apply-templates> выбирает в исходном дереве набор узлов и обрабатывает каждый из них отдельно, находя соответствующее данному узлу шаблонное правило. Набор узлов определяется атрибутом select; порядок их обработки – элементами <xsl:sort> (если они есть), а параметры, передаваемые шаблонным правилам – элементами <xsl:with-param> (если они есть). Режим работы подробно объясняется в следующих разделах.

Атрибут select

Если атрибут select присутствует, выражение определяет узлы, которые будут обработаны. Это должно быть выражение XPath, которое возвращает набор узлов. (Понятие набора узлов объясняется наряду с другими типами данных XSLT в соответствующем разделе главы 2.) Например <xsl:apply-templates select="*"/> выбирает все узлы элементов, которые являются непосредственными потомками текущего узла. Запись <xsl:apply-templates select="@width +3"/> была бы ошибочной, потому что значением данного выражения является число, а не набор узлов.

Выражение может выбирать узлы относительно текущего узла (узла, обрабатываемого в данный момент), как в примере выше. Кроме того, оно может производить выбор относительно корневого узла (например, <xsl:apply-templates select="//item"/>) или просто выбирать узлы, ссылаясь на переменную, инициализированную ранее (например, <xsl:apply-templates select="$sales-figures"/>). Подробнее о выражениях XPath рассказывается в главе 5.

Если атрибут select опущен, обрабатываемыми узлами будут непосредственные потомки текущего узла: то есть элементы, текстовые узлы, комментарии и инструкции обработки, которые содержатся непосредственно в текущем узле. Текстовые узлы, состоящие только из пробельных символов, будут обработаны наряду с другими, если они не были удалены из дерева; подробности об этом можно увидеть в разделе об элементе <xsl:strip-space> на стр. 336. В модели дерева XPath, описанной в главе 2, узлы атрибутов **не** считаются непосредственными потомками содержащего их элемента, поэтому они не обрабатываются: если нужно обработать узлы атрибутов, следует явно использовать атрибут select, например <xsl:apply-templates select="@*"/>. Однако чаще для получения значений атрибутов используют инструкцию <xsl:value-of>, описанную на стр. 366.

Отсутствие атрибута select равносильно заданию выражения «child::node()». В обоих случаях выбираются все узлы (элементы, текстовые узлы, комментарии и инструкции обработки), которые являются непосредственными потомками текущего узла. Если текущий узел не является корневым

узлом или узлом элемента, то он не имеет дочерних узлов, поэтому <xsl:apply-templates/> ничего не делает, так как никаких узлов для обработки нет.

Для каждого узла из выбранного набора узлов по очереди выбирается шаблонное правило, и тело этого шаблона подвергается обработке. В общем случае это могут быть различные шаблонные правила для каждого выбранного узла. Внутри данного тела шаблона обрабатываемый узел становится новым текущим узлом, поэтому к нему можно обращаться, используя выражение XPath «.».

Вызываемый шаблон может также определять относительное положение данного узла в списке узлов, выбранных для обработки: в частности, он может использовать функцию position(), чтобы получить позицию этого узла в списке обрабатываемых узлов (первый обрабатываемый узел имеет position()=1 и так далее), и функцию last(), чтобы получить количество узлов в списке. Эти две функции описаны (с примерами) в главе 7. Они дают вызываемому шаблону возможность по мере обработки выводить порядковые номера узлов или предпринимать разные действия по отношению к первому и последнему узлам, или, возможно, использовать разные цвета фона для узлов с нечетными и четными порядковыми номерами.

Сортировка

Если нет дочерних инструкций <xsl:sort>, выбранные узлы обрабатываются в *порядке появления в документе*. В обычном случае, когда все узлы происходят из одного и того же исходного документа, это означает, что они будут обрабатываться в порядке их появления в исходном документе: например узел элемента обрабатывается раньше его непосредственных потомков. Однако узлы атрибутов, принадлежащие одному и тому же элементу, могут обрабатываться в любом порядке, потому что порядок атрибутов не считается в XML существенным. Если в наборе узлов имеются узлы из различных документов, что случается при использовании функции document() (описанной в главе 7), относительный порядок узлов из различных документов не определен, хотя он сохраняется, если один и тот же набор узлов обрабатывается более одного раза.

Если у инструкции <xsl:apply-templates> есть один или несколько дочерних элементов <xsl:sort>, узлы сортируются перед обработкой. Каждая инструкция <xsl:sort> определяет один ключ сортировки. О подробностях управления сортировкой см. в разделе <xsl:sort> на стр. 330. Если определено несколько ключей сортировки, они применяются в порядке от наиболее важного к наименее важному. Например, если первая инструкция <xsl:sort> определяет сортировку по странам, а вторая – по штатам, то узлы будут обрабатываться в порядке штатов в пределах страны. Если два узла имеют равные ключи сортировки, они будут обрабатываться в порядке появления в документе.

Заметьте, что направление осей, используемых для выбора узлов, несущественно. (Направление различных осей описано в главе 5.) Например выражение «select="preceding-sibling::*"» обработает предшествующие текущему узлу одноуровневые элементы в порядке появления их в документе (начи-

ная с первого из них), даже если ось preceding-sibling имеет направление, обратное порядку появления в документе. Направление оси воздействует только на значение любых позиционных спецификаторов, используемых в выражении для выбора. Например, выражение «select="preceding-sibling::*[1]"» выберет первый предшествующий одноуровневый элемент по направлению оси, который (если он есть) непосредственно предшествует текущему узлу.

Выбор шаблонного правила

Для каждого узла, который подлежит обработке, выбирается шаблонное правило. Выбор шаблонного правила для каждого выбранного узла осуществляется независимо; все они могут быть обработаны одним и тем же шаблонным правилом, или для каждого может быть выбрано отдельное шаблонное правило.

Шаблонным правилом, выбираемым для обработки узла, всегда является или элемент <xsl:template> с атрибутом match, или встроенное шаблонное правило, предоставляемое XSLT-процессором.

Элемент <xsl:template> используется для обработки узла, если только он имеет соответствующий режим: то есть атрибут mode элемента <xsl:apply-templates> должен соответствовать атрибуту mode элемента <xsl:template>. Это означает, что у обоих элементов они должны отсутствовать или у обоих должны иметься. Если у обоих элементов они есть, они должны иметь совпадающие имена, а если имя режима содержит префикс пространства имен, то совпадать должен URI пространства имен, но не обязательно сам префикс.

Заметьте, что если атрибут mode опущен, не имеет значения, какой режим использовался первоначально для выбора шаблонного правила, содержащего инструкцию <xsl:apply-templates>. Режим не сохраняется: он возвращается к значению по умолчанию, как только <xsl:apply-templates> используется без атрибута mode.

Элемент <xsl:template> используется для обработки узла, если только узел соответствует образцу, определенному в атрибуте match элемента <xsl:template>.

Если выбранному узлу соответствует несколько элементов <xsl:template>, выбирается один из них на основании его **преимущества импортирования** и **приоритета**, о чем подробно рассказывается в разделе <xsl:template> на стр. 349.

Если нет соответствующих выбранному узлу элементов <xsl:template>, используется встроенное шаблонное правило. Действие встроенного шаблонного правила зависит от типа узла следующим образом:

Тип узла	Действие встроенного шаблонного правила
корневой узел	Вызывает apply-templates для обработки каждого непосредственного потомка корневого узла, используя режим, указанный в вызове <xsl:apply-templates>. Это равносильно действию шаблона с содержимым: `<xsl:apply-templates mode="mode"/>`

Тип узла	Действие встроенного шаблонного правила
элемент	Вызывает apply-templates для обработки каждого непосредственного потомка элемента, используюя режим, указанный в вызове <xsl:apply-templates>. Это равносильно действию шаблона с содержимым: <xsl:apply-templates mode="*mode*"/>
текст	Копирует текстовое значение узла в вывод. Это равносильно действию шаблона с содержимым: <xsl:value-of select=".">
атрибут	Копирует значение атрибута в вывод как текст. Это равносильно действию шаблона с содержимым: <xsl:value-of select=".">
инструкция обработки	Никаких действий
комментарий	Никаких действий
пространство имен	Никаких действий

В случае корневого узла и узлов элементов встроенное шаблонное правило обрабатывает непосредственных потомков выбранного узла в порядке появления в документе, сопоставляя каждого из них с имеющимися шаблонными правилами, как если бы тело шаблона содержало явный элемент <xsl:apply-templates> без атрибута select. В отличие от ситуации с шаблонными правилами, определяемыми пользователями, здесь режим *сохраняется*: он автоматически передается вызываемым правилам. Так, если выполняется <xsl:apply-templates mode="m"/> для элемента, который не имеет соответствующего шаблонного правила, встроенное шаблонное правило выполнит <xsl:apply-templates mode="m"/> для его непосредственных потомков. Этот процесс может, конечно, рекурсивно повторяться для обработки потомков следующего поколения и так далее.

Хотя встроенные шаблонные правила передают режим, в котором они были вызваны, это не относится к параметрам. Они не передаются встроенными шаблонными правилами, о чем будет рассказано в следующем разделе.

Параметры

Если есть элементы <xsl:with-param>, являющиеся непосредственными потомками элемента <xsl:apply-templates>, они определяют параметры, которые доступны вызываемым шаблонным правилам. Каждому шаблонному правилу, которое применяется, доступны одни и те же параметры, несмотря на то, что для обработки разных узлов в списке могут вызываться разные шаблонные правила.

Каждый элемент <xsl:with-param> вычисляется таким же образом, как элемент <xsl:variable>. В частности:

- Если он имеет атрибут select, он вычисляется как выражение XPath.

- Если атрибута select нет, и элемент <xsl:with-param> пуст, его значение – пустая строка.
- В других случаях значение параметра – временное дерево. Дерево формируется путем применения тела шаблона, являющегося содержимым элемента <xsl:with-param>, а его значением является набор узлов, содержащий один узел – корень этого дерева. Переменные, значениями которых являются деревья, описаны в главе 2, а более формально – в разделе <xsl:variable> на стр. 370.

Не определено, вычисляется ли параметр только однажды, или вычисляется заново для каждого узла из набора узлов. Если значение не требуется (например, когда набор узлов пуст или когда ни один из узлов не соответствует шаблону, использующему этот параметр), тогда не определено, вычисляется ли параметр вообще. Обычно это не имеет значения, потому что неоднократное вычисление параметра будет каждый раз иметь тот же самый эффект. Но за этим нужно следить, если для получения значения параметра используется вызов внешней функции, имеющей побочный эффект, например чтение следующей записи из базы данных.

Если имя дочернего элемента <xsl:with-param> соответствует имени элемента <xsl:param> в выбранном шаблонном правиле, то значение элемента <xsl:with-param> присваивается соответствующей переменной <xsl:param>.

Если есть дочерний элемент <xsl:with-param>, который не соответствует имени ни одного элемента <xsl:param> в выбранном шаблонном правиле, то он игнорируется. Это не считается ошибкой.

Если в выбранном шаблонном правиле есть элемент <xsl:param>, не имеющий соответствующих элементов <xsl:with-param> в элементе <xsl:apply-templates>, то параметру присваивается значение по умолчанию; см. подробнее в разделе <xsl:param> на стр. 316. Опять же, это не ошибка.

Если выбранное шаблонное правило является встроенным правилом, то все передаваемые параметры игнорируются. Они не передаются никаким шаблонным правилам, вызываемым встроенным правилом, как передается режим. Это может вызвать трудности, когда есть два правила, подобные следующим:

```
<xsl:template match="раздел">
<ol>
  <xsl:apply-templates>
    <xsl:with-param name="в-разделе" select="true()"/>
  </xsl:apply-templates>
</ol>
</xsl:template>

<xsl:template match="предложение">
  <xsl:param name="в-разделе" select="false()"/>
  . . .
</xsl:template>
```

Когда <предложение> содержится непосредственно в <разделе>, параметр $в-разделе, как и ожидается, будет принимать значение "истина". Но если есть промежуточный элемент без явного шаблонного правила, из-за которого <предложение> не является непосредственным потомком элемента <раздел>, шаблонное правило <предложение> все еще будет вызываться, но параметр $в-разделе будет иметь значение "ложь". Причина в том, что встроенное шаблонное правило для промежуточного элемента вызывает <xsl:apply-templates> для обработки его непосредственных потомков <предложение>, но не передает никаких параметров.

Назначение вывода

Вызываемое шаблонное правило не возвращает результат общепринятым способом, как это делают функции во многих языках программирования (в том числе в XPath). Единственное, что может сделать шаблонное правило, чтобы сохранить результат, – записать узлы в текущее назначение вывода.

Когда начинается выполнение таблицы стилей, текущим назначением является основное конечное дерево, сформированное при выполнении таблицы стилей, поэтому любой вывод, произведенный шаблоном, поступает в основной выходной документ.

Внутри тела элементов <xsl:variable>, <xsl:param> или <xsl:with-param> текущее назначение вывода заменяется на новое временное дерево. Когда формирование этого дерева завершается, переменная или параметр будет иметь значение, которое является набором узлов, содержащим единственный узел, являющийся корневым узлом временного дерева.

XSLT 1.1 допускает также несколько конечных деревьев. Когда внутри тела шаблона встречается инструкция <xsl:document> (описанная на стр. 231), начинается формирование нового дерева, которое становится текущим назначением вывода. По завершении формирования это дерево сериализуется так же, как и основное конечное дерево.

Некоторые другие инструкции XSLT также изменяют назначение вывода. Примерами являются инструкции <xsl:attribute>, <xsl:comment> и <xsl:processing-instruction>. Для них новым назначением вывода служит текстовое значение создаваемого узла, и запись в вывод нетекстовых узлов будет ошибкой. Другой пример – <xsl:message>, где новым назначением вывода является текст сообщения, которое будет выведено. В этом случае допустимо записывать нетекстовые узлы, но способ их обработки зависит от конкретных программных продуктов.

Вызов инструкции <xsl:apply-templates> не изменяет текущее назначение вывода. Так, если <xsl:apply-templates> вызывается в теле элемента <xsl:variable>, вывод вызванных шаблонных правил и любых шаблонных правил, которые они, в свою очередь, вызывают, добавляется во временное дерево, которое становится значением этой переменной.

Заметьте, что вполне допустимо использовать <xsl:apply-templates> внутри тела определения глобальной переменной, например:

```
<xsl:variable name="оглавление">
   <xsl:apply-templates mode="оглавление"/>
</xsl:variable>
```

В данной ситуации текущий узел служит корнем основного исходного документа, поэтому <xsl:apply-templates> обрабатывает непосредственных потомков корневого узла. Не определено, что происходит, если шаблон, применяемый таким образом, пытается обращаться к глобальной переменной, но это, вероятно, вызовет ошибку.

Использование

В следующих разделах даны некоторые полезные советы по использованию <xsl:apply-templates>. Сначала обсуждается, когда использовать <xsl:apply-templates>, а когда – <xsl:for-each>. Затем объясняется, как использовать режимы.

Сопоставление <xsl:apply-templates> и <xsl:for-each>

Элемент <xsl:apply-templates> наиболее полезен для обработки элемента, который имеет потомков нескольких различных типов в непредсказуемой последовательности. Такой метод проектирования называют *основанным на правилах*: в теле каждого индивидуального шаблонного правила объявляется, к каким узлам оно имеет отношение, в отличие от шаблонного правила для родительского узла, в котором подробно определено, как должен обрабатываться каждый из его непосредственных потомков. Подход, основанный на правилах, особенно хорош для документов, которые со временем предполагается развивать. По мере добавления новых дочерних элементов могут добавляться также шаблонные правила для их обработки без изменения логики для родительских элементов, в которых они содержатся.

Этот стиль обработки иногда называется **форсированной** обработкой (**push** processing). Он знаком тем, кто пользовался языками обработки текстов, такими как awk или Perl, но может быть неизвестен любителям процедурного программирования на C++ или Visual Basic.

Когда структура достаточно регулярна и предсказуема, перемещаться по документу может оказаться проще с использованием <xsl:for-each> или путем обращения к нужным данным, прямо используя <xsl:value-of>. Иногда это называют **извлекающей** обработкой (**pull** processing). Инструкция <xsl:value-of> позволяет выбирать данные из XML-документа, используя выражения XPath различной сложности. В этом смысле она подобна оператору SELECT в SQL.

Уникальным достоинством XSLT является способность совмещать эти два стиля программирования. Более подробно оба эти подхода обсуждаются и сопоставляются в главе 9.

Режимы

Режимы полезны в тех случаях, когда одни и те же данные должны обрабатываться несколько раз.

Классическим примером является формирование оглавления. Основная часть вывода может быть получена при обработке узлов в режиме по умолчанию, в то время как оглавление создается при обработке тех же узлов в другом режиме: «mode="ОГЛ"».

В следующем примере показано нечто подобное: он демонстрирует сцену из пьесы, добавляя в начале страницы список героев, участвующих в этой сцене.

Пример: Использование режимов

Исходный файл

Исходный файл, scene.xml, содержит сцену из пьесы (в частности, 1-ю сцену 1-го акта трагедии Шекспира «Отелло» – размеченной в XML Джоном Босаком (Jon Bosak).[1]

Начинается так:

```
<?xml version="1.0"?>
<SCENE><TITLE>СЦЕНА I.  Венеция. Улица.</TITLE>
<STAGEDIR>Входят РОДРИГО и ЯГО</STAGEDIR>

<SPEECH>
<SPEAKER>РОДРИГО</SPEAKER>
<LINE>Ни слова больше. Это низость, Яго.</LINE>
<LINE>Ты деньги брал, а этот случай скрыл.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>ЯГО</SPEAKER>
<LINE>Я сам не знал. Вы не хотите слушать.</LINE>
<LINE>Об этом я не думал, не гадал.</LINE>
</SPEECH>
  и т.д.
</SCENE>
```

Таблица стилей

Таблица стилей scene.xsl предназначена для отображения этой сцены в HTML. Вот как она начинается:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="SCENE">
<html><body>
   <xsl:apply-templates select="TITLE"/>
```

[1] Перевод Б. Пастернака. – *Примеч. ред.*

```
    <xsl:variable name="speakers"
                  select="//SPEAKER[not(.=preceding::SPEAKER)]"/>
    <h2>Действующие лица: <xsl:apply-templates
                  select="$speakers" mode="cast-list"/></h2>
    <xsl:apply-templates select="*[not(self::TITLE)]"/>
</body></html>
</xsl:template>
```

Шаблонное правило, показанное выше, соответствует элементу <SCENE>. Сначала оно отображает элемент <TITLE> (если он есть), используя соответствующее шаблонное правило. Затем оно устанавливает переменную, названную «speakers», которая представляет набор узлов, содержащих все отдельные элементы <SPEAKER>, встречающиеся в документе. Довольно сложный атрибут select использует выражение «//SPEAKER» для выбора всех элементов <SPEAKER>, которые являются потомками корня (другими словами, всех таких элементов), а затем уточняет это с помощью предиката в квадратных скобках, исключающего те элементы <SPEAKER>, которые встречаются повторно. Результатом является список действующих лиц, в котором каждый из них упоминается только один раз.

После этого шаблонное правило вызывает <xsl:apply-templates> для обработки этого набора действующих лиц в режиме «cast-list» (удачный побочный эффект, что они будут перечислены в порядке появления). Наконец, оно снова вызывает <xsl:apply-templates>, на этот раз в режиме по умолчанию, для обработки всех элементов («*»), которые не являются элементами <TITLE> (потому что заголовок уже обработан).

Таблица стилей продолжается следующим образом:

```
<xsl:template match="SPEAKER" mode="cast-list">
    <xsl:value-of select="."/>
    <xsl:if test="not(position()=last())">, </xsl:if>
</xsl:template>
```

Это шаблонное правило определяет, как элемент <SPEAKER> должен обрабатываться в режиме «cast-list». Тело шаблона выводит после имени действующего лица запятую, если данное действующее лицо – не последнее в списке.

Наконец, остальные шаблонные правила определяют, как должен выводиться каждый элемент при обработке в режиме по умолчанию:

```
<xsl:template match="TITLE">
<h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="STAGEDIR">
<i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="SPEECH">
<p><xsl:apply-templates/></p>
</xsl:template>
```

```
<xsl:template match="SPEAKER">
<b><xsl:apply-templates/></b><br/>
</xsl:template>

<xsl:template match="LINE">
<xsl:apply-templates/><br/>
</xsl:template>
</xsl:transform>
```

Вывод

Форматирование HTML будет зависеть от используемого XSLT-процессора, но без учета деталей форматирования, это должно начинаться примерно так:

```
<html>
   <body>
      <h1>СЦЕНА 1. Венеция. Улица.</h1>
      <h2>Действующие лица: РОДРИГО, ЯГО</h2>
      <i>Входят РОДРИГО и ЯГО</i>
      <p>
         <b>РОДРИГО</b><br>
         Ни слова больше. Это низость, Яго.
         Ты деньги брал, а этот случай скрыл.<br>
      </p>
      <p>
         <b>ЯГО</b><br>
         Я сам не знал. Вы не хотите слушать.
         Об этом я не думал, не гадал.<br>
      </p>
   </body>
</html>
```

Иногда удобно использовать именованные режимы, даже если они не обязательны, чтобы в документе более четко прослеживалась взаимосвязь между вызывающими и вызываемыми шаблонами, а выбор шаблонных правил ограничивался даже более явно, чем это достигается на основании приоритетов шаблонных правил. Это может даже улучшить производительность, поскольку сокращает количество рассматриваемых правил, хотя данный эффект, вероятно, будет незначительным.

Например, предположим, что элемент `<стихотворение>` состоит из ряда элементов `<строфа>` и что первый элемент `<строфа>` должен выводиться иначе, чем остальные. Традиционный способ достижения этого – следующий:

```
<xsl:template match="стихотворение">
. . .
   <xsl:apply-templates select="строфа"/>
. . .
</xsl:template>
<xsl:template match="строфа[1]">
. . .
```

```
</xsl:template>
<xsl:template match="строфа">
. . .
</xsl:template>
```

Это основано на приоритетах по умолчанию, что гарантирует применение к каждой строфе правильного шаблонного правила. Как объясняется в разделе **<xsl:template> на стр. 349**, приоритет по умолчанию для образца «строфа[1]» выше, чем приоритет по умолчанию для «строфа».

Другой способ выполнения того же самого, возможно, менее традиционный, но столь же эффективный – следующий:

```
<xsl:template match="стихотворение">
. . .
   <xsl:apply-templates select="строфа[1]" mode="первая"/>
   <xsl:apply-templates select="строфа[position() &gt; 1]" mode="остальные"/>
. . .
</xsl:template>

<xsl:template match="строфа" mode="первая">
. . .
</xsl:template>

<xsl:template match="строфа" mode="остальные">
. . .
</xsl:template>
```

Еще одно решение, дающее даже более тонкий контроль, – использовать <xsl:for-each> и <xsl:call-template>, чтобы точно регулировать, какие шаблонные правила к каким узлам применять, вообще отказавшись от механизмов сопоставления с образцами элемента <xsl:apply-templates>.

Выбор подходящего способа в значительной степени зависит от индивидуального стиля пользователя, и очень трудно доказать, что какой-то из способов во всех отношениях лучше, чем другой. Однако если обнаруживается, что образцы соответствия, используемые при определении шаблонного правила, становятся чрезвычайно сложными и контекстно-зависимыми, тогда, вероятно, возникнут проблемы как с производительностью, так и с поддержкой шаблонов, и в этом случае имеет смысл обратиться к управлению выбором шаблонных правил в вызывающем коде с использованием режимов или вызовов шаблонов по именам.

Примеры

`<xsl:apply-templates/>`	Обрабатывает всех непосредственных потомков текущего узла.
`<xsl:apply-templates select="para"/>`	Обрабатывает все элементы <para>, которые являются непосредственными потомками текущего узла.

`<xsl:apply-templates select="//*" mode="огл"/>`	Обрабатывает все элементы в документе в режиме «огл».
`<xsl:apply-templates select="para"> <xsl:with-param name="отступ" select="$n+4"/> </xsl:apply-templates>`	Обрабатывает все элементы <para>, которые являются непосредственными потомками текущего узла, устанавливая значение параметра отступ в каждом вызываемом шаблоне равным значению переменной $n плюс 4.
`<xsl:apply-templates select="//книга"> <xsl:sort select="@isbn"/> </xsl:apply-templates>`	Обрабатывает все элементы <книга> в документе, сортируя их по возрастанию значения их атрибута isbn.

См. также

<xsl:for-each> на стр. 248

<xsl:template> на стр. 349

<xsl:with-param> на стр. 383

xsl:attribute

Элемент <xsl:attribute> выводит имя и значение атрибута в текущее назначение вывода. Это удается, если только узел элемента добавлен в это дерево вывода, а после его добавления не добавлялись никакие другие узлы, кроме узлов атрибутов.

Определен в

XSLT, раздел 7.1.3

Формат

```
<xsl:attribute name={ПолноеИмя} namespace={uri}>
      тело шаблона
</xsl:attribute>
```

Расположение

<xsl:attribute> может использоваться или как инструкция в *теле шаблона*, или внутри элемента <xsl:attribute-set>.

Атрибуты

Имя	Значение	Назначение
name обязательный	Шаблон значения атрибута, возвращающий полное имя	Имя атрибута, который будет сгенерирован
namespace необязательный	Шаблон значения атрибута, возвращающий URI	URI пространства имен генерируемого атрибута

Содержимое

Тело шаблона.

Действие

Инструкция <xsl:attribute> может находиться в теле шаблона или внутри элемента <xsl:attribute-set>.

Оба атрибута – и name, и namespace – могут быть написаны в виде шаблонов значений атрибутов: то есть они могут содержать выражения, заключенные в фигурные скобки, например <xsl:attribute name="{$chosenName}"/>. Шаблоны значений атрибутов подробно обсуждаются в соответствующем разделе главы 3.

Инструкцию <xsl:attribute> следует применять только во время применения инструкции для добавления узла элемента в конечное дерево. Это могут быть инструкции <xsl:element> или <xsl:copy> или конечные литеральные элементы. Кроме того, атрибут должен выводиться раньше добавления к узлу элемента любых дочерних узлов (текстовых узлов, элементов, комментариев или инструкций обработки). Очень часто инструкция <xsl:attribute> содержится прямо в инструкции, которая записывает элемент, например:

```
<table>
   <xsl:attribute border="2"/>
</table>
```

но это необязательно, например можно делать и так:

```
<table>
   <xsl:call-template name="set-border"/>
</table>
```

а затем создать атрибут внутри шаблона «set-border».

Правило, что атрибуты элемента должны записываться в конечное дерево раньше добавления любых дочерних узлов, введено для удобства разработчиков процессоров. Благодаря ему XSLT-процессор фактически не обязан формировать конечное дерево в памяти, а вместо этого каждый узел может записываться в XML-файл, как только он сгенерирован. Если бы не это правило, программа до самого конца не могла бы записать первый открывающий тег, потому что всегда бы имелся шанс добавления к нему нового атрибута.

Если во время применения <xsl:attribute> на текущем узле элемента уже имеется атрибут с таким же именем (то есть другой атрибут, имя которого имеет такие же локальную часть и URI пространства имен), то новый атрибут перезаписывает прежний. Это не ошибка: фактически этот механизм очень важен, когда для добавления атрибута к элементу вывода пользуются именованными наборами атрибутов. Именованные наборы атрибутов описаны в разделе <xsl:attribute-set> на стр. 198.

Имя атрибута

Имя нового атрибута получается путем раскрытия атрибута name. Результатом раскрытия шаблона значения атрибута должно быть полное имя, которое является действительным XML-именем с необязательным префиксом пространства имен. В наиболее общем случае это будет простое имя без двоеточия, и в этом случае имя выводимого атрибута будет таким же, как это полное имя.

Локальная часть имени атрибута вывода всегда будет такой же, как локальная часть полного имени, переданного в виде значения атрибута name. Префиксы могут отличаться, как будет показано ниже.

Атрибут namespace

О пространствах имен XML говорилось в соответствующем разделе главы 2. Атрибут namespace инструкции <xsl:attribute> позволяет задавать URI пространства имен генерируемого атрибута.

В спецификации XSLT четко указано, что нельзя использовать <xsl:attribute> для генерирования объявлений пространств имен, задавая имя атрибута «xmlns» или «xmlns:*». Объявления пространств имен будут автоматически добавляться к выводу всякий раз, когда генерируются элементы или атрибуты, требующие их.

Если элемент <xsl:attribute> имеет атрибут namespace, то его значением (при раскрытии шаблона значения атрибута) должен быть URI, задающий пространство имен. Однако система не проверяет его соответствия какому-либо синтаксису URI, поэтому в действительности может использоваться любая строка.

Префикс имени выводимого атрибута обычно такой же, как префикс полного имени, но это не гарантируется, и при некоторых обстоятельствах система может назначить другой префикс. Одна из таких ситуаций – когда к одному элементу добавляются два атрибута, использующие один и тот же префикс, но для разных URI пространств имен; другая – когда префикс принимает зарезервированное значение «xmlns». Если это случается, система может назначить любой префикс пространства имен по своему усмотрению. Какой бы префикс система ни выбрала, должно гарантироваться, что соответствующий элемент в конечном дереве имеет узел пространства имен, который связывает этот префикс с правильным URI пространства имен. Это гарантирует, что конечный вывод содержит объявление пространства имен для этого префикса. Это важно также при записи атрибутов во временное дерево, которое позже будет читаться: благодаря этому функция name() всегда будет возвращать полное имя, префикс которого правильно задает URI пространства имен узла атрибута.

Если нет атрибута namespace:

- Если заданное полное имя содержит префикс, этот префикс должен быть префиксом пространства имен, которое действует в этом месте таблицы стилей. URI пространства имен в выводе будет URI пространства имен, связанного в таблице стилей с этим префиксом.

- Иначе, заданное полное имя используется непосредственно как имя выводимого атрибута. Пространство имен по умолчанию (которое объявляется с помощью выражения «xmlns="uri"») **не** используется.

Если имеется атрибут namespace, и его значение – пустая строка, то выводимый атрибут будет иметь пустой URI пространства имен. Он поэтому будет выводиться без префикса. Любой префикс в значении атрибута name будет игнорироваться.

Если имеется атрибут namespace, и его значение – не пустая строка, имя выводимого атрибута получит URI пространства имен путем вычисления атрибута namespace. Тогда URI пространства имен, связанный с любым префиксом в полном имени, полученном из атрибута name, будет игнорироваться (хотя он все равно должен быть действительным). Если не предоставлен ни один префикс, XSLT-процессору может понадобиться генерировать префикс для имени выводимого атрибута. Например, если записать:

```
<table>
<xsl:attribute name="width" namespace="http://acme.org/">
   <xsl:text>200</xsl:text>
</xsl:attribute>
</table>
```

тогда вывод может выглядеть так:

```
<table ns0001:width="100" xmlns:ns0001="http://acme.org"/>
```

Значение атрибута

Значением нового атрибута является строка, полученная при применении тела шаблона.

Если применяемое тело шаблона генерирует любые другие, а не текстовые узлы (например, элементы или комментарии), это ошибка. Система может сообщить об этом как о фатальной ошибке или может продолжить выполнение, просто проигнорировав эти узлы и все, что они содержат.

Использование

Разработчики, знакомые с диалектом WD-xsl от Microsoft, часто используют инструкцию <xsl:attribute>, когда нужно вывести значение атрибута. Однако в XSLT есть много других способов генерировать атрибут в конечном дереве. В этом разделе сравниваются различные подходы.

Когда выводимый элемент генерируется с использованием конечного литерального элемента, самый простой способ задать атрибуты – включить их непосредственно как атрибуты конечного литерального элемента. Это можно делать, даже когда значение получено из информации в исходном документе, потому что значение может быть сгенерировано, используя шаблон значения атрибута, например:

```
<body bgcolor="#{@red}{@green}{@blue}">
```

Это сцепляет три атрибута текущего узла в исходном дереве для создания единственного атрибута в конечном дереве. Шаблоны значений атрибутов описаны в соответствующем разделе главы 3.

Использование <xsl:attribute> дает больше контроля, чем непосредственная запись атрибута с использованием шаблонов значений атрибутов. Это удобно при следующих обстоятельствах:

- Родительский элемент выводится при помощи <xsl:element> или <xsl:copy> (а не конечного литерального элемента)
- Используется условие для решения, выводить атрибут или нет
- Имя атрибута вычисляется во время выполнения
- Для вычисления значения атрибута применяется сложная логика
- Атрибут является одним из компонентов набора, который может быть удобно сгруппирован с помощью <xsl:attribute-set>
- Выводимый атрибут принадлежит пространству имен, которое не присутствует в исходном документе или в таблице стилей

Третий способ вывода атрибутов состоит в копировании их из исходного дерева в конечное дерево с помощью <xsl:copy> или <xsl:copy-of>. Это работает, если только атрибут, который нужно генерировать, имеет то же название и то же значение, что и атрибут в источнике.

<xsl:copy> может использоваться, когда текущий узел в исходном документе – узел атрибута. Необязательно, чтобы элемент, которому принадлежит атрибут, выводился с использованием <xsl:copy>, например, следующий код гарантирует, что атрибуты ширина, высота и глубина исходного элемента <посылка> копируются в выходной элемент <пакет>, а его атрибуты ценность и отправитель отбрасываются:

```
<xsl:template match="посылка">
  <пакет>
    <xsl:apply-templates select="@*"/>
  </пакет>
</xsl:template>
<xsl:template match="посылка/@ширина | посылка/@высота | посылка/@глубина">
    <xsl:copy/>
</xsl:template>
<xsl:template match="посылка/@ценность | посылка/@отправитель"/>
```

В этом примере для обработки всех атрибутов элемента <посылка> используется <xsl:apply-templates>. Некоторые из них соответствуют одному шаблонному правилу, которое копирует атрибут в элемент вывода, в то время как другие соответствуют пустому шаблонному правилу, которое не делает ничего.

Того же эффекта можно достичь более легким способом, используя <xsl:copy-of>:

```
<xsl:template match="посылка">
<пакет>
```

```
    <xsl:copy-of select="@ширина | @высота | @глубина"/>
</пакет>
</xsl:template>
```

Здесь выражение select выбирает набор узлов, который содержит атрибуты ширина, высота и глубина текущего узла, а инструкция <xsl:copy-of> копирует этот набор узлов. Оператор «|», описанный в главе 5, создает объединение двух наборов узлов.

Если нужно скопировать все атрибуты текущего узла в конечное дерево, проще всего сделать это так: <xsl:copy-of select="@*"/>. Если нужно скопировать все атрибуты, кроме нескольких определенных, можно или использовать описанный выше подход, основанный на правилах, или записать нечто вроде:

```
<xsl:copy-of select=@* [not(name() = 'ценность' or name()='отправитель')]"/>
```

Однако если атрибуты принадлежат пространству имен, лучше избегать этого, потому что в таком случае результат функции name() может меняться в зависимости от выбора префикса. Здесь нельзя использовать выражение «not(self::ценность)», как при работе с элементами, так как выражение «self::ценность» будет соответствовать только элементу с именем ценность, но не атрибуту. Более надежное решение – проверять и local-name(), и namespace-uri().

Примеры

Следующий пример выводит HTML-элемент <OPTION> с атрибутом SELECTED, включаемым, только когда логическая переменная $selected истинна. (Вывод XML был бы <OPTION SELECTED="SELECTED">, но метод вывода HTML преобразует это в <OPTION SELECTED>.)

Пример: Генерирование атрибута при условии

Исходный документ

Исходный файл – страны.xml.

```
<?xml version="1.0"?>
<страны>
  <страна название="Франция"/>
  <страна название="Германия"/>
  <страна название="Израиль"/>
  <страна название="Япония"/>
  <страна название="Польша"/>
  <страна название="Соединенные Штаты" выбрана="да"/>
  <страна название="Венесуэла"/>
</страны>
```

Таблица стилей

Файл таблицы стилей – options.xsl.

Это полная таблица стилей, использующая упрощенный синтаксис таблиц стилей, описанный в главе 3. Она выводит элемент выбора опции HTML, в котором для опции, обозначенной в исходном документе XML как «выбрана="да"», установлен атрибут selected.

```
<html version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<body>
<h1>Выберите страну:</h1>
<select id="country">
<xsl:for-each select="//страна">
   <option value="{@название}">
     <xsl:if test="@выбрана='да'">
         <xsl:attribute name="selected">selected</xsl:attribute>
     </xsl:if>
     <xsl:value-of select="@название"/>
   </option>
</xsl:for-each>
</select>
<hr/>
</body>
</html>
```

Вывод

Вывод (показанный с открытым элементом выбора опции) следующий:

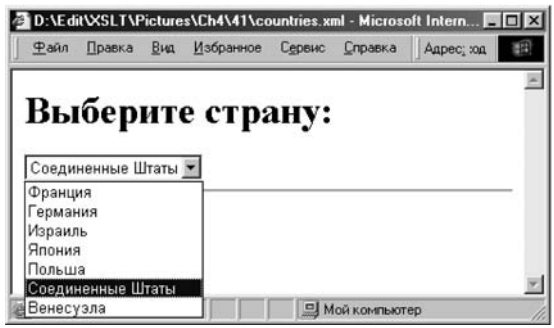


***Рис. 4.1.** Вывод в окне броузера с открытым элементом выбора опции*

В следующем примере выводится элемент <перевод> с одним из атрибутов – код или код-причины – в зависимости от значения переменной $версия-схемы. Логика такого типа может быть полезна в приложении, которое должно обрабатывать различные версии схемы выходного документа.

Пример: Выбор имени атрибута во время выполнения

Исходный документ

Этот пример работает с любым исходным файлом.

Таблица стилей

Таблицу стилей можно найти в файле `conditional.xsl`.

Таблица стилей объявляет глобальный параметр «версия-схемы», который управляет именем атрибута, используемого в выходном файле.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="версия-схемы" select="4.0"/>
<xsl:template match="/">
<перевод>
   <xsl:variable name="attname">
       <xsl:choose>
       <xsl:when test="$версия-схемы &lt; 3.0">код</xsl:when>
       <xsl:otherwise>код-причины</xsl:otherwise>
       </xsl:choose>
   </xsl:variable>
   <xsl:attribute name="{$attname}">17</xsl:attribute>
</перевод>
</xsl:template>
</xsl:stylesheet>
```

Вывод

Со значением по умолчанию параметра «версия-схемы» вывод будет следующим:

```
<перевод код-причины="17"/>
```

При выполнении с параметром «версия-схемы», установленным в значение 2.0, вывод такой:

```
<перевод код="17"/>
```

В этом примере с тем же успехом можно было бы, конечно, использовать `<xsl:attribute>` с фиксированным именем атрибута в обоих ветках элемента `<xsl:choose>`; преимущество показанного выше способа стало бы очевидным, если бы тело шаблона инструкции `<xsl:attribute>` было сложнее, чем просто константа «17».

Это естественным образом приводит к следующему примеру, где используется `<xsl:attribute>`, так как значение требует вычисления. В этом примере значением выводимого атрибута является список разделенных пробельными символами атрибутов `id` дочерних элементов `<предмет>` текущего узла:

```
<корзина>
   <xsl:attribute name="стоимость">
      <xsl:for-each select="предмет">
         <xsl:value-of select="@id"/>
         <xsl:if test="not(position()=last())">
            <xsl:text> </xsl:text>
         </xsl:if>
      </xsl:for-each>
```

```
    </xsl:attribute>
</корзина>
```

См. также

<xsl:element> на стр. 237

<xsl:copy> на стр. 217

<xsl:copy-of> на стр. 221

xsl:attribute-set

Элемент <xsl:attribute-set> – XSLT-элемент верхнего уровня, используемый для определения именованных наборов имен и значений атрибутов. Результирующий набор атрибутов может применяться целиком к любому элементу вывода, обеспечивая способ определения в одном месте часто используемых наборов атрибутов.

Именованные наборы атрибутов предоставляют возможности, подобные возможностям именованных стилей в CSS.

Определен в

XSLT, раздел 7.1.4

Формат

```
<xsl:attribute-set name=ПолноеИмя use-attribute-sets=СписокПолныхИмен>
      <xsl:attribute> *
</xsl:attribute-set>
```

Расположение

<xsl:attribute-set> – элемент верхнего уровня, поэтому он всегда должен быть непосредственным потомком элемента <xsl:stylesheet>.

Атрибуты

Имя	Значение	Назначение
name обязательный	Полное имя	Имя набора атрибутов
use-attribute-sets необязательный	Список полных имен, разделенных пробельными символами	Имена других наборов атрибутов, включаемых в этот набор атрибутов

Содержимое

Нуль или более элементов <xsl:attribute>.

Действие

Атрибут name обязателен и определяет имя набора атрибутов. Он должен быть полным именем, то есть именем с префиксом пространства имен или без него. Если имя использует префикс, этот префикс должен ссылаться на объявление пространства имен, которое действует в этом месте таблицы стилей, и, как обычно, при сопоставлении имен учитывается не префикс, а URI пространства имен. Имена необязательно должны быть уникальными: если имеется несколько наборов атрибутов с одинаковым именем, они объединяются.

Атрибут use-attribute-sets необязателен. Он используется для формирования одного набора атрибутов из нескольких других. Если он есть, его значением должен быть список разделенных пробельными символами лексем, каждая из которых – действительное полное имя, ссылающееся на другой именованный набор атрибутов в таблице стилей. Например:

```
<xsl:attribute-set name="ячейка-таблицы"
   use-attribute-sets="мелкий-шрифт серый-фон центрирование"/>
<xsl:attribute-set name="мелкий-шрифт">
   <xsl:attribute name="font-name">Verdana</xsl:attribute>
   <xsl:attribute name="font-size">6pt</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="серый-фон">
   <xsl:attribute name="bgcolor">#xBBBBBB</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="центрирование">
   <xsl:attribute name="align">center</xsl:attribute>
</xsl:attribute-set>
```

Ссылки не должны зацикливаться: если A ссылается на B, то B не должно прямо или косвенно ссылаться на A. Порядок важен: задание списка именованных наборов атрибутов эквивалентно копированию элементов <xsl:attribute>, которые они содержат, по порядку, в **начало** списка элементов <xsl:attribute>, содержащихся в данном элементе <xsl:attribute-set>.

Если несколько наборов атрибутов имеют одинаковое имя, они объединяются. Если это объединение содержит два атрибута с одинаковым именем, то приоритет будет иметь атрибут с более высоким преимуществом импортирования. Преимущество импортирования обсуждается в разделе <xsl:import> на стр. 256. Если они оба имеют равное преимущество, XSLT-процессор имеет выбор: использовать набор атрибутов, который в таблице стилей встречается позже, или сообщить об ошибке. (Заметьте, что в общем случае невозможно обнаружить эту ошибку в ходе компиляции, потому что имена атрибутов могут вычисляться во время выполнения с использованием шаблона значений атрибутов.)

Порядок, в котором происходит этот процесс объединения, может воздействовать на результат. Когда в элементе <xsl:attribute-set> или <xsl:copy> используется атрибут use-attribute-sets или xsl:use-attribute-sets появляется

в конечном литеральном элементе, он раскрывается, создавая эквивалентную последовательность инструкций <xsl:attribute>. Точные правила довольно витиеваты, даже в исправленной версии, которая опубликована в списке обнаруженных в XSLT 1.0 опечаток. Их лучше проиллюстрировать примером.

Предположим, что имеется следующее определение набора атрибутов:

```
<xsl:attribute-set name="B" use-attribute-sets="A1 A2">
  <xsl:attribute name="p">percy</xsl:attribute>
  <xsl:attribute name="q">queenie</xsl:attribute>
  <xsl:attribute name="r">rory</xsl:attribute>
</xsl:attribute-set>
```

Если есть более одного набора атрибутов с именем A1, сначала они должны быть объединены (с учетом преимущества импортирования), а затем объединенное содержимое помещено в B.

Затем тот же самый процесс применяется к A2. Упомянутые наборы атрибутов раскрываются по порядку. Атрибуты, получаемые при раскрытии A1, выводятся раньше атрибутов из A2. Это означает, что при наличии конфликта приоритет атрибутов из A2 будет выше приоритета атрибутов из A1.

В свою очередь, новый набор атрибутов B должен быть полностью раскрыт, прежде чем он объединится с любым другим набором атрибутов, называемым B. То есть процессор должен заменить ссылки на наборы атрибутов A1 и A2 эквивалентным списком инструкций <xsl:attribute>, прежде чем он объединит этот B с другими наборами атрибутов с таким же именем.

После раскрытия B атрибуты, полученные из A1 и A2, будут выводиться раньше, чем атрибуты p, q и r, поэтому если при раскрытии A1 и A2 генерируют атрибуты с именами p, q и r, они будут переписаны значениями, заданными в B (percy, queenie и rory).

Обычно повторяющиеся имена атрибутов или наборов атрибутов не вызывают ошибку: если несколько атрибутов имеют одинаковое имя, тот из них, который встречается последним (в порядке, произведенном правилами объединения, которые даны выше), будет иметь больший приоритет. Существует только одна ситуация, которая определена как ошибка: если есть два разных набора атрибутов с одинаковыми именами и равным преимуществом импортирования, производящие атрибуты с одинаковым именем. Можно сказать иначе: если конечный результат зависит от относительного положения определений наборов атрибутов в таблице стилей, это является ошибкой. Однако процессору позволено продолжать выполнение после возникновения этой ситуации, рассматривая определения в порядке таблицы стилей, и в данном конкретном случае это настолько проще, чем искать ошибку, что есть подозрение, что большинство процессоров, вероятно, будет поступать именно так.

Использование

Наиболее общим применением наборов атрибутов является формирование пакетов атрибутов, которые составляют стиль отображения, например, совокупностей атрибутов для шрифтов или для таблиц.

Именованный набор атрибутов используется путем ссылки на него в атрибуте use-attribute-sets элементов <xsl:element> или <xsl:copy>, или в атрибуте xsl:use-attribute-sets конечного литерального элемента, или, конечно, в атрибуте use-attribute-sets другого элемента <xsl:attribute-set>. В первых трех случаях в текущее назначение вывода записывается узел элемента и в него добавляются атрибуты из данного набора атрибутов. Любые атрибуты, добавляемые неявным образом из именованного набора атрибутов, могут заменяться узлами атрибутов, добавленными явно, при помощи написания соответствующего кода.

Набор атрибутов – это не просто текстовый макрос. Каждый из атрибутов, содержащихся в наборе атрибутов, имеет тело шаблона для определения значения, и хотя это часто простой текстовый узел, там также могут, например, объявляться переменные или вызываться другие инструкции XSLT типа <xsl:call-template> и <xsl:apply-templates>.

Правила для области действия переменных, описанные в разделе <xsl:variable> на стр. 333, такие же, как и в других случаях, и зависят от позиции определений в исходном документе таблицы стилей. Это означает, что единственный способ параметризации значений атрибутов в именованном наборе атрибутов – ссылки на глобальные переменные и параметры: никакого другого способа передачи параметров набору атрибутов нет. Однако значение сгенерированных атрибутов может зависеть от контекста исходного документа. Контекст не изменяется, когда используется набор атрибутов, поэтому текущий узел («.») и список текущих узлов – те же самые, что и в вызывающем шаблоне.

Это показано в следующем примере.

Пример: Использование набора атрибутов для нумерования

Предположим, что требуется скопировать файл XML, содержащий стихотворение, но так, чтобы в выводе стихотворения элементы <строка> в пределах строфы отображались в форме <строка номер="3" из="18">.

Исходный файл

Исходный файл стихотворение.xml имеет следующую структуру (показана только первая строфа):

```
<?xml version="1.0"?>
<стихотворение>
<автор>Руперт Брук</автор>
<дата>1912</дата>
```

```
<заголовок>Song</заголовок>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
</стихотворение>
```

Таблица стилей

Таблица стилей `нумеровать-строки.xsl` копирует все без изменения, кроме элементов `<строка>`, которые копируются с именованным набором атрибутов:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:strip-space elements="*"/>
<xsl:output method="xml" indent="yes"/>

<xsl:template match="*">
   <xsl:copy>
   <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>

<xsl:template match="строка">
   <xsl:copy use-attribute-sets="последовательность">
   <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>

<xsl:attribute-set name="последовательность">
   <xsl:attribute name="номер"><xsl:value-of
               select="position()"/></xsl:attribute>
   <xsl:attribute name="из"><xsl:value-of
               select="last()"/></xsl:attribute>
</xsl:attribute-set>
</xsl:transform>
```

Вывод

Вывод (снова показана только первая строфа) выглядит так:

```
<стихотворение>
   <автор>Руперт Брук</автор>
   <дата>1912</дата>
   <заголовок>Song</заголовок>
   <строфа>
      <строка номер="1" из="4">And suddenly the wind comes soft,</строка>
      <строка номер="2" из="4">And Spring is here again;</строка>
      <строка номер="3" из="4">And the hawthorn quickens with buds
                              of green</строка>
      <строка номер="4" из="4">And my heart with buds of pain.</строка>
   </строфа>
</стихотворение>
```

Примеры

В следующем примере определен набор атрибутов для генерируемых HTML-элементов <table>:

```
<xsl:attribute-set name="таблица-максимальной-ширины">
   <xsl:attribute name="border">1</xsl:attribute>
   <xsl:attribute name="cellpadding">3</xsl:attribute>
   <xsl:attribute name="cellspacing">0</xsl:attribute>
   <xsl:attribute name="width">100%</xsl:attribute>
</xsl:attribute-set>
```

Этот набор атрибутов можно использовать при генерировании выводимых элементов таким образом:

```
<table xsl:use-attribute-sets="таблица-максимальной-ширины">
<tr></tr>
</table>
```

Это производит следующий вывод:

```
<table border="1" cellpadding="3" cellspacing="0" width="100%">
<tr></tr>
</table>
```

Альтернативно можно использовать набор атрибутов, заменяя некоторые из его определений и добавляя другие, например:

```
<table border="2" rules="cols" xsl:use-attribute-sets="таблица-максимальной-ширины">
<tr></tr>
</table>
```

Вывод теперь становится:

```
<table border="2" rules="cols" cellpadding="3" cellspacing="0" width="100%">
<tr></tr>
</table>
```

Если эта комбинация атрибутов используется неоднократно, ее можно определить в виде набора атрибутов:

```
<xsl:attribute-set name="графленая-таблица"
                   use-attribute-set="таблица-максимальной-ширины">
   <xsl:attribute name="border">2</xsl:attribute>
   <xsl:attribute name="rules">cols</xsl:attribute>
</xsl:attribute-set>
```

Теперь этот новый набор атрибутов также можно вызывать по имени из конечного литерального элемента, инструкции <xsl:element> или инструкции <xsl:copy>.

См. также

<xsl:element> на стр. 237

<xsl:copy> на стр. 217

Раздел «Конечные литеральные элементы» главы 3

xsl:call-template

Инструкция <xsl:call-template> используется для вызова именованных шаблонов. Ее действие аналогично вызову процедуры или подпрограммы в других языках программирования.

Определен в

XSLT, раздел 6

Формат

```
<xsl:call-template name=ПолноеИмя>
   <xsl:with-param> *
</xsl:call-template>
```

Расположение

<xsl:call-template> – инструкция; и всегда используется внутри тела шаблона.

Атрибуты

Имя	Значение	Назначение
name обязательный	Полное имя	Имя шаблона, который будет вызываться

Содержимое

Нуль или больше элементов <xsl:with-param>.

Действие

Последующие разделы описывают правила для имен шаблонов, правила передачи параметров вызываемому шаблону и воздействие этого на контекст.

Имя шаблона

Обязательный атрибут name должен быть полным именем, и он должен соответствовать атрибуту name элемента <xsl:template> в таблице стилей. Если имя имеет префикс пространства имен, имена сравниваются обычным способом, используя соответствующие URI пространств имен. Если префикса нет, то URI пространства имен пустое (пространство имен по умолчанию не используется). Если нет элемента <xsl:template> с соответствующим именем, это ошибка.

Если в таблице стилей присутствует более одного элемента <xsl:template> с соответствующим именем, они должны иметь разное **преимущество импортирования**, при этом будет использоваться элемент с самым высоким преимуществом импортирования. Подробнее о преимуществе импортирования см. в разделе <xsl:import> на стр. 256.

Имя вызываемого шаблона должно быть явно указано в атрибуте name. Нет никаких способов записи этого имени в виде переменной или выражения для вычисления его значения во время выполнения. Если во время выполнения требуется принимать решение, какой из нескольких именованных шаблонов вызывать, единственный способ для этого – применить инструкцию <xsl:choose>.

Параметры

Если имя дочернего элемента <xsl:with-param> соответствует имени элемента <xsl:param> в вызываемом <xsl:template>, то элемент <xsl:with-param> вычисляется (таким же образом, как элемент <xsl:variable>), а его значение присваивается соответствующему параметру <xsl:param> внутри данного именованного шаблона.

Если имеется дочерний элемент <xsl:with-param>, который не соответствует никаким именам элементов <xsl:param> в вызываемом <xsl:template>, то он игнорируется. Это не рассматривается как ошибка.

Если в вызываемом <xsl:template> присутствует элемент <xsl:param>, не соответствующий ни одному элементу <xsl:with-param> в элементе <xsl:call-template>, то переменной <xsl:param> присваивается значение по умолчанию. Подробнее см. раздел <xsl:param> на стр. 316. Это тоже не ошибка.

Контекст

Вызываемый <xsl:template> выполняется без изменения контекста; он использует тот же самый текущий узел, текущий список узлов и то же назначение вывода, что и вызываемый шаблон. Не меняется также и *текущее шаблонное правило* (концепция, которая используется только инструкцией <xsl:apply-imports>, описана на стр. 174).

Использование и примеры

Элемент <xsl:call-template> подобен вызову подпрограммы в традиционных языках программирования, а параметры ведут себя так же, как традиционные передаваемые по значению параметры вызова. Этот элемент полезен при необходимости вызова из различных мест таблицы стилей часто используемого кода.

Назначение вывода

Не существует прямого пути получения результата вызова <xsl:call-template>. Однако если <xsl:call-template> вызван из элемента <xsl:variable>,

эта переменная становится текущим назначением вывода, поэтому любые данные, произведенные вызванным шаблоном (например, с использованием <xsl:value-of>), станут частью значения этой переменной. Значением переменной всегда будет набор узлов, содержащий временное дерево, но на практике она, вероятно, будет чаще использоваться просто как строка.

Например, следующий шаблон выводит переданную строку, заключенную в круглые скобки:

```
<xsl:template name="заключить-в-скобки">
   <xsl:param name="строка"/>
   <xsl:value-of select="concat('(',$строка,')')"/>
</xsl:template>
```

Этот шаблон может быть вызван следующим образом:

```
<xsl:variable name="кредит-в-скобках">
   <xsl:call-template name="заключить-в-скобки">
      <xsl:with-param name="строка" select="@кредит"/>
   </xsl:call-template>
</xsl:variable>
```

Если значение атрибута кредит – «120.00», результирующее значение переменной «$кредит-в-скобках» будет «(120.00)». Строго говоря, это будет временное дерево, состоящее из корневого узла, который имеет единственный текстовый узел, а содержимым этого текстового узла будет «(120.00)»; но для любых практических целей это значение можно использовать так, как если бы это была просто строка.

Смена текущего узла

Для того чтобы использовать <xsl:call-template> для обработки узла, который не является текущим узлом, проще всего вложить <xsl:call-template> внутрь элемента <xsl:for-each>. Есть и другой способ: дать целевому шаблону отдельное имя режима и вызывать его, используя <xsl:apply-templates> с указанием режима.

Например, предположим, что написан шаблон, который выводит глубину текущего узла (число предков, которых он имеет). Шаблону дано уникальное имя и идентичное имя режима:

```
<xsl:template name="глубина-узла" mode="глубина-узла" match="node()">
   <xsl:value-of select="count(ancestor::node())"/>
</xsl:template>
```

Теперь предположим, что нужно найти глубину не текущего узла – скажем, глубину следующего узла в порядке появления в документе, который необязательно находится на том же уровне, что и текущий узел. Можно вызвать этот шаблон любым из двух способов.

Используя <xsl:call-template>:

```
<xsl:variable name="глубина-следующего">
   <xsl:for-each select="following::node()[1]">
```

```
        <xsl:call-template name="глубина-узла"/>
      </xsl:for-each>
   </xsl:variable>
```

или используя `<xsl:apply-templates>` в специальном режиме:

```
<xsl:variable name="глубина-следующего">
   <xsl:apply-templates select="following::node()[1]" mode="глубина-узла"/>
</xsl:variable>
```

В обоих случаях переменная `$глубина-следующего` будет выдавать значение, которое является глубиной узла, следующего в дереве узлов за текущим узлом. Формально значением будет временное дерево, но поскольку такое дерево может быть беспрепятственно преобразовано в строку или число, если этого требует контекст, то это можно использовать без формальностей в контекстах типа арифметических выражений. Например, можно записать такое условие: `<xsl:if test="$глубина-следующего &gt; 4">`.

Рекурсия: Обработка списка значений

Именованные шаблоны используются иногда для обработки списка значений. Язык XSLT не имеет не только никаких обновляемых переменных, которые есть в традиционных языках программирования, но и никаких обычных циклов *for* или *while*, поскольку эти конструкции могут завершиться, только если имеется управляющая переменная, значение которой изменяется. Вместо этого для обработки списков в XSLT можно использовать рекурсию.

Типичная в этих случаях логика иллюстрируется следующим псевдокодом:

```
функция обработать-список(список L) {
   если (не-пуст(L)) {
      обработать(первый(L));
      обработать(остаток(L));
   }
}
```

То есть функция не делает ничего, если список пуст; в противном случае она обрабатывает первый элемент списка, а затем вызывает себя для обработки списка, содержащего все элементы, кроме первого. В результате каждый пункт списка будет обработан, и функция завершится.

Эта логика применяется к двум основным видам списков: наборам узлов и строкам, содержащим символы-разделители. И для того, и для другого случая здесь будут приведены примеры; а позже, в главах 9 и 10, мы рассмотрим более сложные ситуации.

Пример: Использование рекурсии для обработки набора узлов

Это пример обработки набора узлов. XPath 1.0 предоставляет встроенные функции для подсчета узлов и для суммирования их значений, но в нем

нет функции max() или min(), поэтому приходится просматривать набор узлов, сравнивая значение узла с предыдущим самым большим или самым маленьким. Поскольку использовать переменную, чтобы записывать в нее максимальное или минимальное значение среди рассмотренных узлов, нельзя, следует обратиться к рекурсии.

Концептуально это просто: максимальным значением в наборе чисел является или первое число, или максимальное из всех чисел набора, следующих за первым, – которое из них больше. Для манипуляций с набором узлов воспользуемся предикатами XPath: в частности, «[1]» – чтобы найти первый узел в наборе, и «[position()!=1]» – чтобы найти остальные.

Воспользуемся этим подходом, чтобы найти самую длинную реплику в сцене пьесы.

Исходный файл

Исходный файл scene.xml – сцена из пьесы. Он начинается так:

```
<?xml version="1.0"?>
<SCENE><TITLE>СЦЕНА I. Венеция. Улица.</TITLE>
<STAGEDIR>Входят РОДРИГО и ЯГО</STAGEDIR>

<SPEECH>
<SPEAKER>РОДРИГО</SPEAKER>
<LINE>Ни слова больше. Это низость, Яго.</LINE>
<LINE>Ты деньги брал, а этот случай скрыл.</LINE>
</SPEECH>
<SPEECH>
<SPEAKER>ЯГО</SPEAKER>
<LINE>Я сам не знал. Вы не хотите слушать.</LINE>
<LINE>Об этом я не думал, не гадал.</LINE>
</SPEECH>
 и т.д.
</SCENE>
```

Таблица стилей

Ниже показана таблица стилей самая-длинная-реплика.xsl. Она начинается с определения именованного шаблона «max». Этот шаблон принимает в качестве параметра набор узлов, названный «список».

Первое, что он делает – проверяет, не является ли этот набор пустым (<xsl:when test="$список">). Если нет, он присваивает переменной «$голова» количество элементов <LINE> являющихся непосредственными потомками первого узла списка. Затем шаблон рекурсивно вызывает себя, передавая в качестве параметра все узлы, кроме первого, чтобы определить максимальное значение в остальной части списка. После этого он возвращает либо первое значение, либо максимум из остальной части списка – то из них, которое больше. В том случае, если переданный список был пуст, он возвращает нуль.

Шаблонное правило для корневого узла исходного документа просто вызывает шаблон «max», передавая в качестве параметра список всех элементов <SPEECH>.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template name="max">
<xsl:param name="список"/>
<xsl:choose>
<xsl:when test="$список">
   <xsl:variable name="голова" select="count($список[1]/LINE)"/>
   <xsl:variable name="максимальное-из-остальных">
      <xsl:call-template name="max">
         <xsl:with-param name="список" select="$список[position()!=1]"/>
      </xsl:call-template>
   </xsl:variable>
   <xsl:choose>
   <xsl:when test="$голова &gt; $максимальное-из-остальных">
      <xsl:value-of select="$голова"/>
   </xsl:when>
   <xsl:otherwise>
      <xsl:value-of select="$максимальное-из-остальных"/>
   </xsl:otherwise>
   </xsl:choose>
</xsl:when>
<xsl:otherwise>0</xsl:otherwise>
</xsl:choose>
</xsl:template>
<xsl:template match="/">
Длина самой длинной реплики: <xsl:text/>
   <xsl:call-template name="max">
        <xsl:with-param name="список" select="//SPEECH"/>
   </xsl:call-template>
<xsl:text/> строк.
</xsl:template>
</xsl:transform>
```

Вывод

Выводом является просто сообщение, дающее размер самой длинной реплики в этой сцене:

```
<?xml version="1.0" encoding="utf-8" ?>
Длина самой длинной реплики: 20 строк.
```

Как уже говорилось, существует и другое решение этой задачи, которое может оказаться более подходящим в зависимости от обстоятельств. Оно заключается в сортировке набора узлов и выборе первого или последнего элемента. Это происходит так:

```
<xsl:variable name="самая-длинная-реплика">
   <xsl:for-each select="SPEECH">
```

```
      <xsl:sort select="count(LINE)"/>
      <xsl:if test="position()=last()">
        <xsl:value-of select="count(LINE)"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:variable>
```

В принципе рекурсивное решение должно быть быстрее, потому что в нем каждый узел рассматривается только однажды, в то время как сортировка всех значений требует гораздо больше работы, чем это нужно для нахождения максимального. Тем не менее, на практике это довольно зависимо от того, как эффективно осуществлена рекурсия в конкретном процессоре.

Другой случай, где полезна рекурсия, – обработка списков, представленных в форме строк, содержащих списки лексем. Самая простая форма для обработки – список, элементы которого разделяются пробельными символами, так как при помощи функции `normalize-space()` можно гарантировать, что каждый элемент списка отделен точно одним пробелом.

Пример: Использование рекурсии для обработки разделенных строк

В данном примере будут выведены на печать все строки из `scene.xml`, которые содержат имя героя, появляющегося в данной сцене. Пример такой строки:

```
<LINE>Ни слова больше. Это низость, Яго.</LINE>
```

Для выполнения этого сначала нужно нормализовать строку, чтобы преобразовать пунктуацию в пробелы, а нижний регистр – в верхний регистр. Затем нужно обработать строку слово за словом. Единственный способ сделать это – рекурсия: проверить, является ли первое слово в строке именем действующего лица, затем применить эту же логику к остальной части строки после первого слова.

Исходный файл

Исходный файл `scene.xml` – тот же, что и в предыдущем примере.

Таблица стилей

Таблица стилей `строки-с-именами.xsl` начинается с объявления глобальной переменной, значение которой – набор элементов `<SPEAKER>` из документа:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:variable name="действующие-лица" select="//SPEAKER"/>
```

Дальше идет рекурсивный именованный шаблон. У него есть параметр «строка». Первое, что он делает – обрабатывает эту строку, сначала с помощью функции `translate()` преобразуя строчные буквы в верхний регистр, а пунктуацию – в пробелы, а затем с помощью функций `normalize-space()`

и concat() замещая множественные пробелы одним пробелом и добавляя еще один пробел в конце. (Все эти функции описаны в главе 7.)

Затем, используя функцию substring-before(), шаблон извлекает первое слово в строке. Если это слово присутствует в глобальном наборе узлов «$действующие лица», он возвращает значение «истина». В противном случае он проделывает то же самое с остальной частью строки (используя функцию substring-after()), рекурсивно вызывая себя.

Если достигнут конец строки, шаблон возвращает значение «ложь».

```
<xsl:template name="содержит-имя">
  <xsl:param name="строка"/>
  <xsl:variable name="строка1"
        select="translate($строка,
           'абвгдеёжзиклмнопрстуфхцчшщъыьэюя.,:?!;',
           'АБВГДЕЁЖЗИКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ      ')"/>
  <xsl:variable name="строка2"
     select="concat(normalize-space($строка1), ' ')"/>
  <xsl:variable name="первое" select="substring-before($строка2,' ')"/>
  <xsl:choose>
  <xsl:when test="$первое">
     <xsl:choose>
     <xsl:when test="$действующие-лица[.=$первое]">истина</xsl:when>
     <xsl:otherwise>
      <xsl:variable name="остальные" select="substring-after($строка2,' ')"/>
        <xsl:call-template name="содержит-имя">
           <xsl:with-param name="строка" select="$остальные"/>
        </xsl:call-template>
     </xsl:otherwise>
     </xsl:choose>
  </xsl:when>
  <xsl:otherwise>ложь</xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

После этого идет «основная программа» – шаблонное правило, соответствующее корневому узлу. Оно просто вызывает именованный шаблон для каждого элемента <LINE> в документе, копируя элемент в вывод, если именованный шаблон возвращает значение «истина»:

```
<xsl:template match="/">

<xsl:for-each select="//LINE">
    <xsl:variable name="содержит-имя">
        <xsl:call-template name="содержит-имя">
            <xsl:with-param name="строка" select="."/>
        </xsl:call-template>
    </xsl:variable>
    <xsl:if test="$содержит-имя='истина'">
        <xsl:copy-of select="."/>;
    </xsl:if>
```

```
    </xsl:for-each>
    </xsl:template>
    </xsl:transform>
```

Вывод

В вывод попадают все элементы <LINE>, содержащие имена действующих лиц (<SPEAKER>), занятых в этой сцене:

```
<?xml version="1.0" encoding="UTF-8"?>
<LINE>Ни слова больше. Это низость, Яго.</LINE>;
<LINE>Но выбран он. Я на глазах Отелло</LINE>;
<LINE>Я - Яго, а не мавр, и для себя,</LINE>;
<LINE>Брабанцио! Брабанцио, проснитесь!</LINE>;
<LINE>Брабанцио, проснитесь! Караул!</LINE>;
<LINE>Родриго я.</LINE>;
<LINE>Родриго, ты ответишь мне за все.</LINE>;
<LINE>Итак, где эта девочка, Родриго?</LINE>;
<LINE>Тебе, Родриго, ни о чем таком</LINE>;
<LINE>Я награжу за все тебя, Родриго.</LINE>;
```

Таким образом, единственное, о чем следует позаботиться при обработке строк – это чтобы между каждой парой слов был только один пробел и еще один – в конце (иначе вызов функции substring-before() не сработает, когда в списке останется одно слово). Это достигается с помощью функций normalize-space() и concat(). В данном примере они используются довольно неэффективно, потому что строка нуждается в нормализации только однажды, а не при каждом рекурсивном вызове, но это специально оставлено в таком виде, чтобы не усложнять код примера.

Еще один случай, где необходима рекурсия, – когда что-то простое делается фиксированное число раз. Например, предположим, что нужно создать в таблице четыре пустых ячейки. Следующий код иллюстрирует это, создавая одну ячейку, и затем вызывая себя, чтобы создать остальные, завершая работу, только когда заданное число достигает нуля.

```
<xsl:template name="создать-пустые-ячейки">
    <xsl:param name="число"/>
    <xsl:if test="$число != 0">
        <td> </td>
        <xsl:call-template name="создать-пустые-ячейки">
            <xsl:with-param name="число" select="$число - 1"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>
```

Чтобы сформировать четыре пустых ячейки, вызовите этот код с параметром «число», равным четырем, следующим образом:

```
<xsl:call-template name="создать-пустые-ячейки">
        <xsl:with-param name="число" select="4"/>
    </xsl:call-template>
```

См. также

<xsl:apply-templates> на стр. 178

<xsl:param> на стр. 316

<xsl:with-param> на стр. 383

xsl:choose

Инструкция <xsl:choose> определяет выбор между рядом вариантов.

Если присутствуют два варианта, она служит эквивалентом конструкции if-then-else в других языках; если вариантов больше, чем два, она действует эквивалентно конструкциям switch или select.

Определен в

XSLT, раздел 9.2

Формат

```
<xsl:choose>
   <xsl:when> +
   <xsl:otherwise> ?
</xsl:choose>
```

Расположение

<xsl:choose> – инструкция; и всегда используется в теле шаблона.

Атрибуты

Нет.

Содержимое

Один или более элементов <xsl:when>.

Необязательный элемент <xsl:otherwise>, который должен быть последним, если он есть вообще.

Действие

Элемент <xsl:choose> применяется следующим образом:

- Выбирается первый элемент <xsl:when>, чье *выражение* test истинно. Последующие элементы <xsl:when> игнорируются, независимо от значения *выражения* test.
- Если ни в одном из элементов <xsl:when> *выражение* test не является истинным, выбирается элемент <xsl:otherwise>. Если элемента <xsl:otherwise> нет, не выбирается никакой элемент, и поэтому элемент <xsl:choose> не производит никакого действия.
- Выбранный дочерний элемент (если он есть) выполняется путем применения его тела шаблона в текущем контексте. Таким образом, это равно-

сильно тому, как если бы место инструкции <xsl:choose> занимало соответствующее тело шаблона.

Не определено, вычисляется или нет *выражение* test в элементе <xsl:when>, следующем за выбранным элементом, поэтому если оно вызывает функции расширения, которые имеют побочные эффекты, или если оно содержит ошибки, результат не определен.

Использование

Инструкция <xsl:choose> полезна, когда имеется выбор из двух или более альтернативных образов действий. В этом случае она выполняет функции конструкций if-then-else, а также switch или Select Case, имеющихся в других языках программирования.

Использование <xsl:choose> с единственной инструкцией <xsl:when> и без <xsl:otherwise> допустимо и эквивалентно <xsl:if>. Некоторые предлагают писать все инструкции <xsl:if> таким образом, чтобы не переписывать их позже, если понадобится добавить ветвь else.

Когда <xsl:choose> используется внутри тела элемента <xsl:variable> (или <xsl:param>, или <xsl:with-param>), ее действием является условное присваивание: соответствующей переменной в зависимости от условий присваиваются различные значения. Заметьте, однако, что значением переменной в таких случаях всегда является временное дерево.

Примеры

Следующий пример выводит название штата США по заданному двухбуквенному сокращению. Если сокращение не соответствует никакому штату, то выводится само сокращение.

```
<xsl:choose>
   <xsl:when test="штат='AZ'">Аризона</xsl:when>
   <xsl:when test="штат='CA'">Калифорния</xsl:when>
   <xsl:when test="штат='DC'">федеральный округ Колумбия</xsl:when>
   ......
   <xsl:otherwise><xsl:value-of select="штат"/></xsl:otherwise>
</xsl:choose>
```

Следующий пример объявляет переменную ширина и инициализирует ее значением атрибута ширина текущего узла, если такой атрибут есть, или значением 100.

```
<xsl:variable name="ширина">
   <xsl:choose>
      <xsl:when test="@ширина">
         <xsl:value-of select="@ширина"/>
      </xsl:when>
      <xsl:otherwise>100</xsl:otherwise>
   </xsl:choose>
```

```
</xsl:variable>
```

Заметьте: было бы соблазнительно записать это следующим образом:

```
<!--НЕВЕРНО-->
<xsl:choose>
   <xsl:when test="@ширина">
      <xsl:variable name="ширина">
         <xsl:value-of select="@ширина"/>
      </xsl:variable>
   </xsl:when>
   <xsl:otherwise>
      <xsl:variable name="ширина" select="100"/>
   </xsl:otherwise>
</xsl:choose>
<!--НЕВЕРНО-->
```

Это допустимый фрагмент XSLT, но он не производит требуемого действия. Причина в том, что обе переменные «ширина» имеют область действия, ограниченную содержащим их элементом, поэтому они недоступны вне инструкции <xsl:choose>.

См. также

`<xsl:when>` на стр. 381

`<xsl:otherwise>` на стр. 302

`<xsl:if>` на стр. 253

xsl:comment

Инструкция <xsl:comment> используется для записи комментариев в текущее назначение вывода.

Определен в

XSLT, раздел 7.4

Формат

```
<xsl:comment>
   тело шаблона
</xsl:comment>
```

Расположение

<xsl:comment> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Нет.

Содержимое

Тело шаблона.

Действие

Тело шаблона, содержащееся в инструкции <xsl:comment>, может генерировать только текстовые узлы, это ошибка, если оно генерирует другие узлы, такие как элемент, атрибуты или вложенные комментарии. XSLT-процессор может расценить это как фатальную ошибку или может продолжить работу, игнорируя неуместные узлы и их содержимое.

Комментарий не должен включать последовательность «--» и не должен оканчиваться на «-», потому что эти последовательности недопустимы в комментариях XML. Опять же, XSLT-процессор может продолжить выполнение после такой ошибки, добавляя к комментарию дополнительные пробелы, или он может сообщить об ошибке. Если таблица стилей должна быть переносимой, следует избегать генерации этих последовательностей.

В выводе XML или HTML комментарий выглядит так:

```
<!-- текст комментария -->
```

Использование

Теоретически комментарии не имеют значения для программного обеспечения, которое обрабатывает выходной документ, они предназначены только для читателей. Таким образом, в комментариях полезно делать записи о том, когда и как был сгенерирован документ или можно объяснять значение тегов.

Комментарии могут быть особенно полезны для отладки таблицы стилей. Если каждый элемент <xsl:template> в таблице стилей начинается с инструкции <xsl:comment>, это значительно облегчит анализ пути от вывода к таблице стилей.

Комментарии в HTML-выводе используются для некоторых специальных соглашений разметки, например для окружения динамических сценариев HTML. Цель комментариев здесь – гарантировать, что броузеры, не понимающие сценария, пропустят его, а не отобразят как текст. Ниже показан пример.

Примеры

Следующий пример использует функцию расширения, если она есть, для вывода комментария, содержащего дату создания таблицы стилей. Функции расширения описаны в главе 8. Из-за функции расширения этот пример будет работать не со всеми процессорами.

```
<xsl:if test="function-available(Date:toString)" xmlns:Date="java:java.util.Date">
   <xsl:comment>Сгенерирован:
      <xsl:value-of select="Date:toString()"/>
   </xsl:comment>
</xsl:if>
```

Типичный вывод мог бы быть таким:

```
<!--Сгенерирован: Tue Dec 07 23:38:08 GMT 1999-->
```

Следующий пример выводит фрагмент JavaScript в выходной HTML-файл:

```
<script language="JavaScript">
   <xsl:comment>
      function bk(n) {
         parent.frames['content'].location="chap" + n + ".1.html";
      }
   //</xsl:comment>
</script>
```

Вывод будет выглядеть следующим образом:

```
<script language="JavaScript">
   <!--
      function bk(n) {
         parent.frames['content'].location="chap" + n + ".1.html";
      }
   //-->
</script>
```

В таблице стилей комментарий не может быть записан как комментарий, потому что в этом случае XSLT-процессор проигнорирует его полностью. Комментарии в таблице стилей не копируются в назначение вывода.

См. также

`function-available()` в главе 7.

xsl:copy

Инструкция `<xsl:copy>` копирует текущий узел исходного документа в текущее назначение вывода. Это поверхностное копирование; при этом не копируются никакие потомки или атрибуты текущего узла, а только сам теку-

щий узел и (если это элемент) его пространство имен. Для глубокого копирования следует использовать <xsl:copy-of>, см. стр. 221.

Определен в

XSLT, разделе 7.5

Формат

```
<xsl:copy use-attribute-sets=список-полных-имен>
   тело шаблона
</xsl:copy>
```

Расположение

<xsl:copy> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Имя	Значение	Назначение
use-attribute-sets необязательный	Список полных имен, разделенных пробельными символами	Имена наборов атрибутов, применяемых к генерируемому узлу, если это элемент

Содержимое

Необязательное тело шаблона, которое используется, только если текущий узел – это корневой узел или элемент.

Действие

Действие зависит от типа текущего узла:

Тип текущего узла в исходном документе	Действие
корневой узел	В назначение вывода ничего не записывается (нет необходимости записывать корневой узел в назначение вывода, потому что он создается неявно). Атрибут use-attribute-sets игнорируется. Единственный эффект от вызова <xsl:copy> в том, что применяется тело шаблона.
элемент	Узел элемента добавляется в текущее назначение вывода так же, как при вызове <xsl:element>. Он получает то же имя, что и текущий узел элемента в исходном документе. Узлы пространств имен, связанные с текущим узлом элемента, также копируются. Атрибут use-attribute-sets раскрывается: это должен быть список полных имен, разделенных пробельными символами, которые задают именованные наборы атрибутов в таблице стилей. Атрибуты из этих

Тип текущего узла в исходном документе	Действие
	именованных наборов атрибутов вычисляются в порядке их появления и добавляются к новому узлу элемента. Затем применяется тело шаблона.
текст	В назначение вывода записывается новый текстовый узел с тем же значением, что и текущий текстовый узел в исходном документе. Атрибут `use-attribute-sets` и тело шаблона игнорируются.
атрибут	Узел атрибута добавляется в текущее назначение вывода, как при вызове `<xsl:attribute>`. Он имеет то же имя и значение, что и текущий узел атрибута в исходном документе. Если в назначении вывода в этот момент нет открытого узла элемента, чтобы приписать этот атрибут, выдается сообщение об ошибке. Если открытый узел элемента уже содержит атрибут с тем же именем, новый атрибут перезаписывает его. Атрибут `use-attribute-sets` и тело шаблона игнорируются. Хотя гарантируется, что локальное имя, URI пространства имен и значение выводимого атрибута будут такими же, как в оригинале, префикс может измениться (например, если два атрибута, добавляемые к одному и тому же элементу, имеют одинаковый префикс, относящийся к разным URI пространств имен). Правила, управляющие непротиворечивостью пространств имен узлов дерева, гарантируют, что обладающий узлами элемент в конечном дереве будет иметь узел пространства имен, который сопоставляет некоторый префикс пространства имен с URI пространства имен атрибута, но не уточняется, каким должен быть этот префикс.
инструкция обработки	Узел инструкции обработки добавляется в текущее назначение вывода с тем же именем и значением (целью и данными, по терминологии XML), как у текущего узла инструкции обработки в исходном документе. Атрибут `use-attribute-sets` и тело шаблона игнорируются.
комментарий	Узел комментария добавляется в текущее назначение вывода с тем же содержимым, что и у текущего узла комментария в исходном документе. Атрибут `use-attribute-sets` и тело шаблона игнорируются.
пространство имен	Узел пространства имен копируется в назначение вывода. Новый узел пространства имен будет иметь то же самое имя и значение (то есть тот же самый префикс пространства имен и URI), как в оригинале. Если в адресате вывода в данный момент нет открытого узла элемента, чтобы приписать этот узел пространства имен, выдается сообщение об ошибке: поэтому элемент остается открытым до тех пор, пока не будет добавлен его первый дочерний узел или атрибут, или до завершения работы инструкции, создающей узел элемента. Это означает, что добавление к элементу узла пространства имен после того, как к нему были добавлены атрибуты или потомки, является ошибкой. Также будет ошибкой, если узел элемента уже имеет узел пространства имен с тем же самым префиксом пространства имена, но с другим URI пространства имен, или если имя элемента использует пространство имен по умолчанию, а новый узел

Тип текущего узла в исходном документе	Действие
	пространства имен изменяет URI для пространства имен по умолчанию. Если уже имеется узел пространства имен с тем же самым префиксом и тем же самым URI, дубликат узла пространства имен игнорируется.

Использование

Инструкция <xsl:copy> используется, в основном, когда выполняется преобразование XML в XML, при котором части документа должны оставаться неизменными. Она полезна также, когда исходный XML-документ содержит в себе фрагменты XHTML, например, если внутри текстовых данных в источнике используются простые форматирующие элементы HTML типа <i> и <b>, которые должны копироваться в выходной документ HTML без изменений.

Хотя <xsl:copy> производит поверхностное копирование, нетрудно создать глубокую копию, применяя его рекурсивно. Стандартный способ для этого – написать шаблонное правило, которое эффективно вызывает себя:

```
<xsl:template match="@*|node()" mode="копирование">
   <xsl:copy>
      <xsl:apply-templates select="@*" mode="копирование"/>
      <xsl:apply-templates mode="копирование"/>
   </xsl:copy>
</xsl:template>
```

Это шаблонное правило соответствует любому узлу, кроме узла пространства имен или корневого узла, потому что «@*» соответствует любому узлу атрибута, а образец «node()» является сокращенной записью образца «child::node()» и соответствует любому узлу, который является чьим-либо потомком. При применении этого шаблонного правила к узлу оно копирует этот узел, и, если это узел элемента, то же самое шаблонное правило применяется сначала к его атрибутам, а затем к его потомкам. Предполагается, что нет никакого другого шаблонного правила с режимом mode="копирование", которое имело бы более высокий приоритет.

Более простой способ выполнения глубокого копирования – использование <xsl:copy-of>. Однако рекурсивное использование <xsl:copy> позволяет контролировать, какие узлы должны быть включены в вывод.

Примеры

Следующее шаблонное правило полезно, если исходный документ содержит HTML-подобные таблицы, которые должны быть скопированы прямо в вывод без изменения их структуры.

```
<xsl:template match=" table | tbody | tr | th | td ">
   <xsl:copy>
```

```
        <xsl:for-each select="@*">
           <xsl:copy/>
        </xsl:for-each>
        <xsl:apply-templates/>
     </xsl:copy>
  </xsl:template>
```

В результате его применения все эти элементы вместе с их атрибутами копируются в назначение вывода, но их дочерние элементы обрабатываются с помощью других, соответствующих им шаблонных правил; в случае дочернего элемента, являющегося частью модели таблицы, может подойти и это правило, а для некоторых других элементов могут понадобиться другие шаблоны. Продемонстрированное шаблонное правило можно упростить путем копирования атрибутов с помощью `<xsl:copy-of>`:

```
<xsl:template match=" table | tbody | tr | th | td ">
   <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>
```

Следующее шаблонное правило соответствует любым элементам в исходном документе, принадлежащим пространству имен SVG, и без изменения копирует их в вывод вместе с их атрибутами. Сам узел пространства имен SVG также автоматически включается в дерево вывода. (SVG – это Scalable Vector Graphics, масштабируемая векторная графика. Это стандарт XML, являющийся в настоящее время кандидатом в рекомендации. Его назначение – удовлетворить давнишнюю потребность включения векторной графики в веб-страницы.)

```
<xsl:template match="svg:*" xmlns:svg="http://www.w3.org/2000/svg">
   <xsl:copy>
      <xsl:copy-of select="@*">
      <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>
```

См. также

`<xsl:copy-of>` (ниже).

xsl:copy-of

Основная цель инструкции `<xsl:copy-of>` – копировать набор узлов в текущее назначение вывода. Это глубокое копирование: когда копируется узел, копируются также его потомки.

Определен в

XSLT, раздел 11.3

Формат

```
<xsl:copy-of select=Выражение />
```

Расположение

<xsl:copy-of> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Имя	Значение	Назначение
select обязательный	выражение	Набор узлов или другое значение, которое должно копироваться в назначение вывода

Содержимое

Нет; элемент всегда пуст.

Действие

Если результатом вычисления выражения select является набор узлов, в текущее назначение вывода копируется каждый из узлов этого набора узлов, в порядке появления в документе. Это глубокое копирование: вместе с узлом рекурсивно копируются его пространство имен, его атрибуты и его потомки.

Когда значением выражения select является ссылка на переменную, значение которой – временное дерево, это правило копирует все дерево в назначение вывода. Корневой узел дерева не копируется (потому что дерево может иметь только один корень), но все дочерние узлы корня рекурсивно копируются в порядке их появления вместе с их пространствами имен, их атрибутами и их потомками.

Если копируемое дерево включает текст, который был написан с директивой disable-output-escaping="yes" (в инструкции <xsl:text> или <xsl:value-of>), то эта директива копируется в новое дерево наряду с текстом, к которому она применяется, и она будет исполнена, только когда это дерево будет окончательно сериализовываться.

Если результатом является любой другой тип данных (строковый, численный или логический), <xsl:copy-of> имеет такой же эффект, как <xsl:value-of>. Значение преобразуется в строку, как если бы использовалась функция string(), и она записывается как текстовый узел в текущее назначение вывода.

Использование и примеры

Инструкция <xsl:copy-of> имеет два основных применения: ее можно использовать, когда одни и те же данные требуются в нескольких местах в выходном документе, а также она полезна при копировании поддерева без изменений из входного документа в выходной.

Повторяющиеся фрагменты вывода

Использование <xsl:copy-of> в комбинации с временными деревьями требуется, главным образом, когда нужно записать одно и то же множество узлов в несколько мест в выходном документе. Это может понадобиться, например, для верхних и нижних колонтитулов страницы. Эта конструкция позволяет присвоить требуемый фрагмент вывода переменной, а затем копировать его в конечное назначение вывода столько раз, сколько нужно.

Пример: Использование <xsl:copy-of> для повторяющегося вывода

Исходный файл

Исходный файл футбол.xml содержит результаты ряда футбольных матчей, сыгранных во время финальных игр кубка мира 1998 года.

```
<?xml version="1.0"?>
<результаты группа="A">
<матч>
<дата>10-Jun-98</дата>
<команда очки="2">Бразилия</команда>
<команда очки="1">Шотландия</команда>
</матч>
<матч>
<дата>10-Jun-98</дата>
<команда очки="2">Марокко</команда>
<команда очки="2">Норвегия</команда>
</матч>
<матч>
<дата>16-Jun-98</дата>
<команда очки="1">Шотландия</команда>
<команда очки="1">Норвегия</команда>
</матч>
<матч>
<дата>16-Jun-98</дата>
<команда очки="3">Бразилия</команда>
<команда очки="0">Марокко</команда>
</матч>
<матч>
<дата>23-Jun-98</дата>
<команда очки="1">Бразилия</команда>
<команда очки="2">Норвегия</команда>
</матч>
```

```
<матч>
<дата>23-Jun-98</дата>
<команда очки="0">Шотландия</команда>
<команда очки="3">Марокко</команда>
</матч>
</результаты>
```

Таблица стилей

Таблица стилей содержится в файле `футбол.xsl`. Она формирует заголовок таблицы HTML как глобальную переменную, значением которой является дерево, а затем каждый раз, когда требуется вывести этот заголовок, использует `<xsl:copy-of>`. В данном конкретном случае заголовок постоянный, но если он не изменяется, то мог бы содержать и данные из исходного документа. Если бы он содержал вычисляемые значения, кодирование его таким способом было бы эффективнее, чем регенерация заголовка каждый раз заново.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:variable name="заголовок-таблицы">
    <tr>
        <td><b>Дата</b></td>
        <td><b>Хозяева</b></td>
        <td><b>Гости</b></td>
        <td><b>Счет</b></td>
    </tr>
</xsl:variable>

<xsl:template match="/">
<html><body>
    <h1>Матчи в группе <xsl:value-of select="/*/@группа"/></h1>
    <xsl:for-each select="//матч">
    <h2><xsl:value-of select="concat(команда[1], ' против ', команда[2])"/></h2>
    <table bgcolor="#cccccc" border="1" cellpadding="5">
        <xsl:copy-of select="$заголовок-таблицы"/>
        <tr>
        <td><xsl:value-of select="дата"/> </td>
        <td><xsl:value-of select="команда[1]"/> </td>
        <td><xsl:value-of select="команда[2]"/> </td>
        <td><xsl:value-of
            select="concat(команда[1]/@очки, '-', команда[2]/@очки)"/> </td>
        </tr>
    </table>
    </xsl:for-each>
</body></html>
</xsl:template>

</xsl:stylesheet>
```

Вывод

(Автор приносит извинения фанатам футбола, которые отлично знают, что все эти матчи сыграны во Франции, а не на родном поле какой-либо из этих футбольных команд. Это только пример!)

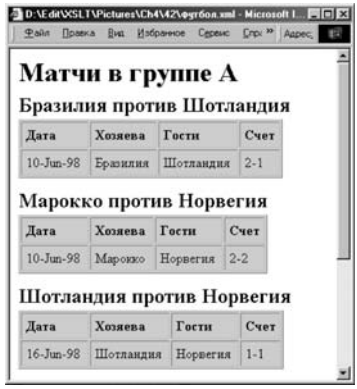


***Рис. 4.2.** Вывод в окне броузера результатов матчей*

В XSLT 1.0 временные деревья (называемые фрагментами конечного дерева; этот термин больше не используется в XSLT 1.1) имеют очень ограниченные функциональные возможности. С ними можно делать только две вещи: можно копировать их в другое дерево, используя `<xsl:copy-of>`, и можно преобразовать их в строки. Некоторые реализации XSLT 1.0 предоставляют функцию расширения для конвертирования фрагмента конечного дерева в набор узлов, что значительно увеличивает полезность фрагментов конечного дерева, потому что это позволяет использовать их как рабочие структуры данных. В XSLT 1.1 эта возможность вошла в стандарт, благодаря чему временное дерево может использоваться везде, где используются наборы узлов. Фактически в XSLT 1.1 временное дерево является набором узлов, поэтому нет необходимости в специальной функции для его конвертирования.

Глубокое копирование

Еще одним применением элемента `<xsl:copy-of>`, которое легко можно упустить из вида из-за способа его описания в спецификации XSLT, является обеспечиваемый им простой способ копирования целого поддерева входного документа прямо в вывод. Поскольку `<xsl:copy-of>` делает глубокую копию, это более просто, чем использование `<xsl:copy>`, но возможно только в том случае, когда все поддерево должно копироваться без изменения. Например, XML-документ, определяющий описание изделия, мог бы иметь элемент, названный `<описание>`, содержимое которого – строгий XHTML. Можно скопировать его в выходной HTML-документ с помощью такого шаблонного правила:

```
<xsl:template match="описание">
   <div>
      <xsl:copy-of select="node()">
   </div>
</xsl:template>
```

В отличие от примеров, использующих <xsl:copy>, здесь нет рекурсивного применения шаблонных правил: каждый дочерний узел элемента <описание> вместе со всеми его потомками копируется в назначение вывода с помощью единственной операции.

Наиболее часто используемый пример применения такой методики – использование <xsl:copy-of select="@*"/> для копирования всех атрибутов текущего элемента. Можно также делать это выборочно – копировать только конкретные атрибуты:

```
<xsl:copy-of select="@название | @высота | @ширина"/>
```

Копирование всех атрибутов, кроме нескольких указанных, несколько труднее. Возможно, лучше всего использовать для этого функцию generate-id(), описанную в главе 7:

```
<xsl:variable name="идентификатор-узла" select="generate-id(@примечание)"/>
<xsl:copy-of select="@*[generate-id() != $идентификатор-узла]"/>
```

См. также

<xsl:copy> на стр. 217

<xsl:variable> на стр. 370

xsl:decimal-format

Элемент <xsl:decimal-format> применяется для определения знаков и символов, используемых при преобразовании чисел в строки с помощью функции format-number().

Заметьте, что <xsl:decimal-format> оказывает влияние только на функцию format-number(). Он никак не влияет на способ, которым <xsl:number> форматирует числа для отображения, или на преобразование чисел в строки по умолчанию, используемое функцией string(), а также на формат, используемый <xsl:value-of> для вывода чисел в виде строк.

Определен в

XSLT, раздел 12.3

Формат

```
<xsl:decimal-format
   name=ПолноеИмя
   decimal-separator=символ
   grouping-separator=символ
   infinity=строка
   minus-sign=символ
   NaN=строка
```

```
percent=символ
per-mille=символ
zero-digit=символ
digit=символ
pattern-separator=символ />
```

Расположение

<xsl:decimal-format> – элемент верхнего уровня. Он может встречаться в таблице сколько угодно раз, но только как непосредственный потомок элемента <xsl:stylesheet>.

Атрибуты

Имя	Значение	Назначение
name необязательный	Полное имя	Имя данного десятичного формата. Если опущено, атрибуты применяются к десятичному формату, используемому по умолчанию.
decimal-separator необязательный	символ	Символ, который нужно использовать для разделения целой и дробной частей числа. Значение по умолчанию – «.».
grouping-separator необязательный	символ	Символ, используемый для разделения групп цифр. Значение по умолчанию – «,».
infinity необязательный	строка	Строка, используемая для представления бесконечности. Значение по умолчанию – «Infinity».
minus-sign необязательный	символ	Символ, используемый в качестве знака «минус». Значение по умолчанию – «-».
NaN необязательный	строка	Строка, используемая для представления не-числа (NaN). Значение по умолчанию – «NaN».
percent необязательный	символ	Символ, используемый для представления знака процента. Значение по умолчанию – «%».
per-mille необязательный	символ	Символ, используемый для представления знака промилле. Значение по умолчанию – «‰».
zero-digit необязательный	символ	Символ, используемый в образцах формата для указания мест, где нужно проставить начальные или конечные нули, даже если это не существенно. Значение по умолчанию – «0».
digit необязательный	символ	Символ, используемый в образцах формата для указания мест, где должна проставляться цифра, если эта цифра значимая. Значение по умолчанию – «#».
pattern-separator необязательный	символ	Символ, используемый в образцах формата, чтобы отделить часть образца для положительных чисел от части образца для отрицательных чисел. Значение по умолчанию – «;».

Содержимое

Нет; элемент всегда пуст.

Действие

Если атрибут name присутствует, элемент <xsl:decimal-format> определяет именованный десятичный формат; если нет – он определяет атрибуты используемого по умолчанию десятичного формата. Именованный десятичный формат используется функцией format-number(), когда она вызывается с тремя параметрами (третий параметр – имя десятичного формата); десятичный формат по умолчанию используется, когда функция format-number() вызывается без третьего параметра.

Считается ошибкой, если имеется более одного элемента <xsl:decimal-format> для десятичного формата, используемого по умолчанию, или для десятичного формата с данным именем, если при этом не все значения их атрибутов совпадают (включая значения по умолчанию). Это будет ошибкой, даже если эти разные элементы <xsl:decimal-format> имеют различное преимущество импортирования.

Элемент <xsl:decimal-format> не определяет непосредственно формат отображения чисел. Он только определяет символы и строки, используемые для представления различных логических знаков. Некоторые из этих логических знаков встречаются в *образце формата*, используемом в качестве параметра функции format-number(), другие – непосредственно в выводимых числах, а третьи используются и там, и там. Реальный формат отображения чисел зависит как от образца формата, так и от выбора символов десятичного формата.

Например, если имеется следующий элемент <xsl:decimal-format>:

```
<xsl:decimal-format name="европейский"
                    decimal-separator=","
                    grouping-separator="." >
```

тогда вызов функции:

```
format-number(1234.5, '#.##0,00', 'европейский')
```

произведет вывод:

```
1.234,50
```

Использование символов «.» и «,» и в образце формата, и в выводе определяется именованным элементом <xsl:decimal-format>, но количество отображаемых разрядов и использование начальных и конечных нулей определяется исключительно образцом формата.

Структура образца формата определена в описании функции format-number() в главе 7. Синтаксис образца формата использует ряд специальных символов: фактически используемые для них символы определены в соответствующем элементе <xsl:decimal-format>. Это следующие символы:

```
decimal-separator
grouping-separator
percent
per-mille
zero-digit
digit
pattern-separator
```

Элемент <xsl:decimal-format> определяет также символы и строки, которые используются, когда это нужно, в выводимом значении. Некоторые из них – те же, что используются в образце формата, другие отличаются от них. Это следующие символы и строки:

```
decimal-separator
grouping-separator
infinity
minus-sign
NaN
percent
per-mille
zero-digit
```

Например, если элемент <xsl:decimal-format> определяет строковое значение бесконечности как «****», то выводом «format-number(1 div 0, $format)» будет «****», независимо от образца формата.

Использование

Элемент <xsl:decimal-format> в сочетании с функцией format-number() применяется для вывода численных данных. Его главное назначение – локализация формата отображаемых чисел для просмотра людьми, но, кроме того, он может быть полезен, когда требуется произвести выходной файл данных, используя, например фиксированное число начальных нулей. Обычно это бывает нужно для чисел из исходных данных или вычисленных из исходных данных, тогда как элемент <xsl:number>, который имеет собственные возможности форматирования, чаще используется для порядковых номеров.

Каждый элемент <xsl:decimal-format> определяет стиль локализованной нумерации, которая служит для поддержки разновидностей форматов, используемых в различных странах и языках, и для других локальных соглашений, например для бухгалтерии, где для отображения отрицательных чисел используются круглые скобки.

Примеры

Следующие таблицы поясняют некоторые результаты, получаемые при использовании элемента <xsl:decimal-format> с различными образцами формата.

Пример 1

Этот десятичный формат принят во многих западноевропейских странах; в нем в качестве десятичной точки используется запятая, а разделителем тысяч служит точка, то есть соглашение, противоположное принятому в Англии и Северной Америке.

В левом столбце показано число, как оно записывается в XSLT. В среднем столбце показан образец формата, передаваемого в качестве второго параметра функции `format-number()`. В правом столбце показана строка, возвращаемая функцией `format-number()`.

Образцы, задействованные в этом примере, используют следующие символы:

- «.» – здесь это разделитель тысяч
- «,» – здесь это десятичная точка
- «#» – это позиция, где может находиться цифра, но где она опускается, если это – незначащий нуль
- «0» – это позиция, где всегда находится цифра, даже если это – незначащий нуль
- «%» – этот знак указывает, что число должно быть выражено как процент
- «;» – этот знак отделяет образец, используемый для положительных чисел, от образца, используемого для отрицательных чисел

```
<xsl:decimal-format decimal-separator="," grouping-separator="."/>
```

Число	Образец формата	Результат
1234.5	#.##0,00	1.234,50
123.456	#.##0,00	123,46
1000000	#.##0,00	1.000.000,00
–59	#.##0,00	–59,00
1 div 0	#.##0,00	Infinity
1234	###0,0###	1234,0
1234.5	###0,0###	1234,5
.00035	###0,0###	0,0004
0.25	#00%	25%
0.736	#00%	74%
1	#00%	100%
–42	#00%	–4200%
–3.12	#,00;(#,00)	(3,12)
–3.12	#,00;#,00CR	3,12CR

Пример 2

Этот пример показывает, как можно использовать другие цифры, а не западные цифры 0-9. Поскольку такие цифры будут незнакомы большинству читателей, в объяснениях вместо них будут использоваться буквы. Этот образец работает очень хорошо, хотя он и не очень полезен:

```
<xsl:decimal-format zero-digit="a" minus-sign="~"/>
```

Число	Образец формата	Результат
10	aa	ba
12.34	##.##	bc.de
−9999999	#,###,###	~j,jjj,jjj

Пример 3

В этом примере показано, как можно отобразить особые числовые значения – не-число и бесконечность – в статистической таблице, например.

```
<xsl:decimal-format NaN="Не применимо" infinity="Значение вне допустимого диапазона"/>
```

Число	Образец формата	Результат
'a'	любой	Не применимо
1 div 0	любой	Значение вне допустимого диапазона

См. также

Функция `format-number()` в главе 7.

`<xsl:number>` на стр. 288.

xsl:document

Инструкция `<xsl:document>` используется для создания нового выходного файла. Это новая инструкция, введенная в рабочем проекте спецификации XSLT 1.1, но ее нестандартные аналоги существовали в нескольких популярных процессорах XSLT 1.0. Эта инструкция позволяет получать в результате преобразования несколько выходных файлов, поэтому можно написать таблицу стилей, которая разобьет большой XML-файл на меньшие XML-файлы или на несколько HTML-файлов, возможно, связанных друг с другом гиперссылками.

Определен в

XSLT 1.1, раздел 16.5

Формат

```
<xsl:document
   href= { uri }
   method= { "xml" | "html" | "text" | ПолноеИмя }
   version= { NMtoken }
   encoding= { строка }
   omit-xml-declaration= { "yes" | "no" }
   standalone= { "yes" | "no" }
   doctype-public= { строка }
   doctype-system= { строка }
   cdata-section-elements= { список-полных-имен }
   indent= { "yes" | "no" }
   media-type= { строка } >
         тело шаблона
</xsl:document>
```

Расположение

<xsl:document> – инструкция, то есть она может находиться в любом месте тела шаблона.

Атрибуты

За исключением атрибута href, атрибуты инструкции <xsl:document> совпадают с атрибутами элемента <xsl:output>, описанного на стр. 303. Эти атрибуты управляют форматированием вторичного выходного документа, созданного с помощью этой инструкции.

Все атрибуты элемента <xsl:document> могут быть записаны как шаблоны значений атрибутов, то есть как XPath-выражения внутри фигурных скобок. Это позволяет передавать значения для таблицы стилей в виде параметров или определять их на основе данных из исходного документа.

Имя	Значение после раскрытия шаблона значения атрибута	Назначение
href обязательный	Относительный или абсолютный URI	Определяет местоположение, куда будет записан выходной документ после сериализации
method необязательный	«xml» | «html» | «text» | полное имя	Определяет требуемый выходной формат
version необязательный	NMtoken	Определяет версию выходного формата
encoding необязательный	строка	Определяет кодировку

Имя	Значение после раскрытия шаблона значения атрибута	Назначение
omit-xml-declaration необязательный	«yes» | «no»	Указывает, должно ли объявление XML включаться в вывод
standalone необязательный	«yes» | «no»	Указывает, что объявление standalone должно быть включено в вывод, и дает его значение
doctype-public необязательный	строка	Указывает открытый идентификатор, который нужно использовать в объявлении DOCTYPE в выходном файле
doctype-system необязательный	строка	Указывает системный идентификатор, который нужно использовать в объявлении DOCTYPE в выходном файле
cdata-section-elements необязательный	Список полных имен, разделенных пробельными символами	Называет те элементы, текстовое содержимое которых должно выводиться в форме секций CDATA
indent необязательный	«yes» | «no»	Указывает, должен ли вывод иметь отступы, отражающие его иерархическую структуру
media-type необязательный	строка	Указывает тип носителя (часто называемый типом MIME), связываемый с выходным файлом

Содержимое

Содержимое элемента <xsl:document> – тело шаблона.

Элемент <xsl:document> может содержать элемент <xsl:fallback>. В этом случае элемент <xsl:fallback> определяет действие, которое должен предпринять процессор XSLT 1.0, когда он сталкивается с инструкцией <xsl:document>. Заметьте, что откат обработки возможен, если только таблица стилей выполняется в режиме совместимости с последующими версиями, то есть если установлен атрибут «version="1.1"» в элементе <xsl:stylesheet> или «xsl:version="1.1"» в любом объемлющем конечном литеральном элементе.

Действие

Когда применяется инструкция <xsl:document>, создается новое конечное дерево, и оно становится текущим назначением вывода для всех выводимых узлов, пока не завершится элемент <xsl:document>.

Обычно это приводит к созданию нового выходного файла. Этот выходной файл является результатом сериализации конечного дерева, и он управляется различными атрибутами элемента <xsl:document> точно так же, как элемент <xsl:output> управляет сериализацией основного выходного документа.

Местоположение нового выходного файла определяется значением атрибута `href`.

- Если это абсолютный URI, файл будет записан так, что он будет доступен по этому абсолютному URI. (На практике это работает, если только XSLT-процессор имеет доступ к данному местоположению по протоколу, используемому в указанном URI. Единственные абсолютные URI, которые гарантированно будут работать, – URI, начинающиеся с «`file:/`», но даже они будут работать, если только правильно установлены разрешения. Тем не менее, в некоторых средах можно задавать URI, начинающиеся, скажем, с «`ftp:`» или «`webdav:`», – удобные, когда преобразование является частью процесса опубликования документа.)
- Если указывается относительный URI, файл будет записан так, что он будет доступен по данному URI относительно URI родительского выходного документа.

Концепция родительского выходного документа более сложна, чем это может показаться. Наиболее общая ситуация – когда таблица стилей генерирует основной выходной документ, возможно, являющийся титульным листом или индексом, и набор вторичных выходных документов, каждый из которых создается, используя инструкцию `<xsl:document>` в точке, где текущим назначением вывода был основной выходной документ. В этом случае родительский выходной документ – всегда основной выходной документ. URI основного выходного документа не устанавливается непосредственно в таблице стилей, обычно он устанавливается в командной строке или через API, используемый для выполнения преобразования. Если основной документ записывается в `file:/c:/results/index.html`, то вторичный документ, созданный с использованием `<xsl:document href="chapter1.html">`, будет записан в `file:/c:/results/chapter1.html`.

Возможно применять инструкцию `<xsl:document>`, когда текущее назначение вывода – другой вторичный документ, созданный с использованием другого элемента `<xsl:document>`. Этим можно воспользоваться при создании структурно сложных HTML-страниц, содержащих страницы разделов, входящих в главы. В этом случае относительный URI в атрибуте `href` интерпретируется относительно файла, созданного активной в настоящий момент инструкцией `<xsl:document>`.

Реальные осложнения возникают при использовании `<xsl:document>` в точке, где текущим назначением не является ни основной выходной документ, ни вторичный выходной документ, созданный с использованием `<xsl:document>`, а временное дерево. Например, при такой записи:

```
<xsl:variable name="дерево">
  <xsl:document href="chap{$nr}.html" method="html">
    <xsl:apply-templates select="содержимое"/>
  </xsl:document>
</xsl:variable>
```

В этой ситуации вторичный выходной документ не записывается сразу; он сохраняется в памяти как дерево, связанное с временным деревом. На самом деле он не является частью временного дерева, поэтому к нему нет доступа из таблицы стилей, но при использовании `<xsl:copy-of>` для копирования временного дерева в другое назначение вывода вторичные документы также переносятся наряду с ним. Если это назначение вывода сериализуется, то вторичные документы также сериализуются в назначение, которое определяется путем интерпретации их атрибутов `href` относительно родительского документа, с которым они теперь связаны.

Благодаря этому можно создать две копии вторичного выходного документа в различных местах файловой системы, используя `<xsl:copy-of>` несколько раз. Наоборот, если переменная дерева никогда не копируется в сериализуемое выходное дерево, то вторичные выходные документы вообще никогда не будут сериализованы. Нужно быть внимательными в этой процедуре, чтобы при каждой сериализации дерево записывалось в другое место, потому что запись двух деревьев в один и тот же URI в ходе одного преобразования является ошибкой.

Использование

Создание множественных выходных файлов часто очень полезно при выполнении преобразований. Типичный случай – когда весомая публикация, например словарь, организована как единый XML-файл, слишком большой для загрузки пользователем, которому нужно посмотреть всего лишь несколько словарных статей. Так что первое, что нужно сделать для подготовки его к использованию людьми, – разбить на небольшие порции: возможно, один документ на каждую букву алфавита или даже один документ на каждую словарную статью. Можно сделать эти порции отдельными HTML-страницами, но лучше проводить преобразование в две стадии: сначала разбить большой XML-документ на множество небольших XML-документов, а затем независимо преобразовать каждый из них в HTML.

Обычная методика – создать один основной выходной файл и целое семейство вторичных выходных файлов. Основной выходной файл может тогда служить индексом. Часто требуется сохранить связи между файлами, чтобы легко можно было снова объединить их (используя функцию `document()`, описанную в главе 7) или чтобы можно было генерировать гиперссылки для удобства пользователей.

Примеры

Эта особенность часто используется для разбиения больших документов на управляемые фрагменты, но для примера вряд ли стоит брать большой документ, удобнее проиллюстрировать все на малом.

Пример: Создание множественных выходных файлов

В этом примере в качестве ввода берется стихотворение, и каждая его строфа выводится в отдельный файл. Более реалистичным примером было бы разбиение книги на главы, но в примере удобнее оперировать небольшими файлами.

Исходный файл

Исходный файл – `стих.xml`. Он начинается так:

```
<стихотворение>
<автор>Rupert Brooke</автор>
<дата>1912</дата>
<заголовок>Song</заголовок>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
<строфа>
<строка>My heart all Winter lay so numb,</строка>
<строка>The earth so dead and frore,</строка>
. . .
```

Таблица стилей

Таблица стилей – `разделение.xsl`.

Требуется начинать новый выходной документ для каждой строфы, поэтому в шаблонном правиле для элемента `<строфа>` используем инструкцию `<xsl:document>`. Ее задача – переназначить весь вывод, произведенный телом шаблона в другой выходной файл. Фактически это очень похоже на действие элемента `<xsl:variable>`, создающего дерево, с той разницей, что здесь дерево не становится временным деревом, а вместо этого сразу сериализуется прямо в выходной файл.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.1">

<xsl:template match="стихотворение">
   <стихотворение>
      <xsl:copy-of select="заголовок"/>
      <xsl:copy-of select="автор"/>
      <xsl:copy-of select="дата"/>
      <xsl:apply-templates select="строфа"/>
   </стихотворение>
</xsl:template>

<xsl:template match="строфа">
   <xsl:variable name="файл" select="concat('строфа', position(), '.xml')"/>
   <строфа номер="{position()}" href="{$файл}"/>
```

```
    <xsl:document href="{$файл}">
        <xsl:copy-of select="."/>
    </xsl:document>
</xsl:template>
</xsl:stylesheet>
```

Вывод

Основной выходной файл содержит нижеприведенную структуру поэмы (разбиение на строки сделано, чтобы было понятнее):

```
<?xml version="1.0" encoding="utf-8" ?>
<стихотворение>
<заголовок>Song</заголовок>
<автор>Rupert Brooke</автор>
<дата>1912</дата>
<строфа номер="1" href="строфа1.xml"/>
<строфа номер="2" href="строфа2.xml"/>
<строфа номер="3" href="строфа3.xml"/>
</стихотворение>
```

Три последующих выходных файла `строфа1.xml`, `строфа2.xml` и `строфа3.xml` создаются в том же каталоге, где основной выходной файл. Ниже приводится `строфа1.xml`:

```
<?xml version="1.0" encoding="utf-8" ?>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
```

Другая версия этого примера, в которой используется функция `element-available()` для проверки, реализована или нет инструкция `<xsl:document>`, и если нет, происходит откат обработки. Этот пример разбирается в разделе `element-available()` главы 7.

См. также

`<xsl:output>` на стр. 303

xsl:element

Инструкция `<xsl:element>` используется для вывода узла элемента в текущее назначение вывода.

Она обеспечивает альтернативу использованию конечного литерального элемента и особенно полезна, когда имя элемента или пространства имен должны вычисляться во время выполнения.

Определен в

XSLT, раздел 7.1.2

Формат

```
<xsl:element name={ПолноеИмя} namespace={uri}
   use-attribute-sets=список-полных-имен  >
      тело шаблона
</xsl:element>
```

Расположение

<xsl:element> используется как инструкция в теле шаблона.

Атрибуты

Имя	Значение	Назначение
name обязательный	Шаблон значения атрибута, возвращающий полное имя	Имя элемента, который должен быть сгенерирован
namespace необязательный	Шаблон значения атрибута, возвращающий URI	URI пространства имен генерируемого элемента
use-attribute-sets необязательный	Список полных имен, разделенных пробельными символами	Список именованных наборов атрибутов, содержащих атрибуты, которые будут добавлены к генерируемому элементу

Содержимое

Тело шаблона.

Действие

Имя генерируемого узла элемента определяется с использованием атрибутов name и namespace.

Атрибуты могут быть добавлены к узлу элемента или с помощью атрибута use-attribute-sets, или записью узлов атрибутов в назначение вывода, используя <xsl:attribute> или <xsl:copy>, или <xsl:copy-of>: это должно быть сделано прежде, чем что-нибудь еще будет записано в назначение вывода. Любые атрибуты, записанные с использованием <xsl:attribute> или <xsl:copy>, или <xsl:copy-of>, перезапишут атрибуты с таким же именем, добавленные с помощью атрибута use-attribute-sets.

Дочерние узлы элемента создаются путем применения внедренного тела шаблона.

Спецификация XSLT написана в терминах записи узлов в конечное дерево. Иногда удобнее представлять это так, что открывающий тег элемента

<xsl:element> формирует открывающий тег в выходном XML-файле, и закрывающий тег элемента <xsl:element> формирует соответствующий закрывающий тег, а в промежутке тело шаблона формирует содержимое выводимого элемента. Однако опасно слишком доверять этой аналогии, потому что запись открывающего и закрывающего тегов – не отдельные действия, которыми можно управлять индивидуально, это просто две вещи, которые происходят вместе в результате применения инструкции <xsl:element>. Более подробно это объясняется в разделе «Конечные литеральные элементы» главы 3.

Оба атрибута – name и namespace – могут задаваться как шаблоны значений атрибутов; то есть они могут содержать выражения, заключенные в фигурные скобки.

Имя элемента

Имя нового элемента получается раскрыванием атрибута name. Результатом раскрывания шаблона значения атрибута должно быть полное имя; то есть действительное XML-имя с необязательным префиксом пространства имен. Например «таблица» или «fo:block». Если есть префикс, он должен соответствовать объявлению пространства имен, которое действует в этом месте таблицы стилей; но это только если нет атрибута namespace, при его наличии считается, что он ссылается на данное пространство имен.

Если имя не является действительным полным именем, XSLT-процессор должен или сообщить об ошибке, или не включать этот узел элемента в генерируемое дерево, включая, тем не менее, его потомков. В силу этого различные процессоры могут по-разному обрабатывать эту ситуацию.

Локальная часть имени выводимого элемента будет всегда той же самой, как локальная часть полного имени, передаваемого как значение атрибута name.

Пространство имен элемента

Как обсуждалось в главе 2, конечное дерево всегда будет соответствовать спецификации XML Namespaces. Иногда, конечно, можно генерировать все выводимые элементы в пространстве имен по умолчанию, но поскольку пространства имен используются все шире, может стать необходимым определять URI пространства имен для имени генерируемого элемента.

Если инструкция <xsl:element> имеет атрибут namespace, он вычисляется (раскрывая шаблон значения атрибута в случае необходимости):

- Если его значение – пустая строка, элемент будет иметь пустой URI пространства имен.
- В остальных случаях значением должен быть URI, задающий пространство имен. Это пространство имен **не** должно обязательно действовать в этом месте таблицы стилей, чаще это будет не так. XSLT-процессор не следит, чтобы значение соответствовало какому-то определенному синтаксису URI, поэтому в действительности можно использовать любую строку. XSLT-процессор выведет любые необходимые узлы пространств

имен, чтобы обеспечить связь имени элемента с данным URI пространства имен в конечном дереве.

- Префикс имени выводимого элемента обычно будет таким же, как префикс указанного полного имени, но XSLT-процессор может при необходимости назначить другой префикс, если он связан с правильным URI. Это может произойти, например, если окажется несколько разных префиксов, связанных с одним и тем же URI пространства имен.

Если нет атрибута namespace:

- Если заданное полное имя включает префикс, это должен быть префикс пространства имен, которое действует в этом месте таблицы стилей: другими словами, должен иметься атрибут xmlns:prefix или непосредственно в инструкции <xsl:element>, или в каком-либо содержащем ее элементе. URI пространства имен в выводе будет URI пространства имен, связанного с этим префиксом в таблице стилей.
- Если полное имя не содержит префикс, используется пространство имен по умолчанию. Это – пространство имен, объявленное с помощью «xmlns="uri"» в каком-либо объемлющем элементе в таблице стилей. Заметьте, что это – один из немногих случаев в XSLT, когда для раскрытия полного имени, не имеющего префикса, берется пространство имен по умолчанию: почти во всех других случаях оставляется пустой URI пространства имен. Это делается для того, чтобы гарантировать согласованность с именем элемента, использованным в открывающем теге конечного литерального элемента.

В сгенерированный узел элемента автоматически войдут все узлы пространств имен, которые требуются для определения префиксов, использованных в его имени и в именах всех его атрибутов, а также копии узлов пространств имен, его родительских элементов в конечном дереве. Эти узлы будут видимы, только когда текущее назначение – временное дерево, которое можно анализировать с помощью XPath-выражений, например: <xsl:for-each select="namespace::*">. Когда назначение вывода сериализуется в файл, узлы пространств имен служат для определения объявлений пространств имен, которые появятся в выводе XML.

Генерирование атрибутов

Если присутствует атрибут use-attribute-sets, он должен содержать список полных имен, разделенных пробельными символами, которые задают именованные элементы <xsl:attribute-set> в таблице стилей. Атрибуты из этих именованных наборов атрибутов применяются в порядке их появления и добавляются к новому узлу элемента. Если в ходе этого процесса обнаруживаются два атрибута с одинаковым именем, то атрибут, добавленный последним, перезаписывает более ранний.

Впоследствии дополнительные узлы атрибутов могут быть добавлены к элементу с использованием инструкции <xsl:attribute>. Часто инструкция <xsl:attribute> является потомком инструкции <xsl:element>, но это не обя-

зательно; она может быть вызвана, например, с помощью <xsl:call-template>. Как только к элементу добавляется узел, не являющийся узлом атрибута (обычно текстовый узел или дочерний узел элемента), дополнительные атрибуты больше не могут добавляться.

> *Смысл этого правила в том, что оно обеспечивает нужную гибкость для генерации вывода в виде XML-файла, исключая необходимость сначала формировать конечное дерево в памяти. Если бы атрибуты могли добавляться в любое время, все конечное дерево должно было бы сохраняться в памяти.*

Снова, если добавляется любой атрибут с таким же именем, как у имеющегося атрибута в узле элемента, новое значение имеет приоритет.

Содержимое элемента

Содержимым нового элемента, то есть узлами его непосредственных и дальнейших потомков, являются узлы, сформированные путем применения тела шаблона, содержащегося в инструкции <xsl:element>.

Использование

В большинстве случаев выводимые элементы могут быть сгенерированы или путем использования конечных литеральных элементов в таблице стилей, или копированием узлов из исходного документа с помощью <xsl:copy>.

Единственно, когда <xsl:element> абсолютно необходим, – это когда тип элемента в выводимом файле не фиксированный и отличается от типа элемента в исходном документе.

Использование <xsl:element> вместо конечного литерального элемента полезно также в случаях, когда используются разные пространства имен. Это позволяет явно определить URI пространства имен генерируемого элемента, а не через ссылку на префикс. Благодаря этому пространство имен может не присутствовать непосредственно в таблице стилей, давая таким образом больший контроль над тем, к какому точно элементу относятся объявления пространства имен.

Пример: Преобразование атрибутов в дочерние элементы

Исходный файл

Исходный документ книга.xml содержит единственный элемент <книга> с несколькими атрибутами:

```
<?xml version="1.0"?>
<книга название="Объектно-ориентированные языки"
   автор="Мишель Бодуэн-Лафон (Michel Beaudouin-Lafon)"
   переводчик="Джек Хаулетт (Jack Howlett)"
   издательство="Chapman &amp;amp; Hall"
   isbn="0 412 55800 9"
   дата="1994"/>
```

Таблица стилей

Таблица стилей `атрибуты-в-элементы.xsl` обрабатывает элемент `<книга>`, по очереди обрабатывая каждый из атрибутов и преобразуя их в элементы.

Таблица стилей такая:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output indent="yes"/>
<xsl:template match="книга">
  <книга>
     <xsl:for-each select="@*">
       <xsl:element name="{name()}">
       <xsl:value-of select="."/>
       </xsl:element>
     </xsl:for-each>
  </книга>
</xsl:template>
</xsl:transform>
```

Выбираются все атрибуты элемента `<книга>` (с помощью выражения «`@*`»), и для каждого генерируется элемент, имя которого такое же, как у исходного атрибута, а содержимым является его значение.

Вывод

Получившийся XML (с использованием Saxon) показан ниже. Фактически эта таблица стилей не обязательно произведет точно такой вывод. Дело в том, что порядок атрибутов – неопределенный. Это означает, что в каждом цикле `<xsl:for-each>` может обрабатывать атрибуты в разном порядке, поэтому порядок дочерних элементов в выводе также непредсказуем.

```
<книга>
   <автор> Мишель Бодуэн-Лафон (Michel Beaudouin-Lafon)</автор>
   <дата>1994</дата>
   <isbn>0 412 55800 9</isbn>
   <издательство>Chapman &amp; Hall</издательство>
   <название>Object-oriented Languages</название>
   <переводчик> Джек Хаулетт (Jack Howlett)</переводчик>
</книга>
```

См. также

`<xsl:attribute>` на стр. 190

`<xsl:copy>` на стр. 217

«Конечные литеральные элементы» в главе 3

xsl:fallback

Инструкция <xsl:fallback> используется для определения действий, которые должны быть произведены, если родительская инструкция не может быть выполнена.

Определен в

XSLT, раздел 15

Формат

```
<xsl:fallback>
   тело шаблона
</xsl:fallback>
```

Расположение

<xsl:fallback> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Нет.

Содержимое

Тело шаблона.

Действие

Инструкция <xsl:fallback> может быть полезна в двух случаях:

- В таблице стилей, которая использует особенности XSLT, определенные в более поздней версии, – для указания действий, которые нужно выполнить, если используется XSLT-процессор, реализующий более раннюю версию стандарта (например, версию 1.0). В частности, в рабочем проекте спецификации XSLT 1.1 введена новая инструкция, <xsl:document>. Если вы желаете использовать инструкцию <xsl:document> в таблице стилей и указать при этом, что должен делать процессор XSLT 1.0, который не понимает эту инструкцию, можно задать предпочтительную реакцию процессора, используя <xsl:fallback>.
- В таблице стилей, которая использует элементы расширения, предоставляемые компанией-разработчиком, пользователем или третьим лицом, – для указания действий, которые нужно выполнить, если таблица стилей используется с XSLT-процессором, не поддерживающим эти расширения.

Если инструкция <xsl:fallback> находится в теле шаблона, которое может быть обработано процессором, она игнорируется наряду с ее содержимым.

XSLT-процессор считает элемент инструкцией, если этот элемент находится в теле шаблона, **и** он находится:

- или в пространстве имен XSLT
- или в пространстве имен, обозначенном как пространство имен расширений путем его включения в:
  - атрибут `extension-element-prefixes` элемента `<xsl:stylesheet>`
  - или атрибут `xsl:extension-element-prefixes` непосредственно данного элемента, или содержащего его конечного литерального элемента, или элемента расширения

Если элемент, считаемый XSLT-процессором инструкцией, известен ему, он применяется. Стандарт не определяет точно, что такое «инструкция, которая известна XSLT-процессору». Обычно это означает, что эта инструкция является специализированным расширением, или расширением, определенным пользователем, которое было установлено или сконфигурировано, согласно инструкциям компании-разработчика.

Если элемент считается инструкцией, но не известен XSLT-процессору, то действия, предпринимаемые процессором XSLT 1.0, могут быть такими:

- Для элемента в пространстве имен XSLT – если действующая версия «1.0», то выводится сообщение об ошибке. Если действующей является другая версия, происходит откат обработки.
- Для элемента расширения – происходит откат обработки.

Процессором XSLT 1.1 неизвестные элементы в пространстве имен XSLT обрабатываются таким способом, только если действующая версия – не «1.0» или «1.1».

Действующая версия – это значение атрибута `xsl:version` самого близкого окружающего конечного литерального элемента, который имеет такой атрибут, или, если таковой отсутствует, атрибута `version` на элементе `<xsl:stylesheet>`. Сравнение является строковым, а не числовым, поэтому номер версии должен быть записан строго как «1.0» или «1.1», а не как «1» или «1.10», например. Идея в том, что в таблице стилей или в части таблицы стилей, которая использует особенности будущих версий XSLT, скажем, 2.1, нужно ставить действующую версию «2.1».

Откат обработки (fallback processing) означает, что если неизвестная инструкция имеет дочерний элемент `<xsl:fallback>`, то применяется инструкция `<xsl:fallback>`; в противном случае выдается сообщение об ошибке. Если имеется несколько инструкций `<xsl:fallback>`, применяются все они.

Инструкция `<xsl:fallback>` касается поведения при откате обработки только инструкций в пределах шаблонов. Элементы верхнего уровня, неопознанные процессором, просто игнорируются. Неопознанный элемент в другом контексте (например, неопознанный потомок инструкций `<xsl:choose>` или `<xsl:call-template>`) вызывает ошибку.

Заметьте, что и атрибут version (или xsl:version), и атрибут extension-element-prefixes (или xsl:extension-element-prefixes) действуют только в пределах модуля таблицы стилей, в котором они встречаются: они не применяются к подключаемым с помощью <xsl:include> или <xsl:import> модулям таблицы стилей.

Использование

Механизм <xsl:fallback> позволяет написать таблицу стилей, которая будет адекватно вести себя с процессорами XSLT, реализующими различные версии XSLT. В большой степени это мотивировано опытом веб-разработчиков с HTML и, особенно, трудностью создания веб-страниц, которые правильно работают в различных броузерах. Поскольку поддержка XSLT броузерами становится широко распространенной, следует позаботиться о том, чтобы создаваемая таблица стилей правильно выполнялась в любом броузере.

Точно так же, весьма вероятно, что каждый поставщик XSLT-процессора (или каждый поставщик броузера) украсит свою реализацию собственными «улучшениями», – на самом деле это уже происходит. Что касается обработки таблицы стилей на стороне сервера – можно использовать патентованные расширения и поэтому обречь себя использованием программ от одной компании-разработчика; но скорее всего, все захотят сделать свои таблицы стилей переносимыми. Механизм <xsl:fallback> позволяет это, определяя для любого патентованного элемента расширения, что должен делать XSLT-процессор, если он не понимает этот элемент. Он может поступать, например так:

- не делать ничего, если процесс несущественен, например подсчет статистики
- использовать альтернативный способ, который производит то же действие
- выдать сообщение о выполнении отката обработки, объясняющее пользователю, что специфическое средство не может быть использовано, и дающее советы по модернизации

Другим способом действий при недоступности какого-либо средства является использование функции element-available() (эта функция описана в главе 7), а также отказ от выполнения соответствующей части таблицы стилей. Эти два механизма частично совпадают, поэтому выбирать можно любой из них.

Примеры

Пример 1: Обработка в режиме совместимости с последующими версиями XSLT

В следующем примере показана таблица стилей, написанная с использованием гипотетического нового средства XSLT версии 6.1. Это средство позволяет вставлять документ, задаваемый по URI, прямо в конечное дерево (это – одно из средств, о которых говорится в списке возможных расширений, из-

данных как приложение к рекомендации по XSLT 1.0). Таблица стилей написана так, чтобы в случае недоступности этого нового средства тот же самый эффект достигался с помощью имеющихся средств.

```
<xsl:template match="заготовка">
   <div id="заготовка" xsl:version="6.1">
      <xsl:copy-to-output href="заготовка.xhtml">
         <xsl:fallback>
            <xsl:copy-of select="document('заготовка.xhtml')"/>
         </xsl:fallback>
      </xsl:copy-to-output>
   </div>
</xsl:template>
```

Пример 2: Переносимая таблица, использующая расширения различных компаний-производителей

Создание таблицы стилей, которая использует расширения компании-производителя, но при этом остается переносимой, – непростая задача, однако существуют механизмы для достижения этого, особенно в случае, когда несколько компаний-производителей предоставляют аналогичные расширения, только слегка различными способами.

Например, несколько процессоров XSLT 1.0 (в их числе xt, Saxon и Xalan) предоставляют средство для генерирования множественных выходных файлов из одной таблицы стилей. С появлением XSLT 1.1 это популярное средство попало в стандарт, но до этого каждый продукт должен был изобретать свой собственный синтаксис. Если требуется написать таблицу стилей, которая использует средство XSLT 1.1, когда оно доступно, но определяет также для вышеназванных процессоров процедуры отката обработки, можно сделать это следующим образом:

```
<xsl:template match="предисловие">
<a href="предисловие.html" xsl:version="1.1"
      xmlns:saxon="http://icl.com/saxon"
      xmlns:xt="http://www.jclark.com/xt"
      xmlns:xalan="com.lotus.xsl.extensions.Redirect"
      xsl:extension-element-prefixes="saxon xt xalan">
  <xsl:document href="предисловие.html">
   <xsl:call-template name="записать-предисловие"/>
   <xsl:fallback>
     <saxon:output file="предисловие.html">
        <xsl:call-template name="записать-предисловие"/>
        <xsl:fallback/>
     </saxon:output>
     <xt:document href="предисловие.html">
        <xsl:call-template name="записать-предисловие"/>
        <xsl:fallback/>
     </xt:document>
     <xalan:write file="предисловие.html">
        <xsl:call-template name="записать-предисловие"/>
```

```
          <xsl:fallback/>
        </xalan:write>
      </xsl:fallback>
    </xsl:document>
  Предисловие</a>
  </xsl:template>
```

Обнадеживает, что такие ужасы исчезнут, когда будет завершена стандартизация XSLT 1.1 и появится множество реализаций этого стандарта. Однако к тому времени поставщики, несомненно, включат в свои продукты новые нестандартные расширения.

Пример 3: Временные деревья

Этот пример фактически не использует <xsl:fallback>; он показывает, как можно достичь того же эффекта другим способом.

Одна из наиболее важных новых особенностей, вводимых в XSLT 1.1, – способность обрабатывать переменную, значением которой является дерево, как набор узлов, используя такие средства, как выражения пути XPath и <xsl:for-each>. Следующий код, который является совершенно законным для XSLT 1.1, процессором XSLT 1.0 будет расцениваться как ошибка:

```
<xsl:variable name="штаты-сша">
  <штат сокр="AZ" название="Аризона"/>
  <штат сокр="CA" название="Калифорния"/>
  <штат сокр="NY" название="Нью-Йорк"/>
</xsl:variable>
. . .
<xsl:value-of select="$штаты-сша/штат[@сокр='CA']/@название"/>
```

Этот пример не использует новые инструкции из XSLT 1.1, поэтому невозможно написать шаблон <xsl:fallback>, который определяет, что должен делать с этим кодом процессор XSLT 1.0: нет подходящей инструкции, в которую можно было бы поместить его. Единственный способ определения действий для отката обработки в этом случае – проверить версию XSLT, используя функцию system-property(). Следующая таблица стилей будет работать как с XSLT 1.0, так и с XSLT 1.1. Она обращается к справочной таблице, определенной в таблице стилей как глобальная переменная. В случае процессора, реализующего версию 1.1, она обращается напрямую к переменной, содержащей поисковую таблицу. А в случае процессора XSLT 1.0 она делает это, используя конструкцию document(''), позволяющую читать таблицу стилей как вторичный входной документ.

```
<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:variable name="штаты-сша">
  <штат сокр="AZ" название="Аризона"/>
  <штат сокр="CA" название="Калифорния"/>
  <штат сокр="NY" название="Нью-Йорк"/>
</xsl:variable>
```

```
<xsl:param name="штат" select="'AZ'"/>

<xsl:variable name="версия-1.0" select="system-property('xsl:version')=1.0"/>

<xsl:variable name="справочная-таблица-1.0"
  select="document('')/*/xsl:variable[@название='штаты-сша']"/>

<xsl:template match="/">
  <xsl:choose>
    <xsl:when test="$версия-1.0">
      <xsl:value-of select="$справочная-таблица-1.0/штат[@сокр=$штат]/@название"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$штаты-сша/штат[@сокр=$штат]/@название"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Здесь для элемента <xsl:stylesheet> использован атрибут «version="1.1"», но фактически можно было использовать и «version="1.0"», потому что данная ситуация, когда синтаксис правилен для обеих версий, но при выполнении в процессоре, поддерживающем версию 1.0, возникнет ошибка времени выполнения, не затрагивается правилами совместимости с последующими версиями. Теоретически процессор XSLT 1.0 мог законно отклонить продемонстрированную выше таблицу стилей во время компиляции, независимо от того, определено в ней «version="1.0"» или «version="1.1"», но, к счастью, процессоры XSLT 1.0, на которых она опробовалась, принимали ее без проблем.

В этом примере для обеспечения отката обработки использована стандартная функция document(), которая работает со всеми процессорами XSLT 1.0. В более сложных примерах это могло бы потребовать использования расширений от компании-разработчика, таких как функция расширения от Microsoft msxml:node-set(). В этом случае для различных процессоров были бы необходимы различные механизмы отката обработки.

См. также

«Расширяемость» в главе 3

«Конечные литеральные элементы» в главе 3

функция element-available() в главе 7

функция system-property() в главе 7

xsl:for-each

Инструкция <xsl:for-each>, используя XPath-выражение, выбирает набор узлов и выполняет одни и те же действия для каждого узла в этом наборе.

Определен в

XSLT, раздел 8

Формат

```
<xsl:for-each select=выражение>
   <xsl:sort> *
   тело шаблона
</xsl:for-each>
```

Расположение

<xsl:for-each> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Имя	Значение	Назначение
select обязательный	Выражение, возвращающее набор узлов	Набор узлов, которые подлежат обработке

Содержимое

Нуль или больше элементов <xsl:sort>, за которыми следует тело шаблона.

Действие

Назначение инструкции <xsl:for-each> – применить тело шаблона, которое она содержит, к каждому узлу из выбранного набора узлов. В следующих разделах описано, как это делается.

Атрибут select

Атрибут select обязателен. Выражение определяет узлы, которые подлежат обработке. Это может быть любое XPath-выражение, соответствующее определению в главе 5, возвращающее набор узлов. Выражение может выбирать узлы относительно текущего узла (узла, обрабатываемого в данный момент) или оно может делать абсолютный отбор от корневого узла, а может просто выбирать узлы путем обращения к переменной, инициализированной ранее. Обращение к переменной, значением которой является дерево, или использование функции document() (описанной в главе 7) позволяет также выбрать корневой узел другого XML-документа.

Тело шаблона, содержащееся в элементе <xsl:for-each>, выполняется для каждого выбранного узла. Внутри этого тела шаблона текущий узел – это обрабатываемый узел (один из выбранных узлов); функция position() возвращает позицию этого узла в порядке обработки (первый обработанный узел имеет position()=1, и так далее), а функция last() возвращает число обрабатываемых узлов.

Часто встречающаяся ошибка состоит в том, что часто забывают о том, что <xsl:for-each> изменяет текущий узел. Например, следующий код, вероятно, не произведет никакого вывода:

```
<xsl:for-each select="para">
  <p><xsl:value-of select="para"/></p>
</xsl:for-each>
```

Почему? Потому что внутри элемента <xsl:for-each> текущим узлом является элемент <para>, так что инструкция <xsl:value-of> должна отобразить другой элемент <para>, который является потомком первого. Вероятно, автор кода имел в виду следующее:

```
<xsl:for-each select="para">
  <p><xsl:value-of select="."/></p>
</xsl:for-each>
```

Сортировка

Если не имеется дочерних элементов <xsl:sort>, выбранные узлы обрабатываются в *порядке появления в документе*. В обычном случае, когда все узлы исходят из одного исходного документа, это означает, что они будут обрабатываться в порядке их появления в первоначальном исходном документе: в частности, узел элемента обрабатывается раньше его потомков. Однако узлы атрибутов, принадлежащие тому же элементу, могут быть обработаны в любом порядке. Если в списке присутствуют узлы из нескольких различных документов, относительный порядок узлов из различных документов не определен (и поэтому может быть разным в разных программах).

Если имеется один или несколько элементов <xsl:sort>, являющихся непосредственными потомками инструкции <xsl:for-each>, узлы сортируются перед обработкой. Каждый элемент <xsl:sort> определяет один ключ сортировки. Подробности об управлении сортировкой см. в разделе <xsl:sort> на стр. 330. Если определено несколько ключей сортировки, они применяются в порядке от наиболее важного к наименее важному. Например, если первый элемент <xsl:sort> определяет сортировку по странам, а второй – по штатам, то узлы будут обрабатываться в порядке штатов в пределах страны. Если два выбранных узла имеют равные ключи сортировки, они будут обрабатываться в порядке появления в документе.

Заметьте, что направление осей, используемых для выбора узлов, несущественно (различные оси и способы их упорядочения описаны в главе 5). Например, выражение «select="preceding-sibling::*"» обработает предшествующие текущему узлу одноуровневые элементы в порядке появления в документе (начиная с первого из них), несмотря на то, что ось preceding-sibling имеет направление, обратное порядку появления в документе. Направление оси воздействует только на значение любых позиционных спецификаторов, используемых в выражении для выбора. Например, выражение «"preceding-sibling::*[1]"» выберет первый предшествующий одноуровневый элемент

по направлению оси, то есть элемент, непосредственно предшествующий текущему узлу, если такой элемент есть.

Если требуется обработать узлы в обратном порядке документа, укажите:

```
<xsl:sort select="position()" order="descending" data-type="number">
```

Использование и примеры

Основная задача инструкции <xsl:for-each> – обход набора узлов в цикле. Кроме того, ее можно использовать просто для смены текущего узла. Оба применения иллюстрируются в следующих разделах.

Обход набора узлов в цикле

Главное использование <xsl:for-each> – обход набора узлов в цикле. По существу, она является альтернативой инструкции <xsl:apply-templates>. Которую из них использовать – в значительной степени дело вкуса. Возможно, <xsl:apply-templates> (*форсированная* обработка) менее строго привязывает таблицу стилей к детальному строению исходного документа и облегчает возможность написания таблицы стилей, более гибко приспосабливаемой к обрабатываемым структурам, в то время как <xsl:for-each> (*извлекающая* обработка) делает логику более ясной для читателя. Она может даже повысить эффективность, потому что исключает необходимость искать шаблонные правила путем сопоставления образцов, хотя эффект, вероятно, невелик.

В следующем примере показана обработка всех атрибутов текущего узла элемента с записью их в виде элементов в конечное дереве. Более подробно этот пример представлен в разделе <xsl:element> на стр. 237.

```
<xsl:template match="книга">
   <книга>
      <xsl:for-each select="@*">
         <xsl:element name="{name()}">
            <xsl:value-of select="."/>
         </xsl:element>
      </xsl:for-each>
   </книга>
</xsl:template>
```

Следующий пример – общий, его можно применить к любому XML-документу.

Пример: Вывод предков узла

Таблицу стилей из следующего примера можно применить к любому XML-документу. Она обрабатывает всех предков каждого узла в порядке, обратном порядку следования в документе (то есть начиная с родительского узла и заканчивая элементом документа), и выводит их имена в комментарий, который показывает позицию текущего узла.

Исходный файл

Эту таблицу стилей можно применить к любому исходному документу.

Таблица стилей

Эта таблица стилей находится в файле `вложенность.xsl`.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="*">
   <xsl:comment>
      <xsl:value-of select="name()"/>
      <xsl:for-each select="ancestor::*">
         <xsl:sort select="position()" order="descending" data-type="number"/>
         <xsl:text> внутри </xsl:text>
         <xsl:value-of select="name()"/>
      </xsl:for-each>
   </xsl:comment>
   <xsl:apply-templates/>
</xsl:template>

</xsl:transform>
```

Вывод

Это могло бы произвести такой вывод:

```
<!--КНИГИ внутри СПИСОК-КНИГ-->
   <!--ПРЕДМЕТ внутри КНИГИ внутри СПИСОК_КНИГ-->
   <!--НАЗВАНИЕ внутри ПРЕДМЕТ внутри КНИГИ внутри СПИСОК-КНИГ-->Число – язык
науки
   <!--АВТОР внутри ПРЕДМЕТ внутри КНИГИ внутри СПИСОК-КНИГ-->Данзиг
   <!--ЦЕНА внутри ПРЕДМЕТ внутри КНИГИ внутри СПИСОК-КНИГ-->5.95
   <!--КОЛИЧЕСТВО внутри ПРЕДМЕТ внутри КНИГИ внутри СПИСОК_КНИГ-->3
```

Смена текущего узла

Другим применением инструкции `<xsl:for-each>` является просто смена текущего узла. Например, если требуется использовать функцию `key()` (описанную в главе 7), чтобы найти узлы в каком-то вспомогательном документе, нужно сначала установить некоторый узел (обычно корень) этого документа в качестве текущего узла, потому что функция `key()` ищет узлы только в том же самом документе, где находится текущий узел.

Например, можно написать:

```
<xsl:variable name="округ">
   <xsl:for-each select="document('код-округа.xml')">
      <xsl:value-of select="key('код-округа', $код)/@название"/>
   </xsl:for-each>
</xsl:variable>
```

Действие данного примера заключается в присвоении переменной значения атрибута название первого элемента, ключ код-округа которого соответствует значению переменной $код.

Здесь инструкция <xsl:for-each> выбирает единственный узел, потому что функция document() при таком использовании возвратит самое большее – один узел. Его даже не надо использовать; единственное, что требуется – сделать его текущим узлом, который влияет на результат функции key().

В таблице стилей, которая обрабатывает несколько входных документов, всегда полезно объявить глобальную переменную:

```
<xsl:variable name="корень" select="/"/>
```

Это позволяет всегда вернуться к первоначальному исходному документу, написав:

```
<xsl:for-each select="$корень">
  . . .
</xsl:for-each>
```

См. также

<xsl:apply-templates> на стр. 178

<xsl:sort> на стр. 330

функция document() в главе 7

функция key()в главе 7

xsl:if

Инструкция <xsl:if> окружает тело шаблона, которое будет применяться, если только заданное условие истинно.

Инструкция <xsl:if> является аналогом оператора *if*, имеющегося во многих языках программирования.

Определен в

XSLT, раздел 9.1

Формат

```
<xsl:if test=выражение >
   тело шаблона
</xsl:if>
```

Расположение

<xsl:if> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Имя	Значение	Назначение
Test обязательный	Выражение	Логическое условие, которое должно проверяться

Содержимое

Тело шаблона.

Действие

Выражение test вычисляется, а результат в случае необходимости преобразуется в логический тип данных с использованием правил, определенных для функции boolean(). Если результат истинен, содержащееся в инструкции тело шаблона применяется; если нет – не предпринимается никаких действий.

Любое значение XPath может быть преобразовано в логический тип. Вкратце, правила такие:

- Если выражение – набор узлов, оно считается истинным, когда этот набор узлов содержит, по крайней мере, один узел. (Это означает, что ссылка на временное дерево всегда считается истиной.)
- Если выражение – строка, оно считается истинным, когда строка не пуста.
- Если выражение – число, оно считается истинным, когда это не нуль.

Использование

Инструкция <xsl:if> полезна, когда действие должно выполняться при выполнении какого-либо условия. Она выполняет функции конструкции if-then, имеющейся в других языках программирования. Если существует два или более альтернативных действий (эквивалентных if-then-else или switch, или Select Case в других языках), вместо <xsl:if> используют <xsl:choose>.

Одним из наиболее общих применений инструкции <xsl:if> является проверка на состояние ошибки. В этом случае она часто используется с <xsl:message>.

Старайтесь избегать использования инструкции <xsl:if> непосредственно внутри <xsl:for-each>. Лучше вместо нее использовать предикат, так как это дает процессору больше возможностей для оптимизации. Например:

```
<xsl:for-each select="para">
  <xsl:if test="@display='yes'">
    <p><xsl:apply-templates/></p>
  </xsl:if>
</xsl:for-each>
```

можно переписать как:

```
<xsl:for-each select="para[@display='yes']">
   <p><xsl:apply-templates/></p>
</xsl:for-each>
```

Примеры

Следующий пример выводит элемент <hr> после обработки последнего из последовательности элементов <para>:

```
<xsl:template match="para">
   <p><xsl:apply-templates/></p>
   <xsl:if test="position()=last()">
      <hr/>
   </xsl:if>
</xsl:template>
```

Следующий пример сообщает об ошибке, если атрибут процент текущего элемента не является числом между 0 и 100. Выражение возвращает истину, если:

- атрибут процент отсутствует
- или значение не может интерпретироваться как число («number(@процент)» является NaN)
- или числовое значение меньше нуля
- или числовое значение больше 100

```
<xsl:if test="not(@процент) or
              (string(number(@процент))='NaN') or
              (number(@процент) &lt; 0) or
              (number(@процент) &gt; 100)">
   <xsl:message>
      атрибут "процент" должен быть числом, находящимся в диапазоне от 0 до 100
      включительно.
   </xsl:message>
</xsl:if>
```

В следующем примере с помощью <xsl:if> форматируется список имен с целью расставить знаки препинания, которые зависят от позиции каждого имени в списке.

Пример: Форматирование списка имен

Исходный файл

Исходный файл авторы.xml содержит единственный элемент <книга> со списком авторов.

```
<?xml version="1.0"?>
<книга>
     <название>Паттерны проектирования</название>
     <автор>Эрих Гамма</автор>
     <автор>Ричард Хелм</автор>
     <автор>Ральф Джонсон</автор>
     <автор>Джон Влиссидес</автор>
</книга>
```

Таблица стилей

Таблица стилей авторы.xsl обрабатывает список авторов, добавляя пунктуацию в зависимости от позиции каждого автора в списке.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="книга">
   <xsl:value-of select="название"/>
    <xsl:for-each select="автор">
      <xsl:value-of select="."/>
      <xsl:if test="position()!=last()-1 and position()!=last()">, </xsl:if>
      <xsl:if test="position()=last()-1"> и </xsl:if>
    </xsl:for-each>
</xsl:template>

</xsl:transform>
```

Вывод

```
Паттерны проектирования
Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес
```

См. также

<xsl:choose> на стр. 213

xsl:import

<xsl:import> – это элемент верхнего уровня, используемый для импорта содержимого одного модуля таблицы стилей в другой. Определения в импортирующем модуле таблицы стилей имеют более высокое *преимущество импортирования*, чем определения в импортируемом модуле, то есть, как правило, предпочтение будет отдаваться им, но детальные правила для каждого типа элементов верхнего уровня различны.

Определен в

XSLT, раздел 2.6.2

Формат

```
<xsl:import href=uri />
```

Расположение

<xsl:import> – элемент верхнего уровня. Это означает, что он должен быть непосредственным потомком элемента <xsl:stylesheet>. Внутри элемента <xsl:stylesheet> дочерние элементы <xsl:import> должны находиться перед любыми другими потомками.

Атрибуты

Имя	Значение	Назначение
href обязательный	URI	URI таблицы стилей, которая должна быть импортирована

Содержимое

Нет; элемент всегда пуст.

Действие

URI, содержащийся в атрибуте href, может быть абсолютным или относительным. Если это относительный URI, то он интерпретируется относительно базового URI документа XML или внешней сущности, содержащей элемент <xsl:import>. Например, если файл main.xsl содержит элемент <xsl:import href="date.xsl"/>, то по умолчанию система будет искать файл date.xsl в том же каталоге, в котором находится main.xsl. В XSLT 1.1 это можно изменить с помощью атрибута xml:base, как описано в разделе «Базовый URI узла» главы 2. Это позволяет указывать для разрешения относительного URI различные базовые URI.

URI должен задавать XML-документ, который является действительным модулем таблицы стилей XSLT. Элементы верхнего уровня этой таблицы стилей логически вставляются во включаемую таблицу стилей в точке, где находится элемент <xsl:import>. Однако:

- Импортируемые элементы верхнего уровня имеют более низкое преимущество импортирования, чем элементы верхнего уровня, определенные в самой импортирующей таблице стилей или во встроенной в нее с помощью <xsl:include> таблице. Это объясняется более подробно ниже.
- Импортируемые элементы сохраняют свой базовый URI, поэтому все, что включает ссылку на относительный URI, использует первоначальный URI импортированной таблицы стилей. Это касается, например, интерпретации последующих элементов <xsl:import> или использования URI в качестве параметра функции document().
- Когда используется префикс пространства имен (обычно в полных именах, но также это относится к автономным префиксам, типа префиксов в атрибуте xsl:exclude-result-prefixes конечного литерального элемента), он интерпретируется только по объявлениям пространств имен первоначального модуля таблицы стилей, в котором находится полное имя. Импортированный модуль таблицы стилей не наследует объявления пространств имен от модуля, который его импортировал. Это включает полные имена, сформированные во время выполнения в результате вычисления выражений, например выражений, которые используются в шаблонах значений атрибутов для атрибутов name или namespace элемента <xsl:element>.
- Значения атрибутов version, extension-element-prefixes и exclude-result-prefixes, которые применяются к элементу во включенной таблице сти-

лей, так же как xml:lang и xml:space, являются значениями, которые определены в элементе <xsl:stylesheet> их собственного модуля таблицы стилей, а не в элементе <xsl:stylesheet> импортирующего модуля.

Импортированный модуль таблицы стилей может использовать синтаксис упрощенной таблицы стилей (*конечный литеральный элемент как таблица стилей*), описанный в главе 3. Это позволяет определить целый модуль как содержимое элемента типа <HTML>. В этом случае он обрабатывается, как если бы это был модуль таблицы стилей, содержащий единственный шаблон, образцом соответствия которого является «/», а содержимым – конечный литеральный элемент.

Импортированный модуль таблицы стилей может содержать инструкции <xsl:include>, чтобы включать последующие модули таблицы стилей, или инструкции <xsl:import>, чтобы импортировать их. Модуль таблицы стилей не должен прямо или косвенно импортировать самого себя.

Импортирование одного и того же модуля таблицы стилей несколько раз, прямо или косвенно, не является ошибкой, но обычно этого не стоит делать. В этом случае одни и те же определения или шаблоны будут иметь различное преимущество импортирования. Это равносильно ситуации, когда импортируются два модуля таблицы стилей с различными именами, но идентичным содержимым.

Преимущество импортирования

Каждый импортированный модуль таблицы стилей имеет определенное преимущество импортирования. Правила такие:

- Преимущество импортированного модуля всегда ниже, чем преимущество модуля, импортирующего его
- Если один модуль импортирует несколько других, то модуль, который он импортирует раньше, имеет более низкое преимущество, чем следующий, и так далее

Это означает, что в структуре, показанной на рис. 4.3, самое высокое преимущество импортирования имеет A, за ним идет C, затем F, B, E и, наконец, D.

Если один модуль таблицы стилей включает другой модуль с помощью <xsl:include>, а не <xsl:import>, то они оба имеют равное преимущество импортирования. Это показано на рис. 4.4.

Здесь J включен в E, поэтому он имеет такое же преимущество импортирования, как E, и аналогично E имеет такое же преимущество импортирования, как C. Если преимущество импортирования выразить числами, чтобы указать порядок (абсолютные величины не имеют значения; единственно важно, чтобы большее число соответствовало более высокому преимуществу), это могло бы выглядеть так:

A	B	C	D	E	F	G	H	J
6	3	5	2	5	1	2	4	5

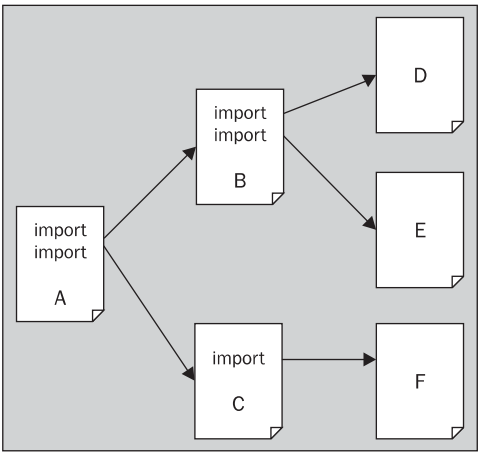


Рис. 4.3. *Преимущество импортирования модулей таблиц стилей (в порядке понижения): A, C, F, B, E, D*

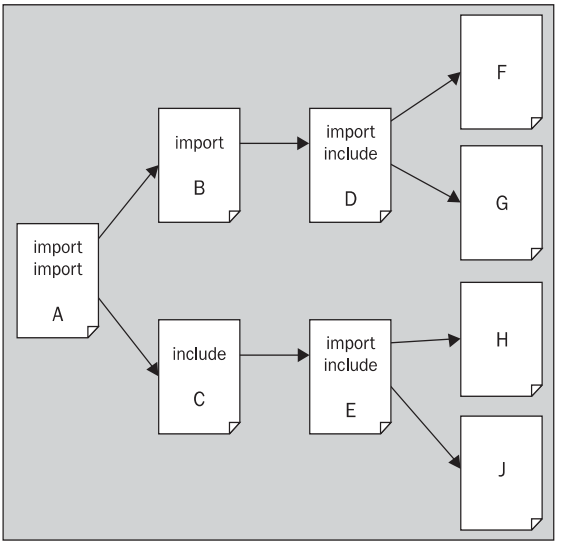


Рис. 4.4. *Включение модулей таблиц стилей с помощью* `<xsl:include>`. *Модули C, E и J (так же как D и G) имеют равные преимущества импортирования*

Преимущество импортирования модуля таблицы стилей относится ко всем элементам верхнего уровня в данном модуле, так, например, элементы `<xsl:template>` в модуле E имеют более высокое преимущество импортирования, чем такие элементы в G.

Поскольку инструкции `<xsl:import>` должны находиться в модуле таблицы стилей перед любыми другими элементами верхнего уровня, смысл этих правил в том, что если бы каждый оператор `<xsl:import>` был заменен содержи-

мым модуля, который он импортирует, элементы верхнего уровня в результирующей объединенной таблице стилей расположились бы по возрастанию преимущества импортирования. Это существенно облегчает жизнь разработчиков, реализующих стандарт. Однако это не означает, что <xsl:import> – дотаточно простой процесс подстановки текста, так как нужно еще различать случаи, когда два объекта (например, шаблонные правила) имеют равное преимущество импортирования, потому что они исходят из одной таблицы стилей или из таблиц стилей, которые были включены, а не импортированы.

Влияние преимущества импортирования

Преимущество импортирования элемента верхнего уровня влияет на его значимость относительно других элементов верхнего уровня того же типа и может использоваться для разрешения конфликтов. Влияние на каждый тип элементов верхнего уровня показано ниже в таблице.

Тип элемента	Правила
<xsl:attribute-set>	Если имеется два набора атрибутов с одинаковым расширенным именем, они объединяются. Если есть атрибут, который присутствует в обоих, то доминирует атрибут из набора атрибутов с более высоким преимуществом импортирования. Будет ошибкой, если в таком случае нет явного победителя (то есть если атрибут имеет несколько значений с равным преимуществом, и оно является самым высоким преимуществом). XSLT-процессор тогда имеет два варианта: сообщить об ошибке или выбрать тот атрибут, который указан последним.
<xsl:decimal-format>	Преимущество импортирования элемента <xsl:decimal-format> не имеет никакого значения. Будет ошибкой включать более одного элемента <xsl:decimal-format> с одинаковым именем (или без имени), если определения не эквивалентны. В этом случае XSLT-процессор должен сообщить об ошибке.
<xsl:import> и <xsl:include>	Никакие конфликты не возникают; преимущество импортирования этих элементов влияет только на определение преимущества импортирования упомянутого модуля таблицы стилей.
<xsl:key>	Используются все определения ключей, независимо от их преимущества импортирования. Подробнее см. раздел <xsl:key> на стр. 272.
<xsl:namespace-alias>	Если для одного и того же префикса таблицы стилей определено несколько псевдонимов, то используется псевдоним с самым высоким преимуществом импортирования. Будет ошибкой, если явного победителя нет. XSLT-процессор тогда имеет два варианта: сообщить об ошибке или выбрать тот псевдоним, который определен последним.
<xsl:output>	Все элементы <xsl:output> в таблице стилей объединяются. В случае атрибута cdata-section-elements элемент выводится в формате CDATA, если он объявлен как таковой в любом из элементов <xsl:output>. В случае других атрибутов, если значение явно при-

Тип элемента	Правила
	сутствует у более чем одного элемента <xsl:output>, то побеждает то из них, которое имеет самое высокое преимущество импортирования. Будет ошибкой, если явного победителя нет. XSLT-процессор тогда имеет два варианта: сообщить об ошибке или выбрать тот атрибут, который был определен последним.
<xsl:strip-space> и <xsl:preserve-space>	Если имеется несколько элементов <xsl:strip-space> или <xsl:preserve-space>, которые соответствуют конкретному имени элемента в исходном документе, то используется элемент с самым высоким преимуществом импортирования. Если после этого все еще остается несколько конкурентов, каждому из них назначается приоритет по правилам, которые используются для образца match в <xsl:template>. В частности, явно указанное полное имя имеет более высокий приоритет, чем форма «prefix:*», которая, в свою очередь, имеет более высокий приоритет, чем «*». После этого используется элемент с самым высоким приоритетом. Будет ошибкой, если после этого остается более одного соответствия (даже если они все дают одинаковый результат). XSLT-процессор тогда имеет два варианта: сообщить об ошибке или выбрать тот элемент, который был определен последним. Если для элемента нет соответствия, узлы пробельных символов сохраняются.
<xsl:template>	При выборе шаблонного правила для использования с <xsl:apply-templates> сначала берутся все шаблонные правила с соответствующим режимом mode. Из них рассматриваются все правила с образцом **match**, соответствующим выбранному узлу. Если после этого остается больше одного, то дальше рассматриваются только правила с самым высоким преимуществом импортирования. Если после этого все еще остается выбор, то выбирается правило с самым высоким приоритетом: о принятии решений по приоритетам говорится в разделе <xsl:template> на стр. 349. Будет ошибкой, если это все еще не даст явного победителя. XSLT-процессор тогда имеет два варианта: сообщить об ошибке или выбрать шаблонное правило, которое было определено последним. (На практике некоторые процессоры выводят предупреждающее сообщение.) При выборе шаблона для использования с <xsl:call-template> рассматриваются все именованные шаблоны с соответствующим именем. Если их несколько, то используется шаблон с самым высоким преимуществом импортирования. Будет ошибкой иметь несколько именованных шаблонов с одинаковым именем и равным преимуществом импортирования; XSLT-процессор должен сообщить об этой ошибке, даже если на эти шаблоны нет ни одной ссылки.
<xsl:variable> и <xsl:param>	При разрешении ссылки на переменную в выражении или образце XSLT-процессор сначала пытается найти соответствующую локальную переменную или определение параметра, то есть то, что

Тип элемента	Правила
	определено в шаблоне. Если он не находит там искомого, он ищет глобальную переменную или параметр – то есть элемент <xsl:variable> верхнего уровня или элемент <xsl:param> с таким же расширенным именем, как у ссылки на переменную. Они могут находиться в любом месте таблицы стилей или в том же самом модуле, или в другом модуле. Кроме того, нет никаких ограничений на опережающие ссылки. Если найдется несколько соответствующих глобальных переменных или параметров, то побеждает имеющий самое высокое преимущество импортирования. Будет ошибкой иметь в таблице стилей более одной глобальной переменной или параметра с одинаковым расширенным именем и равным преимуществом импортирования. Это справедливо, даже если на переменную нигде нет ссылок или если она замаскирована другой переменной с таким же именем, но с более высоким преимуществом импортирования. XSLT-процессор должен сообщить об этой ошибке.

Использование

Правила для <xsl:import> настолько всеобъемлющи, что можно подумать, будто это средство является самым важным при использовании XSLT, как наследование является самым важным при программировании на языке Java. На практике во многих таблицах стилей элемент <xsl:import> вообще не используется, но он почти наверняка понадобится при разработке семейства таблиц стилей для обработки широкого диапазона типов исходных документов.

Подобно наследованию в объектно-ориентированных языках, <xsl:import> предназначен для упрощения создания библиотек или многократно используемых компонентов, только в данном случае эти компоненты – модули таблиц стилей. И механизм действия <xsl:import> очень похож на механизм наследования. Например, можно создать таблицу стилей, которая удобным способом определяет общую цветовую схему как набор глобальных переменных, задающих имена цветов. Другая таблица стилей могла бы служить для создания наборов основных фреймов на данном веб-сайте, обращаясь к этим переменным для определения цвета фона. Далее, если понадобится использовать эту общую структуру, изменив лишь некоторые детали, например один из цветов, поскольку он не гармонирует с изображением на какой-то странице, то можно написать таблицу стилей для этой специфической страницы, переопределяющую только нужный цвет. Это проиллюстрировано на схеме ниже.

Предположим, что модуль таблицы стилей для общих определений цветов выглядит так:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:цвет="http://acme.co.nz/colors"
                version="1.0">
<xsl:variable name="цвет:голубой" select="'#0088ff'"/>
```

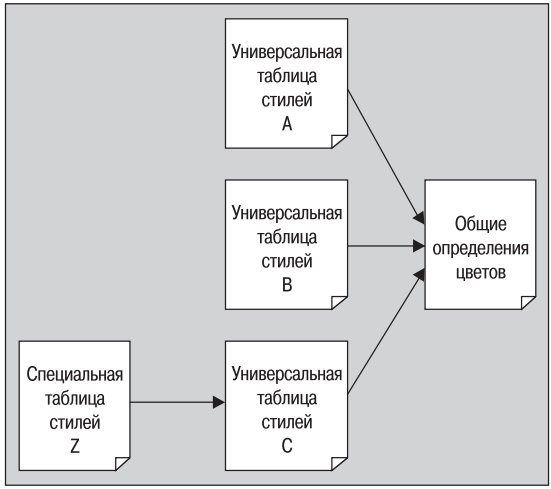


***Рис. 4.5.** Многократное использование модулей таблиц стилей*

```
<xsl:variable name="цвет:розовый" select="'#ff0088'"/>
<xsl:variable name="цвет:сиреневый" select="'#ff00ff'"/>
</xsl:stylesheet>
```

Теперь все универсальные таблицы стилей могут включать эти определения через `<xsl:include>` (нет необходимости в `<xsl:import>`, если определения не изменяются). Это облегчает возможность поддержания общего фирменного стиля, так как все определения содержатся в одном и том же месте.

Однако есть случаи, когда требуется отойти от общего правила, и это можно легко сделать. Если для какого-то документа нужно применить таблицу стилей C, но только изменить в ней используемые цвета, тогда для этого можно определить таблицу стилей Z следующим образом:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:цвет="http://acme.co.nz/colors"
                version="1.0">
<xsl:import href="универсальная-таблица-стилей-C.xsl"/>
<xsl:variable name="цвет:сиреневый" select="'#cc00cc'"/>
</xsl:stylesheet>
```

Фактически это могла бы быть вся таблица стилей (конечно, кроме элемента `<xsl:stylesheet>`). На обычном языке это означает, что стиль Z – то же, что и стиль C, но с другим оттенком сиреневого. Заметьте, что все ссылки на переменную «цвет:сиреневый» интерпретируются как ссылки на определение, данное в Z, даже если ссылки находятся в том же самом модуле таблицы стилей, где находится другое определение «цвет:сиреневый».

Общее правило: чтобы включить стандартное содержимое в таблицу стилей без изменения, используйте `<xsl:include>`. Если есть определения, которые нужно заменить, используйте `<xsl:import>`.

Примеры

Первый пример показывает, какое действие <xsl:import> оказывает на переменные.

Пример 1: Старшинство переменных

Данный пример демонстрирует, как распределяется старшинство глобальных переменных, когда основной модуль таблицы стилей и импортированный модуль объявляют переменные с одинаковым именем.

Исходный файл

Этот пример может быть выполнен с любым исходным XML-файлом.

Таблица стилей

Основной модуль таблицы стилей – переменные.xsl.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:acme="http://acme.com/xslt"
    exclude-result-prefixes="acme">

<xsl:import href="заготовка.xsl"/>
<xsl:output encoding="UTF-8" indent="yes"/>

<xsl:variable name="acme:название-компании" select="'Acme Widgets Limited'"/>

<xsl:template match="/">
<c><xsl:value-of select="$acme:авторское-право"/></c>
</xsl:template>

</xsl:stylesheet>
```

Импортированный модуль таблицы стилей – заготовка.xsl.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:co="http://acme.com/xslt">

<xsl:variable name="co:название-компании" select="'Acme Widgets Incorporated'"/>
<xsl:variable name="co:авторское-право"
              select="concat('Copyright © ', $co:название-компании)"/>

</xsl:stylesheet>
```

Вывод

Вывод этой таблицы стилей будет таким:

```
<?xml version="1.0" encoding="UTF-8" ?>
<c>Copyright © Acme Widgets Limited</c>
```

потому что в объявлении переменной «$co:авторское-право» ссылка на переменную «$co:название-компании» соответствует объявлению этой переменной в основной таблице стилей, так как оно имеет более высокое преимущество импортирования, чем объявление в заготовка.xsl.

То, что в таблицах стилей используются различные префиксы пространств имен, не имеет значения: префикс «acme» в основной таблице стилей соответствует тому же URI пространства имен, что и префикс «co» в заготовка.xsl, так что эти имена эквивалентны.

В этом примере явно указана кодировка: encoding="UTF-8" – как для модулей таблиц стилей, так и для вывода. В большинстве приведенных здесь примеров используются только символы ASCII, а поскольку кодировка символов по умолчанию – UTF-8 – является надмножеством ASCII, все прекрасно работает. Тем не менее, в данном примере использован символ авторских прав «©», который не является символом ASCII, поэтому важно было указать кодировку символов, используемую текстовым редактором автора: iso-8859-1 (в действительности это ее вариант от Microsoft, называемый Windows ANSI, но они достаточно близки и особой разницы нет).[1]

Второй пример показывает, какое влияние <xsl:import> оказывает на шаблонные правила.

Пример 2: Старшинство шаблонных правил

В этом примере показана целиком таблица стилей – стандартный-стиль.xsl, которая преобразовывает стихотворение в HTML, и переопределение одного из правил в импортирующей таблице стилей. Все необходимые файлы находятся в подкаталоге import файла с примерами для этой главы.

Исходный файл

Данный пример работает со стихотворением, использовавшимся в главе 1. В файлах для загрузки это – стих.xml. Файл начинается так:

```
<?xml version="1.0"?>
<стихотворение>
<автор>Руперт Брук</автор>
<дата>1912</дата>
<название>Song</название>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
```

[1] Поскольку пример был переведен на русский язык – кодировка iso-8859-1 была заменена на UTF-8. – *Примеч. науч. ред.*

```
</строфа>
 и т.д.
</стихотворение>
```

Таблица стилей A

Вот таблица стандартный-стиль.xsl:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <head>
   <title><xsl:value-of select="//название"/></title>
   </head>
   <body>
      <xsl:apply-templates/>
   </body>
   </html>
</xsl:template>

<xsl:template match="название">
   <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="автор">
   <div align="right">
      <xsl:apply-templates/>
   </div>
</xsl:template>

<xsl:template match="строфа">
   <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="строка">
   <xsl:apply-templates/><br/>
</xsl:template>

<xsl:template match="дата"/>
</xsl:stylesheet>
```

Вывод A

При выполнении этой таблицы стилей вывод будет следующим (реальная последовательность тегов может быть, конечно, иной, в зависимости от используемого XSLT-процессора):

```
<html>
   <head>
   <title>Song</title>
   </head>
   <body>
      <div align="right">Руперт Брук</div>
      <h1>Song</h1>
```

```
<p>
   And suddenly the wind comes soft,<br>
   And Spring is here again;<br>
   And the hawthorn quickens with buds of green<br>
   And my heart with buds of pain.<br>
</p>
```

Таблица стилей B

Теперь создадим вариант, в котором строки стихотворения будут пронумерованы. Когда нужна такая форма вывода, этот вариант будет действовать как основная таблица стилей. Эта таблица содержится в файле `стиль-с-нумерацией.xsl`:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import href="стандартный-стиль.xsl"/>

<xsl:template match="строка">
   <xsl:number level="any" format="001"/>  
   <xsl:apply-imports/>
</xsl:template>

</xsl:stylesheet>
```

Обратите внимание на использование ссылки на символ «` `» для вывода неразрывного пробела. В HTML для этой цели служит «` `». При желании можно использовать в таблице стилей эту ссылку на сущность (это – просто символическое имя для символа Unicode `#xa0`), но только если она будет объявлена как сущность в DTD. Обычно проще использовать числовую ссылку на символ.

Вывод B

На этот раз вывод начинается следующим образом. Точный формат снова зависит от процессора (например, некоторые процессоры могут вместо «` `» выводить «` `» или «` `», но при отображении в окне броузера это выглядит одинаково.

```
<html>
   <head>
   <title>Song</title>
   </head>
   <body>
      <div align="right">Руперт Брук</div>
      <h1>Song</h1>
      <p>
         001  
         And suddenly the wind comes soft,<br>
         002  
         And Spring is here again;<br>
         003  
         And the hawthorn quickens with buds of green<br>
```

```
        004  
        And my heart with buds of pain.<br>
    </p>
```

Все шаблонные правила, определенные в `стандартный-стиль.xsl`, используются как обычно, кроме случая, когда текущий узел соответствует образцу `«строка»`. В этой ситуации для узла подходят два возможных шаблонных правила, поэтому выбирается то, у которого более высокое преимущество импортирования. В нашем случае это будет правило в импортирующем модуле таблицы стилей, а именно `стиль-с-нумерацией.xsl`. В результате строки поэмы выводятся с порядковым номером, рассчитанным с помощью инструкции `<xsl:number>`, которая описана на стр. 288. Эффект использования `<xsl:apply-imports>` в том, что после того, как выведется номер строки, выводится нормальным способом сама строка с помощью шаблонного правила из таблицы стилей `стандартный-стиль.xsl`.

См. также

`<xsl:include>` в разделе ниже

`<xsl:apply-imports>` на стр. 174

xsl:include

`<xsl:include>` – элемент верхнего уровня, используемый для включения содержимого одного модуля таблицы стилей в другой. Определения в обоих модулях таблиц стилей – во включаемом и во включающем – будут иметь равное *преимущество импортирования*, поэтому эффект такой, как будто эти определения текстуально содержатся во включающем модуле в том месте, где находится элемент `<xsl:include>`.

Определен в

XSLT, раздел 2.6.1

Формат

```
<xsl:include href=uri />
```

Расположение

`<xsl:include>` – элемент верхнего уровня, то есть он должен быть непосредственным потомком элемента `<xsl:stylesheet>`. Нет никаких ограничений на его расположение в таблице стилей относительно других элементов верхнего уровня.

Атрибуты

Имя	Значение	Назначение
href обязательный	URI	URI таблицы стилей, которая будет включена

Содержимое

Нет; элемент всегда пуст.

Действие

URI, содержащийся в атрибуте href, может быть абсолютным или относительным URI. Если это относительный URI, он интерпретируется относительно базового URI документа XML или внешней сущности, содержащей элемент <xsl:include>. Например, если файл main.xsl содержит элемент <xsl:include href="date.xsl"/>, тогда по умолчанию система будет искать date.xsl в том же каталоге, где находится main.xsl. В XSLT 1.1 это можно изменить, используя атрибут xml:base для указания базового URI явным образом, как описано в разделе «Базовый URI узла» главы 2.

URI должен задавать XML-документ, который является действительной таблицей стилей XSLT. Элементы верхнего уровня этой таблицы стилей логически вставляются во включающий модуль таблицы в том месте, где находится элемент <xsl:include>. Однако:

- Эти элементы сохраняют свой базовый URI, поэтому все ссылки на относительный URI разрешаются относительно первоначального URI включенной таблицы стилей. Это правило действует, например, при последующем расширении элементов <xsl:include> и <xsl:import> или при использовании относительных URI в качестве параметров функции document().
- Когда используется префикс пространства имен (обычно в полном имени, но также это относится к автономным префиксам, таким как префиксы в атрибуте xsl:exclude-result-prefixes конечного литерального элемента), он интерпретируется с помощью только объявлений пространств имен в первоначальном модуле таблицы стилей, откуда происходит полное имя. Включенный модуль таблицы стилей не наследует объявления пространств имен от модуля, который включает его. Это относится даже к полному имени, созданному во время выполнения в результате вычисления выражения, например выражения, которое используется в шаблоне значения атрибута для атрибутов name или namespace элемента <xsl:element>.
- Значениями атрибутов version, extension-element-prefixes и exclude-result-prefixes, которые применяются к элементу во включенном модуле таблицы стилей, а также атрибутов xml:lang и xml:space, являются те значения, которые были определены в их собственном элементе <xsl:stylesheet>, а не в элементе <xsl:stylesheet> включающего модуля таблицы стилей.

- Исключение сделано для элементов <xsl:import> во включенном модуле таблицы стилей. Элементы <xsl:import> должны появляться раньше любых других элементов верхнего уровня, поэтому вместо размещения их во включающем модуле в их естественной последовательности они передвигаются, так что они размещаются после всех элементов <xsl:import>, но перед любыми другими элементами верхнего уровня во включающем их модуле таблицы стилей. Это важно в ситуациях, когда имеются повторяющиеся определения, и XSLT-процессору позволено выбирать то из них, которое появляется последним.

Включенный модуль таблицы стилей может использовать упрощенный синтаксис таблиц стилей, описанный в главе 3 (*конечный литеральный элемент как таблица стилей*). Это позволяет целый модуль таблицы стилей определить как содержимое элемента типа <HTML>. Тогда он обрабатывается, как будто это модуль, содержащий единственный шаблон, образцом соответствия которого является «/» и содержимым которого является конечный литеральный элемент.

Включенный модуль таблицы стилей может содержать инструкции <xsl:include> для включения дальнейших таблиц стилей или инструкции <xsl:import> – для их импорта. Таблица стилей не должна прямо или косвенно включать саму себя.

Не будет ошибкой включение несколько раз одного и того же модуля таблицы стилей, прямо или косвенно, но этого не стоит делать. Это легко может привести к ошибкам из-за наличия повторяющихся объявлений. Фактически такие ошибки неизбежны, когда таблица стилей содержит определения глобальных переменных или именованных шаблонов и включена более одного раза с тем же самым преимуществом импортирования. В некоторых других случаях от реализации зависит, сообщит ли XSLT-процессор о повторяющихся объявлениях как об ошибке или нет, поэтому при работе с разными процессорами поведение может быть разным.

Использование

Элемент <xsl:include> обеспечивает простое средство для вложения текстов, аналогичное директиве #include в языке C; это хороший способ разработки модульных таблиц стилей, чтобы часто используемые определения могли были собраны в библиотеки и использовались везде, где они необходимы.

При обработке широкого круга различных типов документов наверняка найдутся общие для них всех элементы, которые во всех случаях должны обрабатываться одинаково. Например, все они могут включать стандартные определения инструментальных панелей, фонов и навигационных кнопок для перемещения по веб-страницам сайта, а также стандартные стили, применяемые для элементов данных, таких как названия программ, контактные адреса электронной почты или даты.

Для включения такого стандартного содержимого в таблицу стилей без изменения используйте <xsl:include>. Если есть определения, которые нужно заменить, используйте <xsl:import>.

Элемент <xsl:include> – это средство, действующее во время компиляции; оно используется для компоновки всей таблицы стилей до ее выполнения. Иногда спрашивают, как можно включить другие таблицы стилей условно, во время выполнения, на основании условий, находящихся в исходном документе. Ответ простой: это невозможно. Это было бы подобно написанию на языке Visual Basic программы, которая изменяет себя во время выполнения. Если нужно, чтобы в разное время в таблице стилей были активны различные наборы правил, можно использовать режимы или рассмотреть другой вариант: вместо универсальной таблицы стилей, которая в различных случаях включает различные наборы правил, можно сделать главным модулем таблицы стилей тот модуль, который наиболее хорошо подходит к обстоятельствам, и использовать <xsl:import>, чтобы импортировать в него универсальные правила с более низким преимуществом импортирования, чем у специализированных правил.

Ситуация несколько меняется, когда в таблице стилей есть инструкция <xsl:include>. Для некоторых типов объектов – особенно для шаблонных правил – при отсутствии других решений XSLT-процессор имеет право предоставлять приоритет тому из них, которое в таблице стилей появляется последним. Этим не всегда можно пользоваться, потому что в таких случаях процессор может вместо этого сообщить об ошибке. Обычно лучше всего разместить инструкции <xsl:include> в начале файла, так как благодаря этому при любых случайных наложениях определений инструкции основной таблицы стилей или заменят определения, включаемые из других мест, или вызовут сообщение об ошибке.

Примеры

Пример: Использование <xsl:include> с именованными наборами атрибутов

Исходный файл

Этот пример можно использовать с любым исходным документом.

Таблица стилей

Рассмотрим основную таблицу стилей картинка.xsl, которая включает таблицу стилей атрибуты.xsl.

Модуль картинка.xsl:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href="атрибуты.xsl"/>
<xsl:template match="/">
   <картинка xsl:use-attribute-sets="атрибуты-картинки">
```

```
        <xsl:attribute name="цвет">красный</xsl:attribute>
    </картинка>
</xsl:template>
</xsl:stylesheet>
```

Модуль атрибуты.xsl:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:attribute-set name="атрибуты-картинки">
    <xsl:attribute name="цвет">синий</xsl:attribute>
    <xsl:attribute name="прозрачность">100</xsl:attribute>
</xsl:attribute-set>
</xsl:stylesheet>
```

Именованный набор атрибутов во включенной таблице стилей используется так же, как если бы он находился в главной таблице стилей в том месте, где помещена инструкция <xsl:include>.

Вывод

Вывод будет следующим:

```
<картинка прозрачность="100" цвет="красный"/>
```

Объяснение: атрибуты, генерируемые при использовании <xsl:attribute>, заменяют атрибуты, сгенерированные при использовании именованного набора атрибутов; в данном случае неважно, что набор атрибутов находится во включенной таблице стилей.

См. также

<xsl:import> на стр. 256

xsl:key

Элемент <xsl:key> – элемент верхнего уровня, используемый для объявления именованных ключей, которые применяются с функцией key() в выражениях и образцах.

Определен в

XSLT, раздел 12.2

Формат

```
<xsl:key name=ПолноеИмя match=Образец use=Выражение />
```

Расположение

<xsl:key> – элемент верхнего уровня, то есть он должен быть непосредственным потомком элемента <xsl:stylesheet>. В таблице стилей он может встречаться любое число раз.

Атрибуты

Имя	Значение	Назначение
name обязательный	Полное имя	Имя ключа
match обязательный	Образец	Определяет узлы, к которым применим этот ключ
use обязательный	Выражение	Выражение, используемое при определении значения ключа для каждого из этих узлов

Определения конструкций ПолноеИмя и Выражение дается в главе 5, а конструкции образец – в главе 6.

Ни образец в атрибуте match, ни выражение в атрибуте use не могут содержать ссылку на переменную. Это должно предотвратить зацикливание определений. Определения ключей обрабатываются раньше определений глобальных переменных. Благодаря этому можно использовать ключи при определении значения глобальных переменных, но нельзя использовать глобальные переменные при определении ключей.

Еще одно ограничение для предотвращения зацикливания определений заключается в том, что ни образец match, ни выражение use не могут использовать функцию key().

Содержимое

Нет; элемент всегда пуст.

Действие

Атрибут name определяет имя ключа. Он должен быть действительным полным именем; если он содержит префикс пространства имен, то префикс должен идентифицировать объявление пространства имен, в области действия которого находится элемент <xsl:key>. Действующее имя ключа – расширенное имя, состоящее из URI пространства имен и локальной части имени. Пространства имен описаны в соответствующем разделе главы 2.

Атрибут match определяет узлы, к которым применяется ключ. Его значением является образец, описанный в главе 6. Если узел не соответствует образцу, то он не будет иметь никаких значений для именованного ключа. Если узел соответствует образцу, то он будет иметь нуль или больше значений для именованного ключа, в зависимости от значения атрибута use.

Самый простой случай – когда значения ключей уникальны. Например, рассмотрим следующий исходный документ:

```
<автомобили>
<автомобиль номер="P427AGH" владелец="Джо Карлов"/>
<автомобиль номер="T788PHT" владелец="Прунелла Хиггс"/>
<автомобиль номер="V932TXQ" владелец="Вильям Абикомбо"/>
</автомобили>
```

В таблице стилей можно следующим образом определить ключи для регистрационных номеров этих автомобилей:

```
<xsl:key name="номер-автомобиля" match="автомобиль" use="@номер"/>
```

Атрибут use указывает выражение, используемое для определения значения или значений ключа. Это выражение не должно быть атрибутом, подобно «@номер» в примере выше. Например, это может быть дочерний элемент. Если это – повторяющийся дочерний элемент, можно для каждого экземпляра создать вхождение индекса. Формальные правила следующие: для каждого соответствующего образцу узла вычисляется выражение, при этом данный узел является текущим узлом, и только он один входит в текущий список узлов. А затем:

- Если результатом является набор узлов, то каждый узел из этого набора дает одно значение для ключа. Значение ключа – строковое значение этого узла.
- В противном случае результат преобразуется в строку, и эта строка принимается за значение ключа.

Нет никаких правил в отношении двух узлов, имеющих одинаковое значение ключа. Например, объявление ключа для регистрационных номеров автомобилей в примере выше не подразумевает, что все регистрационные номера должны быть различны. Таким образом, узел может иметь несколько значений ключа, а одно и то же значение ключа может относиться к нескольким узлам.

Более формально, каждый именованный ключ может рассматриваться как множество, состоящее из пар узел–значение. Узел может быть связан с несколькими значениями, а значение может быть связано с несколькими узлами. Значение всегда является строкой. Пара узел–значение (N, V) принадлежит множеству, если узел N соответствует образцу, указанному в атрибуте match, и если выражение в атрибуте use при применении к узлу N производит или набор узлов, который содержит узел со строковым значением V, или значение, которое не является набором узлов и которое при преобразовании в строковый тип данных дает V. Чтобы стало еще сложнее, в таблице стилей могут находиться несколько определений ключей с одинаковым именем. В этом случае множество пар узел–значение для ключа является объединением множеств, произведенных каждым определением ключа независимо. Преимущество импортирования для определений ключей не имеет значения.

Ключ можно использовать для выбора узлов в любом документе, а не только в основном исходном документе. Функция key() всегда выдает узлы, кото-

рые находятся в том же документе, где находится контекстный узел во время вызова функции. Поэтому лучше считать, что имеется одно множество пар узел–значение для каждого именованного ключа в каждом документе.

Цель вызова key(K, V), где K – имя ключа, а V – строка – отыскать набор пар узел–значение для ключа по имени K и контекстного документа и вернуть набор узлов, содержащий по одному узлу из каждой пары, имеющей значение V.

Если рассматривать это в терминах SQL, то можно представить таблицу ЗНАЧЕНИЯ-КЛЮЧЕЙ с четырьмя столбцами: ИМЯ-КЛЮЧА, ДОКУМЕНТ, УЗЕЛ и ЗНАЧЕНИЕ. Тогда вызов key('K', 'V')эквивалентен оператору SQL:

```
SELECT DISTINCT УЗЕЛ FROM ЗНАЧЕНИЯ-КЛЮЧЕЙ WHERE
   ИМЯ-КЛЮЧА='K' AND
   ЗНАЧЕНИЕ='V' AND
   ДОКУМЕНТ=текущий-документ;
```

Использование и примеры

Объявление ключа производит два эффекта: во-первых, упрощает код, который нужно написать, чтобы найти узлы с данными значениями, и во-вторых, может ускорить доступ к узлам.

Повышение эффективности, конечно, полностью зависит от используемой реализации. Вполне законно, если конкретная реализация будет проводить полный поиск по документу при каждом вызове функции key(). На практике, однако, большинство программ, вероятно, будет формировать индекс или хэш-таблицу, поэтому полный поиск по данному документу будет производиться только один раз при формировании индекса (для каждого документа), а после этого доступ к узлам, значение ключей для которых известно, должен быть очень быстрым.

Элемент <xsl:key> обычно используется для индексации элементов, но в принципе, с его помощью можно индексировать любой вид узлов, кроме узлов пространств имен.

Использование простого ключа

Детальные правила для ключей выглядят сложно, но большинство практических применений ключей очень просты. Рассмотрим следующее определение ключа:

```
<xsl:key name="код-продукта" match="продукт" use="@код"/>
```

Здесь определяется ключ, имя которого «код-продукта» и который можно использовать для отыскания элементов <продукт> по значениям их атрибута код. Если продукт не имеет атрибута код, его нельзя будет найти с помощью этого ключа.

Для того чтобы найти продукт, код которого «ABC-456», можно написать, например, так:

```
<xsl:apply-templates select="key('код-продукта', 'ABC-456')"/>
```

Заметьте, что точно так же можно было произвести индексацию узлов атрибутов:

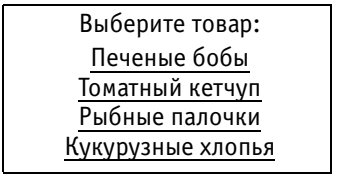
```
<xsl:key name="код-продукта" match="продукт/@код" use="."/>
```

Тогда для отыскания соответствующего продукта нужно было бы написать:

```
<xsl:apply-templates select="key('код-продукта', 'ABC-456')/.."/>
```

Здесь в качестве примера рассмотрена инструкция <xsl:apply-templates>. Ее задача – выбрать в текущем документе все элементы <продукт>, которые имеют код «ABC-456» (никто не говорит, что это должен быть уникальный идентификатор), и применить к каждому по очереди соответствующий шаблон, обрабатывая их, как обычно, в порядке следования в документе. Можно было взять и любую другую инструкцию, которая использует выражение XPath; например, можно было бы присвоить набор узлов переменной или использовать его в элементе <xsl:value-of>.

Второй параметр у функции key – обычно строка данных. Это, как правило, будет не литерал, как в данном примере, а более вероятно, что это будет строка, полученная из исходного документа или являющаяся параметром, переданным таблице стилей. Это, прежде всего, мог бы быть один из параметров URL, используемого для выбора этой таблицы стилей. Например, веб-страница могла бы отображать список имеющихся продуктов:



Каждой из этих гиперссылок, показываемых пользователю, соответствует URL вида:

```
http://www.cheap-food.com/servlet/product?code=ABC-456
```

Тогда можно на своем веб-сервере написать сервлет (или ASP-страницу, если хотите), который извлечет параметр запроса code и запустит ваш любимый XSLT-процессор с файлом продукты.xml в качестве исходного документа, файлом вывести-продукт.xsl – в качестве таблицы стилей и строкой «ABC-456» – в качестве значения, которое будет присвоено глобальному параметру таблицы стилей, названному код-прод. Таблица стилей в этом случае выглядела бы следующим образом:

```
<xsl:param name="код-прод"/>
<xsl:key name="код-продукта" match="продукт" use="@код"/>
<xsl:template match="/">
   <html>
   <body>
   <xsl:variable name="продукт" select="key('код-продукта', $код-прод)"/>
   <xsl:if test="not($продукт)">
```

```
      <p>Не найден продукт, код которого равен данному</p>
    </xsl:if>
    <xsl:apply-templates select="$продукт"/>
    </body>
    </html>
</xsl:template>
```

Многозначные ключи

Ключ может быть многозначным, то есть один узел может иметь несколько значений, каждое из которых можно независимо использовать для отыскания узла. Например, книга может иметь несколько авторов, и имя каждого автора может служить значением ключа. Это можно записать следующим образом:

```
<xsl:key name="автор-книги" match="книга" use="автор/имя"/>
```

Выражение use, «автор/имя», возвращает набор узлов, поэтому строковое значение каждого из его узлов (то есть имя каждого автора книги) используется как одно из значений в наборе пар узел–значение, которые составляют ключ.

В данном конкретном примере одна книга имеет несколько авторов, а каждый автор, возможно, написал несколько книг, поэтому, если использовать такое выражение XPath:

```
<xsl:for-each select="key('автор-книги', 'Агата Кристи')">
```

будут выбираться все книги, в которых одним из авторов является Агата Кристи. Что если требуется найти все книги, в которых Алекс Хомер (Alex Homer) и Дэвид Суссман (David Sussman) являются соавторами? Невозможно непосредственно объединить результаты двух различных ключей, но есть косвенный способ.

По крайней мере, два процессора, Saxon и xt, поддерживают функцию расширения intersection(), которая позволяет написать следующее выражение XPath:

```
prefix:intersection(
    key('автор-книги', 'Алекс Хомер'),
    key('автор-книги', 'Дэвид Суссман'))
```

Однако такой функции нет в стандарте. Вместо этого, когда нужно найти пересечение двух ключей, требуется написать довольно мудреное выражение:

```
<xsl:variable name="множество1" select="key('автор-книги', 'Алекс Хомер')"/>
<xsl:variable name="множество2" select="key('автор-книги', 'Дэвид Суссман')"/>
<xsl:variable name="результат" select="$множество1[ count(.|$множество2) =
                                                    count($множество2)]"/>
```

Это работает, потому что результат вызова функции «count(X|Y)» будет таким же, как «count(Y)», если только X является подмножеством Y (вспомните, что «|» – оператор объединения набора); таким образом, предикат фильтрует

содержимое $множество1, выбирая только те элементы, которые есть также и в $множество2.

Можно передать набор узлов в качестве второго параметра функции key(). Например, можно написать:

```
<xsl:variable name="ак" select="key('автор-книги', 'Агата Кристи')">
<xsl:for-each select="key('автор-книги', $ак/автор/имя)">
```

Результат выражения select в инструкции <xsl:for-each> – множество всех книг, в число авторов которых входит Агата Кристи или один из ее соавторов. В данном выражении $ак – это набор всех книг, в которых Агата Кристи – автор, поэтому «$ак/автор/имя» – набор всех авторов этих книг, а использование этого набора авторов как значения ключа дает набор книг, в число авторов которых входит один из этих авторов. Как уже говорилось, нет простого способа найти книги, в которых они **все** являются авторами.

Пример: Многозначные неуникальные ключи

В этом примере приведена ситуация, когда узел имеет несколько значений для одного ключа, а каждое значение ключа идентифицирует более одного узла. Здесь имя автора является ключом для отыскания элементов <книга>.

Исходный файл

Исходный файл – книги.xml:

```
<книги>
<книга>
    <название>Паттерны проектирования</название>
    <автор>Эрих Гамма</автор>
    <автор>Ричард Хелм</автор>
    <автор>Ральф Джонсон</автор>
    <автор>Джон Влиссидес</автор>
</книга>
<книга>
    <название>Поиск паттернов</название>
    <автор>Джон Влиссидес</автор>
</книга>
<книга>
    <название>Построение инфраструктуры приложений</название>
    <автор>Мохамед Фаяд</автор>
    <автор>Дуглас Шмидт</автор>
    <автор>Ральф Джонсон</автор>
</книга>
<книга>
    <название>Реализация инфраструктуры приложений</название>
    <автор>Мохамед Фаяд</автор>
    <автор>Дуглас Шмидт</автор>
    <автор>Ральф Джонсон</автор>
</книга>
</книги>
```

Таблица стилей

Таблица стилей – ключ-автор.xsl.

В ней объявляется ключ, а затем просто копируются элементы <книга>, которые соответствуют имени автора, указанному в качестве параметра. Таким образом, можно вызвать эту таблицу стилей следующим образом:

```
saxon книги.xml ключ-автор.xsl автор="Ральф Джонсон"
```

Заметьте, что параметры, содержащие пробелы, в командной строке должны заключаться в кавычки. Точный синтаксис для каждого процессора различен, подробнее см. в соответствующих приложениях. Для удобства при опробовании этой таблицы стилей параметр снабжен значением по умолчанию.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:key name="имя-автора" match="книга" use="автор"/>
<xsl:param name="автор" select="'Джон Влиссидес'"/>
<xsl:template match="/">
    <xsl:copy-of select="key('имя-автора', $автор)"/>
</xsl:template>
</xsl:transform>
```

Вывод

С параметром, установленным в его значение по умолчанию: «Джон Влиссидес», вывод будет таким:

```
<?xml version="1.0" encoding="utf-8" ?>
<книга>
    <название>Паттерны проектирования</название>
    <автор>Эрих Гамма</автор>
    <автор>Ричард Хелм</автор>
    <автор>Ральф Джонсон</автор>
    <автор>Джон Влиссидес</автор>
</книга>
<книга>
    <название>Поиск паттернов</название>
    <автор>Джон Влиссидес</автор>
</книга>
```

Множественные именованные ключи

Ничто не мешает определению нескольких ключей для одних и тех же узлов. Например:

```
<xsl:key name="isbn-книги" match="книга" use="isbn"/>
<xsl:key name="автор-книги" match="книга" use="автор/фамилия"/>
```

Это позволяет отыскать книгу, если известен или ее автор, или ISBN.

Однако прежде чем делать это, стоит дважды подумать. Предположим, что XSLT-процессор работает с ключами, формируя индекс или хеш-таблицу, а не просматривая каждый раз весь документ. В этом случае следует сопоставить затраты ресурсов на формирование индекса с затратами на отыскание информации поиском. Если цель преобразования – найти только одну книгу, используя ее код ISBN, то проще и быстрее сделать это следующим образом:

```
<xsl:for-each select="//книга[isbn='0-13-082676-6']"/>
```

и вообще не использовать ключ.

Множественные определения одного и того же ключа

Возможно также иметь несколько определений ключа с одним и тем же именем. Например:

```
<xsl:key name="ключ-создателя" match="книга" use="автор/имя"/>
<xsl:key name="ключ-создателя" match="CD" use="композитор"/>
<xsl:key name="ключ-создателя" match="CD" use="исполнитель"/>
```

Тогда можно использовать функцию `key()` в выражении такого типа:

```
<xsl:apply-templates select="key('ключ-создателя', 'Ринго Старр')"/>
```

Набор узлов, который выдаст это выражение, будет состоять из элементов `<книга>` или элементов `<CD>`, или из комбинации обоих; единственное, что можно сказать заранее, – что каждый элемент будет представлять или книгу, один из авторов которой – Ринго Старр, или компакт-диск с Ринго Старром в качестве композитора или исполнителя.

Если бы выражение `use` каждый раз было одно и то же, можно было бы упростить его. Например, чтобы найти книги и компакт-диски конкретного издателя, можно написать:

```
<xsl:key name="ключ-издателя" match="книга | CD" use="издатель"/>
```

В этом примере использован образец «`книга | CD`», который соответствует всем элементам `<книга>` и всем элементам `<CD>`. Образцы описываются в главе 6.

Разные определения необязательно должны находиться в одном и том же модуле таблицы стилей; все определения ключей во включенных и импортированных таблицах стилей объединяются, независимо от их преимущества импортирования.

См. также

Функция `key()` в главе 7

xsl:message

Инструкция <xsl:message> выводит сообщение и может прекратить (опционально) выполнение таблицы стилей.

Определен в

XSLT, раздел 13

Формат

```
<xsl:message terminate="yes" | "no" >
   тело шаблона
</xsl:message>
```

Расположение

<xsl:message> – инструкция, и всегда используется в теле шаблона.

Атрибуты

Имя	Значение	Назначение
terminate необязательный	«yes» | «no»	Значение «yes» указывает, что после вывода сообщения обработка прекращается. Значение по умолчанию – «no».

Содержимое

Тело шаблона. Нет специального указания, что оно должно генерировать только текстовые узлы; оно может производить любой фрагмент XML. Однако в стандарте не определено, что должно происходить с разметкой.

Действие

Если атрибут terminate опущен, подразумевается значение «no».

Значение, получаемое раскрытием шаблона, выводится там, где пользователь может его видеть. Фактически в спецификации XSLT не уточняется, где именно; это зависит от реализации, а также может быть определено в конфигурационных параметрах. Спецификация предлагает два возможных назначения: окно с сообщением на экране и регистрационный журнал.

Если атрибут terminate имеет значение «yes», выполнение таблицы стилей немедленно прекращается, а любой сгенерированный ею вывод никуда не выводится.

Использование

Инструкция <xsl:message> обычно используется для сообщения о сбойных ситуациях, обнаруженных логикой таблицы стилей. Примером может слу-

жить ситуация, когда, скажем, элемент <продажи> должен иметь числовое значение, а найдено не числовое значение.

- При «terminate="no"» (значение по умолчанию) таблица стилей может сообщить об ошибке и продолжить обработку.
- При «terminate="yes"» таблица стилей может сообщить об ошибке и прекратить работу.

Перед использованием <xsl:message> в таблицах стилей, предназначеных для конечных пользователей, проверьте, что происходит с сообщениями и можно ли их переадресовать. Следует особенно четко продумать, для кого предназначены сообщения: для автора исходного документа, автора таблицы стилей или для конечных пользователей, – от этого зависит текст сообщения и способ его вывода.

Вывод, производимый <xsl:message>, может быть непредсказуемым, так как последовательность выполнения таблицы стилей в стандарте не определена. Например, некоторые процессоры (в частности, xt) откладывают вычисление переменной до момента ее первого использования, поэтому порядок вычисления различных переменных трудно угадать. Если вычисление переменной запускает выполнение <xsl:message>, порядок сообщений может быть неожиданным. Безусловно, все это зависит от используемого XSLT-процессора.

Обычным применением <xsl:message> является генерирование диагностических сообщений с целью выяснения, почему таблица стилей не ведет себя ожидаемым образом. Это хорошо работает с процессорами типа Saxon и Xalan, которые имеют довольно предсказуемую последовательность выполнения, но с xt это может дать удивительные результаты, поскольку он часто производит обработку в неожиданном порядке. Вероятно, наиболее гибким решением является помещение диагностики в виде комментариев в конечное дерево (с помощью <xsl:comment>). Некоторые продукты, конечно, имеют определенные поставщиком встроенные средства отладки.

> ***Анализатор MSXML3 от Microsoft в настоящее время игнорирует установку*** <xsl:message terminate="no">***, поэтому сообщение никуда не выводится. При*** «terminate="yes"» ***он генерирует сообщение об ошибке, которое может быть обработано сценарием HTML-страницы, вызвавшей данное преобразование.***

Примеры

В следующем примере показан код для выдачи сообщения и прекращения обработки, если значение элемента <продажи> окажется не числовым:

```
<xsl:if test="string(number(продажи))='NaN'">
   <xsl:message terminate="yes">
      <xsl:text>Значение элемента <продажи> не является числовым</xsl:text>
   </xsl:message>
</xsl:if>
```

К сожалению, в стандарте XSLT не определен механизм, который позволил бы локализовать ошибку в исходном документе и отразить это в сообщении.

Следующий пример – более расширенный. Он допускает наличие в одном прогоне нескольких ошибок, о которых будет сообщаться, а выполнение прекращается только после того, как будут выявлены все ошибки. Для этого глобальной переменной присваивается набор узлов с ошибками.

```
<xsl:variable name="неверные-продажи"
              select="//продажи[string(number(current()))='NaN']"/>
<xsl:template match="/">
   <xsl:for-each select="$неверные-продажи">
      <xsl:message>Значение элемента <продажи> <xsl:value-of select="."/>
         не является числовым
      </xsl:message>
   </xsl:for-each>
...
   <xsl:if test="$неверные-продажи">
      <xsl:message terminate="yes">
         <xsl:text>Обработка прекращена</xsl:text>
      </xsl:message>
   </xsl:if>
</xsl:template>
```

Локализованные сообщения

XSLT разрабатывался с явным учетом интернационализации, и, несомненно, вопрос о локализации текстов сообщений рассматривался при проектировании языка. Было твердо решено, что не нужны никакие специальные средства, а вместо этого в спецификацию XSLT включен детальный пример, показывающий, как можно локализовать текст сообщения (выводить его на родном языке пользователя). Этот пример стоит повторить, поскольку он демонстрирует общую методику.

Сообщения для конкретного языка сохраняются в файле, имя которого задает язык, например, сообщения на немецком языке могут содержаться в файле `messages/de.xml`. Примерная структура файла сообщений следующая:

```
<сообщения>
   <сообщение код="запуск">Angefangen</сообщение>
   <сообщение код="подождите"/>Bitte warten!</сообщение>
   <сообщение код="завершение"/>Fertig</сообщение>
</сообщения>
```

В таблицу стилей, которая предполагает вывод сообщений на соответствующем локальном языке, нужно ввести параметр для задания этого языка (это можно сделать также через функцию `system-property()`, описанную в главе 7, но при этом ухудшается переносимость между разными системами). Тогда таблица стилей получит доступ к файлу сообщений для соответствующего языка и будет читать сообщения оттуда:

```
<xsl:param name="язык" select="'en'"/>
```

```
<xsl:template name="вывести-сообщение">
   <xsl:param name="код"/>
   <xsl:variable name="файл-сообщений"
                 select="concat('messages/', $language, '.xml')"/>
   <xsl:variable name="текст-сообщения"
                 select="document($файл-сообщений)/сообщения"/>
   <xsl:message>
      <xsl:value-of select="$текст-сообщения/сообщение[@код=$код]"/>
   </xsl:message>
</xsl:template>
```

Этой же методикой можно воспользоваться и для создания локализованных текстов, которые будут включены таблицей стилей в выходной файл.

xsl:namespace-alias

Элемент <xsl:namespace-alias> позволяет отображать пространство имен, использующееся в таблице стилей, в другое пространство имен, используемое в выводе. Это обычно требуется для записи преобразований, которые производят в качестве вывода таблицу стилей XSLT.

Определен в

XSLT, раздел 7.1.1

Формат

```
<xsl:namespace-alias
     stylesheet-prefix=ИмяБезДвоеточия
     result-prefix=ИмяБезДвоеточия />
```

Расположение

<xsl:namespace-alias> — элемент верхнего уровня, то есть он должен быть непосредственным потомком элемента <xsl:stylesheet>. В таблице стилей он может встречаться любое число раз.

Атрибуты

Имя	Значение	Назначение
stylesheet-prefix обязательный	ИмяБезДвоеточия | «#default»	Префикс пространства имен, используемый в таблице стилей
result-prefix обязательный	ИмяБезДвоеточия | «#default»	Префикс соответствующего пространства имен, который будет использован в выводе

Содержимое

Нет; элемент <xsl:namespace-alias> всегда пуст.

Действие

Элемент <xsl:namespace-alias> воздействует на обработку пространств имен у конечных литеральных элементов.

Обычно, когда узел элемента выводится при обработке конечного литерального элемента, имя выводимого элемента будет иметь ту же локальную часть, тот же префикс и тот же URI пространства имен, как и у самого конечного литерального элемента. Не гарантируется, что префикс будет тем же самым, но обычно это так. То же самое относится и к атрибутам конечного литерального элемента. Узлы пространств имен конечного литерального элемента должны копироваться в конечное дерево без изменений, используя тот же самый префикс и URI пространства имен. (В спецификации XSLT заявлено, что при обработке конечного литерального элемента все пространства имен, в области действия которых находится элемент, за некоторым определенным исключением, также должны присутствовать в выводе, даже если они не используются. Избыточные узлы пространств имен могут быть исключены с помощью атрибута xsl:exclude-result-prefixes. Подробнее об этом см. в разделе «Конечные литеральные элементы» главы 3.)

Предположим, требуется, чтобы выводимый документ был таблицей стилей XSLT. Тогда нужно создавать элементы типа <xsl:template>, которые находятся в пространстве имен XSLT. Однако элемент <xsl:template> нельзя создавать как конечный литеральный элемент, так как по определению, если элемент использует пространство имен XSLT, он обрабатывается как элемент XSLT.

В этом случае в таблице стилей для конечного литерального элемента нужно использовать другое пространство имен и включить объявление <xsl:namespace-alias>, чтобы при выводе конечного литерального элемента новое пространство имен преобразовывалось в пространство имен XSLT. Таким образом, конечный литеральный элемент в таблице стилей можно назвать <out:template>, и можно использовать элемент <xsl:namespace-alias>, чтобы указать, что префикс «out» таблицы стилей должен преобразовываться в выводе в префикс «xsl».

Элемент <xsl:namespace-alias> объявляет, что один URI пространства имен – URI таблицы стилей – при выводе конечных литеральных элементов должен быть заменен другим URI – конечным URI. URI пространств имен не указываются прямо, но подразумеваются через префиксы, связанные с данным URI пространства имен в объявлениях пространств имен, которые действуют в текущий момент. Любой из URI пространств имен может быть URI пространства имен по умолчанию, что указывается использованием псевдопрефикса «#default».

Таким образом, хотя элемент <xsl:namespace-alias> описывает отображение через префиксы, изменяются вовсе не префиксы, а URI.

Замена одного URI пространства имен на другой отражается непосредственно на именах конечных литеральных элементов и именах всех атрибутов конечных литеральных элементов. Она отражается также на URI узлов про-

странств имен, копируемых в конечное дерево из конечного литерального элемента. Эта замена не воздействует на элементы, созданные с помощью <xsl:element>, на атрибуты, созданные с помощью <xsl:attribute>, или на узлы, скопированные с помощью <xsl:copy>. Префикс узлов пространств имен не изменяется (он будет stylesheet-prefix, а не result-prefix), и поэтому есть вероятность, что префиксы элементов и атрибутов также не изменятся, но как всегда, процессор имеет возможность дать этим именам другой префикс, но обязательно соответствующий правильному URI пространства имен.

Если есть несколько элементов <xsl:namespace-alias>, в которых задан один и тот же stylesheet-prefix, то используется тот, который имеет самое высокое преимущество импортирования. Если их оказывается более одного, будет сообщено об ошибке или будет выбран элемент, который появляется в таблице стилей последним.

Введение псевдонимов URI пространств имен применяется, когда происходит обработка конечного литерального элемента в таблице стилей для создания элемента в конечном дереве. Не имеет значения, является ли конечное дерево окончательным выводом или временным деревом. Это означает, что если производится разбор временного дерева, в которое были скопированы конечные литеральные элементы, то соответствующие элементы и атрибуты будут использовать URI пространства имен, связанный с конечным префиксом, а не с префиксом таблицы стилей.

Использование и примеры

Основная мотивация для введения этого средства – обеспечение возможности создания таблиц стилей, оно генерирует таблицы стилей в качестве вывода. Возможно, это звучит не очень правдоподобно, но для использования этой методики есть много веских причин, включая следующие:

- В настоящее время используется много частных языков шаблонов. Трансляция этих шаблонов в таблицы стилей XSLT выглядит привлекательным способом перехода, и почему бы этим трансляторам не быть написанными на XSLT?
- Довольно долго может сохраняться потребность в языке шаблонов, не таком сложном и мощном, как XSLT, для использования людьми, далекими от программирования. Эти простые шаблоны также могут быть легко преобразованы в таблицы стилей XSLT.
- Некоторые части таблицы стилей XSLT нелегко параметризовать. Например, невозможно программно сконструировать XPath-выражение, а затем выполнить его (XSLT – не рефлексивный язык). Необходимость в этом возникает при разработке визуальных инструментальных средств, когда нужно в интерактивном режиме определять запросы и отчеты. Одним из способов осуществления таких инструментальных средств является создание настраиваемой таблицы стилей из универсальной таблицы, а это преобразование тоже можно производить с помощью XSLT.

- Возможна и такая ситуация: имеется много готовых таблиц стилей, имеющих некоторую общую характеристику. Например, все они могут генерировать HTML, в котором используется тег <CENTER>. Может понадобиться заменить тег <CENTER> на другой, тогда нужно будет изменить все эти таблицы стилей, чтобы использовать <DIV ALIGN="CENTER">. Почему бы для их конвертирования не написать XSLT-преобразование?
- Уже появляются инструментальные средства, позволяющие генерировать таблицы стилей XSLT из схем (см. например, Schematron на веб-странице *http://www.ascc.net/xml/resource/schematron/schematron.html*). Поскольку и схема, и таблица стилей – XML-документы, это будет преобразование XML в XML, так что его можно написать на языке XSLT.

Фактически после того, как приложено столько стараний, чтобы определить таблицы стилей XSLT как правильно построенные XML-документы, будет просто удивительно, если окажется невозможным манипулировать ими с помощью самого языка XSLT.

Есть и другие ситуации, где элемент <xsl:namespace-alias> оказывается полезным. Одна из них упоминается в спецификации XSLT – когда требуется избежать использования URI пространств имен, которые связаны с безопасностью в области цифровых подписей. Другая потребность возникает, когда таблицы стилей и другие документы сохраняются в системе управления конфигурацией; в этом случае необходимо гарантировать, что пространства имен, принятые системой управления конфигурацией, например, для описания авторства и перечня изменений документа, не используются прямо в таблице стилей.

Пример <xsl:namespace-alias>

Следующий пример генерирует таблицу стилей XSLT, состоящую только из одного объявления глобальной переменной, имя и значение по умолчанию которой заданы как параметры. Хотя эта таблица стилей тривиальна, она может пригодиться для включения с помощью <xsl:include> или <xsl:import> в другую, более полезную таблицу стилей.

Этот пример доступен для загрузки как файл alias.xsl.

Исходный файл

Эта таблица стилей может использоваться с любым исходным XML-документом. Исходный документ не используется (хотя он должен существовать).

Таблица стилей

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xslt="output.xsl">
<xsl:param name="variable-name">v</xsl:param>
<xsl:param name="default-value"/>
```

```
<xsl:output indent="yes"/>

<xsl:namespace-alias stylesheet-prefix="xslt" result-prefix="xsl"/>

<xsl:template match="/">
   <xslt:stylesheet version="1.0">
   <xslt:variable name="{$variable-name}">
      <xsl:value-of select="$default-value"/>
   </xslt:variable>
   </xslt:stylesheet>
</xsl:template>

</xsl:stylesheet>
```

Вывод

Если за значения параметров «variable-name» и «default-value» берутся значения по умолчанию, вывод мог бы быть следующим. Это вывод с процессором Saxon, с другими процессорами он может быть немного другим.

```
<?xml version="1.0" encoding="utf-8"?>
<xslt:stylesheet
       xmlns:xslt="http://www.w3.org/1999/XSL/Transform"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       version="1.0">
   <xslt:variable name="v"/>
</xslt:stylesheet>
```

Зачем здесь два объявления пространств имен, в частности объявление для префикса «xsl»? Это, вероятно, причуда процессора Saxon: хотя из правил, по-разному изложенных в XSLT 1.0 и в XSLT 1.1, ясно в обоих случаях, что это объявление пространства имен допустимо, но не обязательно. Saxon выводит его, потому что его алгоритм для исключения ненужных объявлений пространств имен рассматривает только URI пространств имен, но не префиксы пространств имен.

Факт, что процессор может вывести объявление пространства имен также и для префикса «xsl», означает, что он может использовать этот префикс в именах элементов <xslt:variable> и <xslt:stylesheet>. Однако он должен вывести объявление пространства имен для префикса «xslt», даже если этот префикс не используется, так как он соответствует непосредственно узлу пространства имен в конечном дереве.

См. также

«Конечные литеральные элементы» в главе 3.

xsl:number

Элемент <xsl:number> выполняет две функции. Он может использоваться для назначения порядкового номера текущему узлу, а также для форматирова-

ния номеров для вывода. Эти функции часто совмещены, но могут выполняться и отдельно.

Заметьте, что средства для форматирования номеров в элементе <xsl:number> не связаны со средствами, предлагаемыми в функции format-number() и элементе <xsl:decimal-format>.

Определен в

XSLT, раздел 7.7

Формат

```
<xsl:number
   level="single" | "multiple" | "any"
   count=Образец
   from=Образец
   value=Выражение
   format={строка-формата}
   lang={код-языка)
   letter-value={ "alphabetic" | "traditional" }
   grouping-separator={символ}
   grouping-size={число} />
```

Расположение

<xsl:number> – инструкция, и всегда используется в пределах тела шаблона.

Атрибуты

Имя	Значение	Назначение
level необязательный	«single» | «multiple» | «any»	Контролирует способы распределения порядковых номеров по позиции узла в дереве
count необязательный	Образец	Определяет, какие узлы подсчитаны, чтобы определить порядковый номер
from необязательный	Образец	Определяет точку остановки: точку в документе, от которой последовательность нумерации начинается заново
value необязательный	Выражение	Определенное пользователем число, которое будет форматироваться (вместо использования порядкового номера узла)
format необязательный	Шаблон значений атрибутов, который выдает форматирующую строку, как определено ниже	Определяет выходной формат номера

Имя	Значение	Назначение
lang необязательный	Шаблон значений атрибутов, который выдает код языка, как определено в XML для атрибута xml:lang	Указывает язык, соглашения которого для форматирования чисел должны использоваться
letter-value необязательный	Шаблон значений атрибутов, который выдает «alphabetic» | «traditional»	Различает альтернативные схемы нумерования, используемые в данном языке
grouping-separator необязательный	Шаблон значения атрибута, который выдает единственный символ	Символ, который нужно использовать как разделитель между группами цифр (например, запятая как разделитель для тысяч)
grouping-size необязательный	Шаблон значения атрибута, который выдает число	Количество разрядов в каждой группе, указывающее, где должен быть вставлен разделитель групп

Синтаксис выражений, см. в главе 5.

Синтаксис образцов, см. в главе 6.

Содержимое

Никакого; элемент всегда пуст.

Действие

Инструкция <xsl:number> выполняет четыре задачи:

- Определяет порядковый номер
- Разбирает форматирующую строку на последовательность лексем формата
- Форматирует каждую часть порядкового номера, используя соответствующую лексему формата
- Записывает результирующую строку в текущем назначении вывода как текстовый узел

Эти последовательные шаги рассматриваются по отдельности в следующих разделах.

Определение порядкового номера

Если атрибут value определен, порядковый номер получается вычислением выражения в атрибуте value и последующим конвертированием результата в число (в случае необходимости) с использованием правил для функции number(). Полученный результат затем округляется до целого числа с использованием правил для функции round(). В этом случае атрибуты level, count и from игнорируются.

Будет ошибкой, если номер окажется бесконечностью, не-числом, нулем или отрицательным числом (или, на практике, меньше чем 0.5), так как эле-

мент <xsl:number> предназначен для обработки положительных целых чисел. XSLT-процессор может посчитать эту ошибку фатальной, или он может произвести представление номера, предусмотренное на случай сбойных ситуаций, используя те же конверсионные правила, которые применяются к функции string(). Элемент <xsl:number> предназначен для обработки натуральных чисел, которые являются результатом подсчета узлов, поэтому при необходимости обрабатывать другие ситуации лучше использовать функцию format-number(), описанную в главе 7.

Если атрибут value не задан, <xsl:number> определяет порядковый номер по позиции текущего узла в исходном документе.

Правила для определения порядкового номера зависят от значения атрибутов level, count и from. Если любой из этих атрибутов опущен, значения по умолчанию следующие:

Атрибут	Значение по умолчанию
level	«single»
count	Образец, который соответствует узлам того же типа, что и текущий узел, и который, если текущий узел имеет имя, соответствует узлам с тем же именем. Как всегда, имена с префиксом пространства имен сопоставляются, используя значимый URI пространства имен, а не префикс.
from	Образец, который не соответствует никаким узлам, например «*[false()]».

Порядковый номер – это обычно список положительных целых чисел. Если атрибут level имеет значение «single» или «any», то порядковый номер будет обычно содержать одно целое число, если значение атрибута – «multiple», тогда номер может содержать несколько чисел (например, «3.6.1»). Кроме того, список может быть пустым.

Порядковые номера определяются следующим образом:

Уровень (level)	Правила
single	Это значение предназначено для нумерования равноправных узлов на одном и том же уровне структуры, например пунктов списка. Если текущий узел соответствует образцу count, то целевым узлом становится текущий узел. В противном случае процессор ищет родителя текущего узла, который соответствует образцу count, и делает его целевым узлом. Поиск прекращается, когда найден родитель, соответствующий образцу from, если такой родитель имеется. Если целевой узел найден, порядковый номер определяется подсчетом количества предшествующих целевому узлу равноуровневых элементов, соответствующих образцу count, и добавлением к этому количеству единицы для учета самого целевого узла. Например, если целевой узел имеет шесть предшествующих равноуровневых элементов, которые соответствуют образцу count, тогда его порядковый номер – 7.

Уровень (level)	Правила
	Если никакой целевой узел не найден, список порядковых номеров будет пустым.
any	Это значение предназначено для нумерования узлов, которые могут находиться на любом уровне структуры: например, сноски или уравнения в главе книги. Начиная с текущего узла процессор просматривает документ в обратном направлении, в обратном порядке документа, считая число узлов, которые соответствуют образцу count, и останавливаясь, когда найден узел, соответствующий образцу from, если такой узел имеется. Рассматриваемыми узлами являются узлы на осях preceding, ancestor и self. Порядковый номер – количество подсчитанных узлов. Если ни один из рассмотренных узлов не соответствует образцу count, порядковым номером будет пустой список. Узлы атрибутов и пространств имен никогда не подсчитываются.
multiple	Это значение предназначено для выведения составных порядковых номеров, которые отражают иерархическую позицию узла, например «2.17.1». Процессор составляет список всех родителей текущего узла вместе с самим текущим узлом, но останавливается, когда находит родителя, соответствующего образцу from, если такой родитель имеется. Узел, соответствующий образцу from, не включается в список. Процессор составляет список в порядке документа, то есть самым первым идет самый внешний родитель. Для каждого узла в этом списке, соответствующего образцу count, процессор подсчитывает, сколько данный узел имеет предшествующих одноуровневых элементов, которые также соответствуют образцу count, и добавляет к этому количеству единицу для учета этого узла непосредственно. Из полученного списка чисел выводится составной порядковый номер. Этот список также может быть пустым.

Эти правила кажутся сложными, но на практике наиболее частые случаи достаточно просты, что будет продемонстрировано в последующих примерах.

Анализ форматирующей строки

Следующей стадией после определения порядкового номера является форматирование его в строковые данные.

Порядковый номер, как было показано, является списком – пустым или содержащим какое-то количество целых чисел. Форматирование контролируется, прежде всего, использованием форматирующей строки, представленной в атрибуте format. Если этот атрибут опущен, значением по умолчанию является «1».

Форматирующая строка состоит из последовательности чередующихся лексем форматирования и лексем пунктуации. Любое сочетание последовательных алфавитно-цифровых символов считается лексемой форматирования, любая другая последовательность считается лексемой пунктуации. Например, если атрибут формата имеет значение «1((a))», он разбивается на лексе-

му форматирования «1», лексему пунктуации «((», лексему форматирования «a» и лексему пунктуации «))». Термин «алфавитно-цифровой» основан на категориях символов Unicode и включает в себя символы и цифры из любого языка.

В наиболее общем случае порядковый номер – это единственное число. Тогда строка вывода состоит из начальной лексемы пунктуации, если она есть, затем идет результат форматирования номера с помощью первой лексемы формата, а после него – заключительная лексема пунктуации, если она есть. Так, если порядковый номер – «42», а значение атрибута формата – «[1]», то конечным выводом будет «[42]».

Когда порядковым номером является список чисел, правила несколько сложнее, но все равно достаточно интуитивны: например, если список чисел – «3, 1, 6», а значение атрибута формата – «1.1(a)», тогда конечным выводом будет «3.1(f)» (потому что «f» – шестой символ в алфавите). Детальные правила следующие:

- Для форматирования *n*-го номера в списке, по возможности, используется *n*-я лексема форматирования, применяя правила из следующего раздела.
- Если в списке больше чисел, чем лексем форматирования, то избыточные числа форматируются с использованием последней лексемы форматирования. Например, если список – «3,1,2,5», а значение атрибута формата – «A.1», тогда конечным выводом будет «C.1.2.5».
- Если нет никаких лексем форматирования, то используется лексема форматирования «1».
- Если лексем форматирования больше, чем чисел в списке, то избыточные лексемы форматирования игнорируются.
- В выводе каждому номеру предшествует лексема пунктуации, которая, если она есть, стоит перед лексемой форматирования, используемой для форматирования данного номера. Если никакой предшествующей лексемы пунктуации нет, и номер – не первый в списке, то перед ним ставится «.».
- Если форматирующая строка заканчивается лексемой пунктуации, эта лексема пунктуации добавляется в конец строки вывода.

Имейте в виду, что если порядковый номер – пустой список, результат будет состоять из начальной и заключительной лексем пунктуации. Например, если строка формата – «[1]», пустой список будет отформатирован как «[]». Наиболее вероятная причина получения пустого списка в том, что никакие узлы не соответствовали образцу `count`.

Форматирование частей порядкового номера

В этом разделе описывается, как форматируется одно число, используя одну лексему форматирования и создавая строку, которая является частью окончательной выводимой строки.

В спецификации XSLT этот процесс определен только частично. Есть некоторые характерные правила, некоторые рекомендации для конструкторов, но многие ситуации остаются неопределенными.

Характерные случаи перечислены в таблице ниже:

Лексемы форматирования	Выводимая последовательность
1	1, 2, 3, 4...
01	01, 02, 03, ... 10, 11, 12 ... Чаще всего, если лексеме формата «1» предшествует *n* нулей, числа в выводе будут в виде десятичных чисел, состоящих минимум из *n+1* цифр.
Другие цифры Unicode	Вышеупомянутые два правила относятся также к любым другим цифрам Unicode, эквивалентным нулю и единице, например тайским или тамильским цифрам. Номер выводится с помощью того же семейства цифр, которое используется в лексемах форматирования.
a	a, b, c, d, ... x, y, z, aa, ab, ac ...
A	A, B, C, D, ... X, Y, Z, AA, AB, AC ...
i	i, ii, iii, iv, ... x, xi, xii, xiii, xiv, ...
I	I, II, III, IV, ... X, XI, XII, XIII, XIV, ...

Спецификация не определяет эти последовательности подробно, например, там не говорится, как нужно представлять римскими цифрами числа больше 1000 (сами римляне имели различные соглашения, например, ставить над символом горизонтальную черту или рамку вокруг символа, – таких эффектов было бы трудно достичь в выводе XML).

Атрибуты `grouping-separator` и `grouping-size` могут использоваться для контроля над разделением групп цифр. Например, установка «`grouping-separator=" "`» (один пробел), а «`grouping-size="2"`» заставила бы номер 12345 выводиться как «`1 23 45`». При формировании групп отсчет цифр всегда ведется с правой стороны.

Для других лексем форматирования спецификация XSLT не дает четких предписаний. В ней говорится, что можно использовать любую лексему форматирования для указания последовательности, начинающейся с этой лексемы, при условии, что используемый процессор XSLT поддерживает такую последовательность. Если же последовательность не поддерживается, номер будет форматироваться с использованием лексемы формата «1». Так, например, если процессор поддерживает последовательность нумерации «`eins, zwei, drei, vier`», то можно вызывать эту последовательность, используя лексему формата «`eins`»; если он поддерживает последовательность нумерации «α, β, γ, δ», то можно вызывать ее с помощью лексемы формата «α».

На случай, когда лексема формата не идентифицирует однозначно последовательность нумерования, имеются два атрибута, которые обеспечивают больший контроль.

- Атрибут `lang` предназначен для указания целевого языка: например, последовательность, начинающаяся с «a» различна для английского языка («lang="en"») и для шведского («lang="se"»). Код языка задается в том же виде, как и в атрибуте `xml:lang`, определенном в спецификации XML.
- Атрибут `letter-value` предназначен для языков типа иврита, которые имеют несколько возможных последовательностей, начинающихся с одной и той же лексемы. Двумя допустимыми значениями являются алфавитное «alphabetic» и традиционное «traditional».

Детальное действие этих атрибутов полностью зависит от реализации, поэтому нельзя надеяться, что разные программы обязательно будут вести себя одинаково.

Все атрибуты, контролирующие форматирование, являются шаблонами значения атрибута, поэтому их можно параметризовать при помощи выражений, заключенных в фигурные скобки. Это полезно, главным образом, когда требуется выбрать из файла локализации значения на предпочтительном для текущего пользователя языке. Для этого можно воспользоваться теми же методами, что и для локализации сообщений: см. раздел <xsl:message> на стр. 281.

Вывод номера

Заключительное действие `<xsl:number>` – записать сгенерированную строку как текстовый узел в текущее назначение вывода.

Если с номером требуется сделать что-то еще (возможно, записать его как атрибут или скопировать в заголовок каждой страницы), можно сохранить его как значение переменной:

```
<xsl:variable name="номер-раздела"><xsl:number/></xsl:variable>
```

Присваивание значения номера переменной позволяет также производить дальнейшие манипуляции. Например, если требуется использовать традиционную последовательность нумерования сносок (*, †, ‡, §, ¶), этого нельзя сделать прямо в `<xsl:number>`, потому что эти символы – символы пунктуации, а не алфавитно-цифровые. Однако здесь можно использовать стандартную десятичную нумерацию, а затем конвертировать ее:

```
<xsl:template match="footnote"/>
<xsl:variable name="footnote-number">
   <xsl:number level="any" from="section"/>
</xsl:variable>
<xsl:value-of select="translate($footnote-number, '12345', '*†‡§¶')"/>
```

На практике надежнее было бы использовать для этих специальных символов ссылки на символы, чтобы избежать искажения их текстовым редактором, который не понимает Unicode. Функция `translate()` замещает символы,

указанные в ее втором параметре, на соответствующие символы из третьего параметра: это описано в главе 7.

Здесь остался незатронутым сложный вопрос о том, что когда требуется, чтобы нумерация сносок начиналась заново на каждой странице, их невозможно расставить, пока весь документ не разбит на страницы. Некоторые виды нумерования, действительно, подвластны скорее языку XSL Formatting, чем языку XSL Transformations.

Использование и примеры

Хотя правила для <xsl:number> не очень детализованы и иногда запутанны, наиболее типичные применения этой инструкции достаточно просты.

Общие правила позволяют нумерование узлов любых типов, но на практике инструкция <xsl:number> почти всегда используется для нумерования элементов. В этом разделе также предполагается, что текущий узел – это узел элемента.

level="single"

Эта опция (значение по умолчанию) используется для нумерования одноуровневых элементов.

Простейший способ использования <xsl:number> – вообще без атрибутов:

```
<xsl:number/>
```

Если текущий элемент – восьмой дочерний элемент <item> своего родительского элемента, то указанный выше код запишет в текущем адресате вывода текстовое значение «8». Формально процессор считает все элементы, которые соответствуют образцу в атрибуте count, а значение по умолчанию для атрибута count в этом случае – образец, соответствующий элементам <item>.

Часто для такого простого вида нумерования лучше использовать функцию position(), особенно когда требуется пронумеровать много узлов. Причина в том, что в случае стандартной реализации каждый узел, пронумерованный с помощью <xsl:number>, вызовет подсчет предшествующих ему одноуровневых элементов, что с увеличением их количества будет занимать все более и более длительное время. С функцией position() больше вероятности, что система будет знать позиции элементов и не должна будет совершать многократные проходы по дереву и сопоставления с образцом. Конечно, это поможет только в тех случаях, когда «position()» и <xsl:number/> дают один и тот же результат, то есть когда обрабатываемый список текущего узла состоит только из одноуровневых элементов определенного типа.

Другой возможностью для нумерования является использование функции count(), например: «count(preceding-sibling::item)+1». Это часто удобнее, если номера нужны в дальнейшей обработке, а не для форматирования их для вывода.

Атрибут count элемента <xsl:number> можно использовать двумя способами.

Во-первых, он полезен, если нужно пронумеровать несколько различных типов одноуровневых элементов. Например, есть элемент, имеющий в качестве непосредственных потомков элементы <предмет> и элементы <специальный-предмет>:

```
<список-покупок>
   <предмет>бананы</предмет>
   <предмет>яблоки</предмет>
   <специальный-предмет>цветы для бабушки</специальный-предмет>
   <предмет>виноград</предмет>
   <специальный-предмет>шоколад для тети Мод</специальный-предмет>
   <предмет>вишня</предмет>
</список-покупок>
```

Если требуется пронумеровать их в одной последовательности, можно сделать так:

```
<xsl:template match="предмет | специальный-предмет">
   <xsl:number count="предмет | специальный-предмет"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="."/><br/>
</xsl:template>
```

что при обработке элемента <список-покупок> приведете к следующему выводу:

```
1 бананы<br/>
2 яблоки<br/>
3 цветы для бабушки<br/>
4 виноград<br/>
5 шоколад для тети Мод<br/>
6 вишня<br/>
```

В этом случае тот же результат можно получить, используя «count="*"». Если бы атрибут count был опущен, вывод был бы таким:

```
1 бананы<br/>
2 яблоки<br/>
1 цветы для бабушки<br/>
3 виноград<br/>
2 шоколад для тети Мод<br/>
4 вишня<br/>
```

потому что каждый элемент здесь нумеровался только с учетом элементов того же типа.

Еще одно применение атрибута count – указание, что должен определяться порядковый номер не текущего узла, а его родителя. Например, в шаблонном правиле для элемента <заголовок> можно с помощью <xsl:number> определить номер раздела, которому принадлежит этот заголовок:

```
<xsl:template match="заголовок">
   <xsl:number count="раздел"/>
. . .
</xsl:template>
```

Это применение менее типично: в большинстве случаев номер раздела проще вывести из шаблонного правила, которое обрабатывает непосредственно элемент <заголовок>.

Атрибут from редко нужен при «level="single"». Фактически трудно привести пример, который не был бы надуманным.

Если нужна нумерация, начинающаяся не с единицы или продолжающаяся с приращением, отличным от единицы, можно фиксировать результат <xsl:number> в переменной и оперировать с ним, используя XPath-выражения. Например, следующее шаблонное правило нумерует пункты в списке, начиная с номера, указанного в параметре:

```
<xsl:template match="предмет">
   <xsl:param name="начальное-значение" select="1"/>
   <xsl:variable name="число"><xsl:number/></xsl:variable>
   <xsl:value-of select="$начальное-значение + $число - 1"/>
. . .
</xsl:template>
```

level="any"

Эта опция полезна, когда нужно, независимо от позиции в иерархической структуре документа, пронумеровать объекты, которые имеют собственную нумерацию. Примерами могут служить схемы и иллюстрации, таблицы, уравнения, сноски, а также мероприятия в рамках различных собраний.

Атрибут count в этом случае можно оставить в значении по умолчанию. Например, для нумерования цитат в документе можно написать такое шаблонное правило:

```
<xsl:template match="цитата">
<table><tr><td width="90%" valign="top">
   <i><xsl:value-of select="."/></i></td>
<td><xsl:number level="any"/></td>
</tr></table>
</xsl:template>
```

Опять же, атрибут count пригодится, когда в одну последовательность нумерации входит несколько различных типов элементов, например, в последовательность, которая включает и диаграммы, и фотографии.

Заметьте, что каждое вычисление <xsl:number> вполне независимо от всех предыдущих вычислений. На результат влияет только относительное положение текущего элемента в исходном документе, а не то, сколько раз вычислялся элемент <xsl:number>. Таким образом, нет гарантии, что номера в документе вывода будут последовательными. Фактически, если порядок вывода отличается от порядка ввода, то номера в выводе определенно не будут последовательными. Если нужна нумерация, основанная на позиции объектов в документе вывода, можно получить ее, используя функцию position(). Если это дает неадекватный результат, можно выполнить второй проход, чтобы добавить порядковые номера.

Средства рабочего проекта XSLT 1.1 позволяют делать это, записывая результат первого прохода во временное дерево. В следующем примере из документа извлекаются все элементы <элемент-глоссария>, сортируются в алфавитном порядке и нумеруются в порядке их вывода. Сохранение временного дерева обеспечивает переменная глоссарий. Для достижения того же эффекта с процессором XSLT 1.0 нужно будет использовать специальную функцию расширения от поставщика node-set().

Вообразите исходный документ, который содержит определения глоссария, рассеянные по всему документу:

```
<элемент-глоссария>
     <термин>XML</термин>
     <определение>Расширяемый язык разметки</определение>
</элемент-глоссария>
```

Соответствующий шаблон выглядит так:

```
<xsl:template name="создать-глоссарий">
  <xsl:variable name="глоссарий">
    <xsl:for-each select="//элемент-глоссария">
    <xsl:sort select="термин"/>
      <xsl:copy-of select="."/>
    </xsl:for-each>
  </xsl:variable>
  <table>
    <xsl:for-each select="$глоссарий/элемент-глоссария">
     <tr>
       <td><xsl:number format="[1]"/></td>
       <td><xsl:value-of select="термин"/></td>
       <td><xsl:value-of select="определение"/></td>
     </tr>
    </xsl:for-each>
  </table>
</xsl:template>
```

В этом примере, правда, номера можно было с тем же успехом сгенерировать на первом проходе, используя функцию position().

С помощью атрибута from удобно указывать, где нумерация должна перезапуститься:

```
<xsl:template match="сноска">
   <xsl:number level="any" from="глава"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="."/>
</xsl:template>
```

Приведенный выше код пронумеровал бы сноски последовательно в пределах главы, снова начиная с 1 для каждой последующей главы.

Пример: Нумерование строк стихотворения

В следующем примере нумеруются строки стихотворения, причем в выводе справа от строки отображается номер каждой третьей строки. Здесь предполагается, что структура ввода содержит элемент <стихотворение>, элемент <строфа> и элемент <строка>: строки должны быть пронумерованы по всей поэме, а не в пределах каждой строфы.

Исходный файл

Таблица стилей может использоваться с исходным файлом стих.xml, используемым в главе 1.

Таблица стилей

Эта таблица стилей – стих.xsl. В ней для получения номера каждой строки используется <xsl:number>, но в выводе отображает его только каждая третья строка, что достигается с помощью оператора «mod», вычисляющего остаток от деления номера строки на три.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html><body>
   <p><xsl:apply-templates select="/стихотворение/строфа"/></p>
   </body></html>
</xsl:template>

<xsl:template match="строфа">
   <p><table><xsl:apply-templates/></table></p>
</xsl:template>

<xsl:template match="строка">
   <tr>
   <td width="350"><xsl:value-of select="."/></td>
   <td width="50">
      <xsl:variable name="номер-строки">
         <xsl:number level="any" from="стихотворение"/>
      </xsl:variable>
      <xsl:if test="$номер-строки mod 3 = 0">
         <xsl:value-of select="$номер-строки"/>
      </xsl:if>
   </td>
   </tr>
</xsl:template>
</xsl:stylesheet>
```

Вывод

Рис. 4.6. *Полученный вывод в окне броузера*

level="multiple"

Эта опция обычно используется для выведения иерархических порядковых номеров, часто встречающихся в технических или юридических документах, например 1.12.3 или A2(iii).

Следует отметить, что альтернативный способ получения таких номеров заключается в использовании нескольких вызовов <xsl:number> с «level="single"» и различными атрибутами count, например:

```
<xsl:number count="глава"/>.<xsl:number count="раздел"/>
   (<xsl:number count="пункт"/>)
```

Другая методика, которая может работать несколько быстрее, состоит в вычислении один раз номера главы и подстановке его как параметра для шаблона, обрабатывающего раздел. После этого и номер главы, и номер раздела вводятся как параметры для шаблона, обрабатывающего каждое предложение.

Однако использование «level="multiple"» удобнее, и в некоторых случаях (особенно с рекурсивными структурами, где элементы <раздел> содержатся в пределах других элементов <раздел>) могут быть единственным способом достижения требуемого эффекта.

Атрибут count определяет, какие родительские элементы должны быть включены. Обычно это выражается через образец объединения:

```
<xsl:template match="пункт">
   <xsl:number
      format="1.1.1. "
```

```
        level="multiple"
        count="глава | раздел | пункт"/>
    <xsl:apply-templates/>
</xsl:template>
```

Эффект применяемых правил в том, что будет формироваться составной порядковый номер, содержащий один компонент номера для каждого родителя (или самого элемента), которыми являются <глава>, <раздел> или <пункт>. Если структура регулярна и главы, разделы и пункты вложены аккуратно, то перед каждым пунктом будет выводиться номер типа 1.13.5, где 1 – номер главы, 13 – номер раздела в главе, а 5 – номер пункта в разделе.

Если структура нерегулярна: например, если есть разделы, которые не принадлежат главе, или если есть и пункты, и разделы как элементы одного уровня, или если есть разделы, вложенные в другие разделы, то результаты могут быть неожиданными, хотя внимательный анализ правил должен объяснить происходящее.

Иногда возникает проблема, которая состоит в том, что нумерование является контекстно-зависимым. Например, в пределах регулярной главы пункты имеют формат номера 1.2.3, а в приложении – A.2.3. Этого эффекта можно добиться, используя тот факт, что образец формата является шаблоном значений атрибутов. Можно, например, написать:

```
<xsl:template match="пункт">
    <xsl:variable name="формат">
        <xsl:choose>
            <xsl:when test="ancestor::глава">1.1.1. </xsl:when>
            <xsl:otherwise>A.1.1 </xsl:otherwise>
        </xsl:choose>
    </xsl:variable>
    <xsl:number
        format="{$формат}"
        level="multiple"
        count="приложение | глава | раздел | пункт"/>
    <xsl:apply-templates/>
</xsl:template>
```

См. также

Функция `count()` в главе 7

Функция `position()` в главе 7

Функция `format-number()` в главе 7

`<xsl:decimal-format>` на стр. 226

xsl:otherwise

Элемент <xsl:otherwise> используется в пределах инструкции <xsl:choose> для указания действия, которое должно быть предпринято, когда не удовлетворяется ни одно из условий <xsl:when>.

Определен в

XSLT, раздел 9.2

Формат

```
<xsl:otherwise>
   тело-шаблона
</xsl:otherwise>
```

Расположение

<xsl:otherwise> может встречаться только как непосредственный потомок элемента <xsl:choose>. Если он есть, то он должен быть последним прямым потомком элемента <xsl:choose>, и он не может появляться более одного раза.

Атрибуты

Никакие.

Содержимое

Тело шаблона.

Действие

Тело шаблона элемента <xsl:otherwise> применяется, если (и только в этом случае) вычисление ни одного из элементов <xsl:when> в содержащем их элементе <xsl:choose> не возвращает значение истина.

Использование и примеры

См. <xsl:choose> на стр. 213.

См. также

<xsl:choose> на стр. 213

<xsl:when> на стр. 381

xsl:output

Элемент <xsl:output> – элемент верхнего уровня, используемый для контроля над форматом вывода таблицы стилей. Концептуально таблица стилей XSLT обрабатывается в две стадии: на первой стадии должно формироваться конечное дерево, а на второй конечное дерево должно быть записано в последовательный выходной файл. Элемент <xsl:output> контролирует именно вторую стадию, которую часто называют *сериализацией*.

Эта вторая стадия обработки – сериализация дерева в выходной документ – не является обязательной для XSLT-процессора; стандарт позволяет процессору делать дерево доступным и другими способами, например через DOM API. Процессору, который не записывает дерево в выходной файл, разрешается игнорировать этот элемент. Некоторые процессоры, производящие сериализацию, также могут заменить определения, имеющиеся в этом элементе, на параметры, установленные через API при вызове процессора. Подробнее о TrAX API см. в приложении F.

Определен в

XSLT, раздел 16

Формат

```
<xsl:output
   method= { "xml" | "html" | "text" | ПолноеИмя }
   version= { ЛексемаИмени }
   encoding= { строка }
   omit-xml-declaration= { "yes" | "no" }
   standalone= { "yes" | "no" }
   doctype-public= { строка }
   doctype-system= { строка }
   cdata-section-elements= { список-ПолныхИмен }
   indent= { "yes" | "no" }
   media-type= { строка } />
```

Расположение

`<xsl:output>` – элемент верхнего уровня, то есть он должен быть непосредственным потомком элемента `<xsl:stylesheet>`. В таблице стилей он может встречаться любое число раз.

Атрибуты

В XSLT 1.0 все атрибуты элемента `<xsl:output>` должны быть написаны буквально: невозможно использовать шаблоны значений атрибутов для их параметризации.

Это положение изменено в рабочем проекте спецификации XSLT 1.1, которая разрешает записывать все атрибуты `<xsl:output>` как шаблоны значений атрибутов, то есть как XPath-выражения, заключенные в фигурные скобки. Это позволяет вводить значения в таблицу стилей в виде параметров или определять их при разборе исходного документа. Правила, приведенные ниже в таблице, относятся к значению атрибута после раскрытия любого шаблона значения атрибута.

Содержимое

Никакого; элемент всегда пуст.

Действие

В таблице стилей может быть более одного элемента <xsl:output>. Если их несколько, то атрибуты, определяемые ими, в действительности как бы объединяются в единый концептуальный элемент <xsl:output> следующим образом:

- Для атрибута cdata-section-elements списки полных имен отдельных элементов <xsl:output> объединяются, при этом имя элемента, присутствующее в каком-либо из списков, обрабатывается как элемент секции CDATA.
- Для всех других атрибутов элемент <xsl:output>, в котором определено значение для атрибута, имеет приоритет над элементом, использующим значение атрибута по умолчанию. Если несколько элементов <xsl:output> определяют значение для атрибута, то используется значение с самым высоким преимуществом импортирования. Если это оставляет более одного значения (и даже если они идентичны), XSLT-процессор может или сообщить об ошибке, или использовать то значение, которое в таблице стилей встречается последним.

Атрибут method контролирует формат вывода, а это, в свою очередь, воздействует на указанные значения и значения по умолчанию других атрибутов.

В спецификации определены три выходных формата: «xml», «html» и «text». Альтернативно, выходной формат может быть задан как полное имя, которое должно включать непустой префикс, идентифицирующий пространство имен, действующее в текущий момент. Эта опция введена для расширений поставщика, и значение не определено в стандарте. Определенный поставщиком формат может давать свою собственную интерпретацию значений других атрибутов элемента <xsl:output>, а также он может определять для элемента <xsl:output> дополнительные атрибуты, при условии, что они не входят в пространство имен по умолчанию.

Если атрибут method опущен, вывод будет в формате XML, если конечное дерево не выглядит как явный HTML. Конечное дерево опознается как HTML, если:

- Корневой узел имеет, по крайней мере, один дочерний элемент, и
- Первый дочерний элемент корневого узла называется <html> (в любой комбинации верхнего и нижнего регистра) и имеет пустой URI пространства имен, и
- Перед элементом <html> нет никаких текстовых узлов, кроме (необязательно) текстовых узлов, состоящих только из пробельных символов.

Правила для вывода XML

Когда метод вывода – «xml», выходной файл обычно будет правильно построенным XML-документом, но фактическое требование – только чтобы это была корректная внешняя общая анализируемая сущность XML. Другими словами, это должно быть что-то, что могло бы быть встроено в XML-документ с

помощью ссылки на сущность типа «&doc;». В следующем примере показана корректная внешняя общая анализируемая сущность, которая не является корректным документом:

```
A <b>bold</b> and <emph>emphatic</emph> statement
```

Спецификация несколько неоднозначна в этой области. Хотя там говорится, что выводом всегда будет корректная внешняя общая анализируемая сущность, контекст проясняет, что это может включать также такие вещи, как объявление standalone-документа и объявление типа документа, которые, согласно синтаксису XML, могут появляться только в сущности документа, а не во внешней общей анализируемой сущности.

Пример корректного документа, который не является корректной внешней общей анализируемой сущностью, такой:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<p>A <b>bold</b> and <emph>emphatic</emph> statement</p>
```

Правила для сущностей документа и внешних общих анализируемых сущностей перекрываются, как показано на следующей диаграмме:

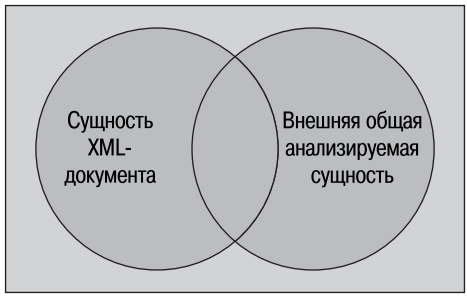


Рис. 4.7. *Перекрывание сущностей документа и внешних общих анализируемых сущностей*

По существу, таблица стилей XSLT может выводить все, что соответствует более темной области: все, что является корректной сущностью XML-документа, корректной внешней общей анализируемой сущностью или и тем и другим.

Ну, или почти все:

- Это должно также соответствовать Рекомендации по пространству имен
- Нет никаких явных условий для формирования внутреннего подмножества DTD, хотя этого можно с трудом достичь, генерируя текст и запрещая экранирование выводимых символов
- Аналогично нет никаких явных условий для формирования ссылок на сущности, хотя этого также можно достичь, генерируя текст и запрещая экранирование выводимых символов

В стандарте XML правило для внешней общей анализируемой сущности такое:

```
extParsedEnt ⇒ TextDecl ? content
```

в то же время правило для сущности документа выглядит следующим образом:

```
document ⇒ XMLDecl ?  Misc *  doctypedecl ?  Misc *  element  Misc *
```

где Misc разрешает пробельные символы, комментарии и инструкции обработки.

Таким образом, главные различия между этими двумя случаями следующие:

- TextDecl (объявление текста) – не совсем то же самое, что XMLDECL (объявление XML), как будет показано ниже.
- Документ может содержать doctypedecl (объявление типа документа), но внешняя общая анализируемая сущность – не должна. Объявление типа документа – это заголовок <!DOCTYPE ... >, задающий DTD и, возможно, включающий внутреннее подмножество DTD.
- Телом документа является element (элемент), в то время как тело внешней анализируемой сущности – content (содержимое). Здесь под содержимым имеется в виду практически все содержимое элемента, но без открывающего и закрывающего тегов.

TextDecl (объявление текста) на первый взгляд выглядит аналогично объявлению XML: например, <?xml version="1.0" encoding="utf-8"?> можно использовать и как объявление XML, и как объявление текста. Однако имеются различия:

- В объявлении XML атрибут version обязателен, а в объявлении текста он необязателен
- В объявлении XML атрибут encoding необязателен, а в объявлении текста он обязателен
- Объявление XML может включать атрибут standalone, а объявление текста – не может

Содержимое – это последовательность компонентов, включая дочерние элементы, символьные данные, ссылки на сущности, секции CDATA, инструкции обработки и комментарии, каждый из которых может встречаться любое число раз и в любом порядке.

Так что все следующие примеры – корректные внешние общие анализируемые сущности:

```
<реплика>Привет!</реплика>

<реплика>Привет!</реплика><реплика>До свидания!</реплика>

Привет!

<?xml version="1.0" encoding="utf-8"?>Привет!
```

Следующий пример – корректный XML-документ, но он *не* является корректной внешней общей анализируемой сущностью: и из-за атрибута `standalone`, и из-за объявления типа документа. Вывод ниже также допустим:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE реплика SYSTEM "hello.dtd">
<реплика>Привет!</реплика>
```

Следующий пример не является ни корректным XML-документом, ни корректной внешней общей анализируемой сущностью.

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE реплика SYSTEM "hello.dtd">
<реплика>Привет!</реплика>
<реплика>До свидания!</реплика>
```

Это не может быть XML-документом, потому что здесь более одного элемента верхнего уровня, но это не может быть и внешней общей анализируемой сущностью, потому что присутствует объявление `<!DOCTYPE>`. В документе, перечисляющем ошибки в XSLT 1.0, объясняется, что таблица стилей, производящая такой вывод, ошибочна.

В спецификации XSLT указаны также два других ограничения на форму вывода, хотя это скорее рекомендации для изготовителей, чем правила, которые прямо касаются авторов таблиц стилей. Это следующие правила:

- Вывод должен соответствовать правилам рекомендации по пространствам имен XML (XML Namespaces Recommendation). В отношении XML-документа значение этого достаточно ясно, но если речь идет просто о внешней сущности, необходимо некоторое дальнейшее пояснение. В стандарте оно есть: когда сущность помещается в документ путем заключения ее содержимого в теги элемента, результирующий документ должен соответствовать правилам пространства имен XML.
- Выходной файл должен точно отражать конечное дерево. В таком виде это требование звучит просто, но спецификация включает более официальную формулировку, которая удивительно сложна.

Есть разница в том, как формулируются правила относительно объявлений пространства имен в XSLT 1.0 и XSLT 1.1. В XSLT 1.0 генерирование объявлений пространств имен является задачей сериализатора. Это относится не только к узлам пространств имен, явно существующим в конечном дереве, но также и к любым пространствам имен, которые используются в конечном документе, но не представлены узлами пространств имен. Дело в том, что в правилах, например, нет ничего по поводу элементов `<xsl:element>` и `<xsl:attribute>`, что требует создания узлов пространств имен на дереве URI пространств имен, использованных в именах этих узлов. Однако в спецификации XSLT 1.1 описан процесс, названный записью пространств имен (namespace fixup), который гарантирует, что элемент в конечном дереве всегда имеет узел пространства имен для каждого пространства имен, используемого или в имени элемента, или в имени любого из его атрибутов. Благодаря этому создание этих объявлений пространств имен больше не является задачей

сериализатора. Поводом для такого изменения послужило то, что в XSLT 1.1 содержимое временных деревьев (включая их узлы пространств имен) становится видимым таблице стилей, поэтому правила нужно было урегулировать.

Хотя требуется, чтобы вывод был корректным XML, нет никаких правил о том, что это должен быть действительный XML (вспомните, что действительным XML-документом, грубо говоря, является XML-документ, который соответствует правилам в его собственном DTD). Если генерируется объявление типа документа, которое ссылается на конкретный DTD, не надейтесь, что XSLT-процессор проверит, что документ вывода фактически соответствует этому DTD; это – полностью ответственность автора таблицы стилей.

В «xml»-методе вывода другие атрибуты <xsl:output> интерпретируются следующим образом:

Атрибут	Интерпретация
version	Версия XML, используемая в выходном документе. В настоящее время единственная существующая версия XML – версия 1.0, но тут предусмотрена возможность появления других версий в будущем. Значением по умолчанию, и единственным значением, которое следует использовать в настоящее время, является «1.0».
encoding	Этот атрибут определяет предпочтительную кодировку символов для документа вывода. Все XSLT-процессоры должны поддерживать значения «UTF-8» и «UTF-16» (являющиеся также единственными значениями, которые должны поддерживать синтаксические анализаторы XML). Это имя кодировки используется в атрибуте кодировки XML или в объявлении Text в начале выходного файла, и все символы в файле будут закодированы, используя это соглашение. Стандарт имени кодировки не учитывает регистр. Если кодировка не позволяет прямого использования всех имеющихся символов XML, например кодировка «iso-8859-1», то символы, не входящие в эту кодировку, представляются, по возможности, ссылками на символы, используемыми в XML (например, «₤»). Будет ошибкой, если такие символы появятся в контекстах, где не приняты ссылки на символы (например, в пределах инструкции обработки или комментария, а также в имени элемента или атрибута).
indent	Если этот атрибут имеет значение «yes», это означает, что вывод XML должен иметь отступы, отражающие его иерархическую структуру. XSLT-процессор не обязан выполнять это требование, и в этом случае трудно предугадать точную форму вывода. Есть некоторые ограничения на способ выведения отступов. В действительности это можно делать путем добавления к дереву текстовых узлов, состоящих только из пробельных символов, и они не могут быть добавлены рядом с существующим текстовым узлом. Пробельные символы, уже имеющиеся в конечном дереве, не могут быть удалены. Так, если вывод уже содержит многострочные текстовые узлы, возможности эстетической правки вывода сужаются.

Атрибут	Интерпретация
	Заметьте, что даже при таких ограничениях добавление узлов пробельных символов к выводу может повлиять на способ их интерпретации. Это особенно касается моделей со смешанным содержимым, где элемент может иметь в качестве потомков как элементы, так и текстовые узлы.
`cdata-section-elements`	Это список имен элемента, где каждое имя выражено как полное имя и отделено от других пробельными символами. Любой префикс в полном имени обрабатывается обычным способом как ссылка на соответствующий URI пространства имен, использованный в объявлении пространства имен, действующего для фактического элемента `<xsl:output>`, в котором находится атрибут `cdata-section-elements`. Поскольку это имена элементов, если у имени нет префикса, считается, что оно принадлежит пространству имен по умолчанию. При выводе текстового узла, если имя родительского элемента данного узла есть в этом списке, то текстовый узел выводится как секция `CDATA`. Например, текстовое значение `«James»` выводится как `«<![CDATA[James]]>»`, а текстовое значение `«AT&T»` выводится как `«<![CDATA[AT&T]]>»`. В других случаях это значение, вероятно, выводилось бы как `«AT&amp;T»`. XSLT-процессор может выбрать другие эквивалентные представления, например ссылки на символы, но в стандарте указывается, что он не должен использовать CDATA, если это не запрашивается явным образом. (Однако обратите внимание на слово «должен»: оно означает правило, рекомендацию, а не обязательное требование соответствия.) Секция `CDATA` в случае необходимости разбивается на части: или когда в данных появляется терминатор последовательности `«]]>»`, или когда встречается символ, который может выводиться только с использованием ссылки на символ, так как он не поддерживается в выбранной кодировке.
`omit-xml-declaration`	Если этот атрибут имеет значение `«yes»`, XSLT-процессор не должен выводить XML-объявление (или, по обстоятельствам, объявление текста; вспомните, что XML-объявления используются в начале сущности документа, а объявления текста – в начале внешней общей анализируемой сущности). Если атрибут опущен или имеет значение `«no»`, то объявление должно выводиться. Объявление должно включать атрибуты и `version`, и `encoding` (чтобы гарантировать, что оно действительно и как объявление XML, и как объявление текста). Атрибут `standalone` оно должно включать только в том случае, если этот атрибут определен в элементе `<xsl:output>`.
`standalone`	Если этот атрибут имеет значение `«yes»`, то в XML-объявлении будет указано `«standalone="yes"»`. Если этот атрибут имеет значение `«no»`, то в XML-объявлении будет указано `«standalone="no"»`. Если атрибут опущен, то XML-объявление не будет включать атрибут `standalone`. Это сделает его действительным объявлением текста,

Атрибут	Интерпретация
	позволяя его использование во внешней общей анализируемой сущности. Этот атрибут не должен использоваться, если вывод не является правильно построенным XML-документом.
doctype-system	Если этот атрибут определен, выходной файл, должен включать объявление типа документа (то есть <!DOCTYPE>) после XML-объявления и открывающим тегом первого элемента. Имя типа документа будет такое же, как имя первого элемента. Значение этого атрибута используется как системный идентификатор в объявлении типа документа. Этот атрибут не должен использоваться, если вывод не является правильно построенным XML-документом.
doctype-public	Этот атрибут игнорируется, если не определен атрибут doctype-system. Он указывает значение открытого идентификатора, которое должно входить в объявление типа документа. Если не определен ни один открытый идентификатор, в объявлении типа документа не указывается никакой.
media-type	Этот атрибут определяет тип носителя выходного файла (часто называемый типом MIME). Значение по умолчанию – «text/xml». В спецификации не говорится, где можно применить эту информацию: она не влияет на содержимое выходного файла, но в зависимости от среды может повлиять на его имя, способ сохранения или способ передачи. Например, эта информация могла бы использоваться в заголовке HTTP-протокола.

Помните, что спецификации в элементе <xsl:output> будут эффективны, только если XSLT-процессор используется для сериализации XML. Если вывод преобразования записывается в DOM, а потом уже используется сериализатор, который имеется в реализациях DOM (например, используя метод save или свойство xml в случае реализации DOM от Microsoft), то спецификации <xsl:output> не дадут никакого эффекта.

Правила для вывода HTML

Когда атрибут method установлен в «html» или в значение по умолчанию, а конечное дерево признается представлением HTML, выводом будет файл HTML. По умолчанию он будет соответствовать HTML 4.0.

В спецификации XSLT не упоминается возможность создания вывода XHTML, что не удивительно, поскольку она была опубликована раньше спецификации XHTML. XHTML – это чистый XML, поэтому когда требуется получить XHTML, используйте метод вывода «xml». Некоторые поставщики приспособили вывод XML к XHTML, чтобы сделать его приемлемым для традиционных HTML-броузеров: например, генерируя пустые теги типа <hr /> с пробелом после имени элемента. Однако это – вне рамок стандарта XSLT.

HTML выводится таким же образом, как XML, кроме некоторых конкретных различий. Эти различия следующие:

- Некоторые элементы признаются как пустые элементы. Они опознаются в любой комбинации верхнего и нижнего регистра. Эти элементы выводятся с открывающим тегом, но без закрывающего. Для HTML 4.0 это следующие элементы:

```
<area>          <frame>         <isindex>
<base>          <hr>            <link>
<basefont>      <img>           <meta>
<br>            <input>         <param>
<col>
```

- Элементы <script> и <style> (снова, в любой комбинации верхнего и нижнего регистров) не требуют экранировать специальные символы. В текстовом содержимом этих элементов символ «<» будет выводиться как «<», а не как «<».
- Атрибуты HTML, значение которых – URI (например, атрибут href элемента <a> или атрибут src элемента <img>), признаются, а специальные символы в пределах URI – экранируются, как определено в технических требованиях HTML. В частности, символы, не являющиеся ASCII, в URI должны быть представлены путем преобразования каждого байта в представлении UTF-8 в «%HH», где HH – шестнадцатеричное представление байта.
- Специальные символы могут выводиться, используя ссылки на сущности типа «é», когда они определены в соответствующей версии HTML. Это – на усмотрение XSLT-процессора, он не обязан использовать эти имена сущностей.
- Инструкции обработки заканчиваются на «>», а не на «?>». Инструкции обработки не часто используются в HTML, но стандарт HTML 4.0 рекомендует, чтобы любые расширения поставщика были осуществлены таким способом, а не добавлением к языку тегов элемента. Так что, возможно, они будут чаще встречаться в будущем.
- Атрибуты, которые традиционно пишутся только с ключевым словом и без всяких значений, признаются и выводятся в такой же форме. Типичные примеры – <TEXTAREA READONLY> и <OPTION SELECTED>. Это сокращения, разрешенные в SGML, но не в XML, для атрибута, который имеет только одно разрешенное значение, причем такое же, как имя атрибута. В XML эти теги должны быть написаны как <TABLE BORDER="BORDER"> и <OPTION SELECTED="SELECTED">. Метод вывода HTML обычно использует сокращенный вариант, поскольку это единственная форма, которую признают ранние HTML-броузеры.
- Допустимо специальное использование символа амперсанда в динамических HTML-атрибутах. Например, тег <TD WIDTH="&{width};"> – корректный HTML, хотя из-за символа амперсанда он был бы некорректным в XML. Чтобы произвести этот вывод из конечного литерального элемента, тег в таблице стилей должен быть написан как <TD WIDTH="&{{width}};">; обратите внимание на двойные фигурные скобки – они предотвращают их интерпретацию в их особом значении в шаблонах значений атрибутов.

Общий источник неприятностей при выводе HTML – использование амперсандов в URL. Например, предположим, что нужно генерировать вывод:

```
<a href="http://www.acme.com/search.asp?product=widgets&country=spain>
Spanish Widgets
</a>
```

Фактически невозможно произвести это, используя стандартный XSLT: амперсанд всегда выйдет как «&». Причина этого проста: «&», хотя и широко используется и принимается, не является фактически корректным HTML, и согласно стандарту, он должен выводиться как «&». Все представительные броузеры принимают корректную выводимую форму, поэтому можно не волноваться на этот счет.

Когда выбирается вывод HTML, атрибуты элемента <xsl:output> интерпретируются следующим образом:

Атрибут	Интерпретация
version	Версия HTML, использованная в документе вывода. Поддерживаемая версия HTML зависит от конкретной реализации, хотя можно надеяться, что все реализации поддерживают версию по умолчанию – версию 4.0.
encoding	Этот атрибут определяет предпочтительную кодировку символов для документа вывода. Он используется для генерирования атрибута charset элемента <META>, вставляемого сразу после открывающего тега элемента <HEAD>, если он есть. Если кодировка не позволяет прямого использования всех имеющихся символов XML, например кодировка «iso-8859-1», то символы, не входящие в эту кодировку, представляются, по возможности, ссылками на сущности или на цифры. Процессор может использовать такие ссылки даже для символов, имеющихся в кодировке: например, неразрывный пробел может выводиться непосредственно (он выглядит точно так же, как обыкновенный пробел) или как « », «&#a0;» или « ». Будет ошибкой если символы, которые не могут быть представлены прямо, появятся в контекстах, где ссылки на символы не допускаются (например, в пределах элемента сценария, в пределах комментария или в имени элемента или атрибута).
indent	Если этот атрибут имеет значение «yes», это означает, что HTML-вывод должен иметь отступы, отражающие его иерархическую структуру. XSLT-процессор не обязан выполнять это требование, и в этом случае трудно предугадать точную форму вывода. При создании вывода с отступами процессор имеет намного больше свободы в добавлении или удалении пробельных символов, чем в случае XML, так как в HTML пробельные символы обрабатываются иначе. Процессор может добавлять или удалять пробельные символы где угодно, пока это не изменяет отображения HTML в броузере.
cdata-section-elements	Этот атрибут не применим к выводу HTML.

Атрибут	Интерпретация
omit-xml-declaration	Этот атрибут не применим к выводу HTML.
standalone	Этот атрибут не применим к выводу HTML.
doctype-system	Если этот атрибут определен, выходной файл будет включать объявление типа документа сразу перед открывающим тегом первого элемента. Имя типа документа будет «HTML» или «html». Значение этого атрибута используется как идентификатор system в объявлении типа документа.
doctype-public	Если этот атрибут определен, выходной файл будет включать объявление типа документа сразу перед открывающим тегом первого элемента. Имя типа документа будет «HTML» или «html». Значение этого атрибута используется как идентификатор public в объявлении типа документа.
media-type	Этот атрибут определяет тип носителя выходного файла (часто называемый типом MIME). Значение по умолчанию – «text/xml». В спецификации не говорится, где можно применить эту информацию: она не влияет на содержимое выходного файла, но в зависимости от среды может повлиять на его имя, способ сохранения или способ передачи. Например, эта информация могла бы использоваться в заголовке HTTP-протокола.

Правила для текстового вывода

Когда «method="text"», конечное дерево выводится как простой текстовый файл. Значения текстовых узлов дерева копируются в вывод, а все другие узлы игнорируются. В пределах текстовых узлов все символьные значения выводятся, используя соответствующую кодировку, которая определена атрибутом encoding; нет никаких специальных символов типа «&», которые нужно экранировать.

Способ вывода окончаний строк (например, LF или CRLF) не определен; процессор может использовать заданные по умолчанию соглашения для концов строк той платформы, на которой он работает.

Атрибуты, которые применяются для текстового вывода, перечислены ниже. Все другие атрибуты игнорируются.

Атрибут	Интерпретация
encoding	Этот атрибут определяет предпочтительную кодировку символов для документа вывода. Значение по умолчанию зависит от реализации, а также от платформы, на которой работает процессор. Если кодировка не позволяет прямого использования всех имеющихся символов XML, например кодировка «iso-8859-1», то любой символ вне этого подмножества будет вызывать сообщение об ошибке. Этот атрибут определяет тип носителя выходного файла (часто называемый типом MIME). Значение по умолчанию – «text/plain». В специфи-

Атрибут	Интерпретация
media-type	кации не говорится, где можно применить эту информацию: она не влияет на содержимое выходного файла, но в зависимости от среды может повлиять на его имя, способ сохранения или способ передачи. Например, эта информация могла бы использоваться в заголовке HTTP-протокола.

Использование

Механизм настроек по умолчанию гарантирует, что обычно не нужно включать элемент <xsl:output> в таблицу стилей. По умолчанию метод вывода XML используется, если первый выводимый объект – не элемент <HTML>, присутствие которого подразумевает метод вывода HTML.

От элемента <xsl:output> зависит, каким образом конечное дерево превращается в выходной файл. Если XSLT-процессор позволяет делать с конечным деревом что-то еще, например, передает его приложению как Document DOM или как поток событий SAX, то элемент <xsl:output> не имеет значения.

Атрибут encoding может быть очень полезен для гарантии, что выходной файл может быть легко просмотрен и отредактирован. К сожалению, тем не менее, набор возможных значений различен в разных реализациях XSLT и может также зависеть от платформы. Например, многие XSLT-процессоры написаны на языке Java и используют средства Java для кодирования выходного потока, но набор кодировок, поддерживаемых различными реализациями виртуальной машины Java, различен. Однако поддержка кодировки iso-8859-1 довольно универсальна, поэтому при возникновении затруднений с просмотром выходного файла из-за наличия в нем символов UTF-8 Unicode, часто помогает переключение кодировки на iso-8859-1.[1]

> ***Если таблица стилей генерирует символы с ударением или другие специальные символы, и кажется, что они выходят неправильно в выводе, то возможно, что они правильно представляются в UTF-8, но искажаются при просмотре в текстовом редакторе, который не понимает UTF-8.***

Атрибут encoding определяет, каким образом XSLT-процессор сериализует вывод в поток байтов, но он не имеет отношения к тому, что случается с байтами впоследствии. Если процессор записывает в файл, вероятно, файл будет записан в выбранной кодировке. Но если к выводу обращаются через API как к строке символов, или он записывается в текстовом поле в базе данных, кодировка символов может быть изменена прежде, чем пользователь увидит вывод. Классический пример такого эффекта – интерфейс transformNode() от Microsoft (см. приложение А), который возвращает результат преобразования как строку типа BSTR. Поскольку это BSTR, она будет всегда кодироваться

[1] Последнее утверждение неприменимо к русскому языку. Для русского языка нет другой общепринятой кодировки, которая поддерживалась бы большинством XSLT-процессоров, кроме UTF-8. – *Примеч. науч. ред.*

в UTF-16, независимо от запрашиваемой кодировки. То же самое происходит с интерфейсом TrAX (см. приложение F); если используется `StreamResult`, основанный на `Writer`, то кодировка в этом случае зависит от того, как конкретный `Writer` кодирует символы Unicode, а XSLT-процессор не имеет над этим никакого контроля.

Примеры

Следующий пример запрашивает вывод XML, используя кодировку iso-8859-1. Для читабельности вывод имеет отступы, а содержание элементов `<script>` из-за наличия в них множества специальных символов выводится как секция CDATA. Выходной файл ссылается на DTD `книги.dtd`: заметьте, что это полностью на совести пользователя – гарантировать, что вывод таблицы стилей фактически соответствует этому DTD и что это, действительно, правильно построенный XML-документ.

```
<xsl:output
   method="xml"
   indent="yes"
   encoding="iso-8859-1"
   cdata-section-elements="script"
   doctype-system="книги.dtd" />
```

Следующий пример может использоваться, если вывод таблицы стилей – файл с разделителями-запятыми, использующий только символы US ASCII:

```
<xsl:output
   method="text"
   encoding="us-ascii" />
```

xsl:param

Элемент `<xsl:param>` используется или на верхнем уровне, описывая глобальный параметр, или непосредственно в пределах элемента `<xsl:template>`, описывая локальный параметр для шаблона. Он определяет имя параметра и значение по умолчанию, которое используется, если вызывающая программа не указывает никакого значения для параметра.

Определен в

XSLT, раздел 11

Формат

```
<xsl:param name=ПолноеИмя select=Выражение >
   тело шаблона
</xsl:param>
```

Расположение

<xsl:param> может встречаться как элемент верхнего уровня (прямой потомок элемента <xsl:stylesheet>) или как прямой потомок элемента <xsl:template>. В последнем случае элементы <xsl:param> должны появляться перед любыми другими дочерними элементами.

Атрибуты

Имя	Значение	Назначение
name обязательный	Полное имя	Имя параметра
select необязательный	Выражение	Значение по умолчанию параметра, если никакое явное значение не указано при вызове

Конструкции полное имя и выражение определены в главе 5.

Содержимое

Тело шаблона (необязательно). Если присутствует атрибут select, элемент должен быть пуст.

Действие

Элемент <xsl:param> на верхнем уровне таблицы стилей объявляет глобальный параметр; элемент <xsl:param>, встречающийся как непосредственный потомок элемента <xsl:template>, объявляет локальный параметр для этого шаблона.

Элемент <xsl:param> определяет имя параметра и его значение по умолчанию. Значение по умолчанию используется, если только никакое другое значение не задано явным образом при вызове.

Явное значение может быть задано для локального параметра с помощью элемента <xsl:with-param>, когда шаблон вызывается через <xsl:apply-templates>, <xsl:call-template> или (только в XSLT 1.1) <xsl:apply-imports>.

Способ задания явных значений глобальных параметров определяется поставщиком (например, они могут задаваться из командной строки или через переменные среды или могут вводиться через определенный поставщиком API). Подробнее об этом говорится в соответствующих приложениях по конкретным продуктам и в приложении F – относительно TrAX API.

Значение параметра

Значение по умолчанию параметра можно давать с помощью выражения в атрибуте select или через содержимое тела шаблона. Если есть атрибут select, элемент <xsl:param> должен быть пуст. Если атрибута select нет и тело шаблона пусто, то значение параметра по умолчанию – пустая строка.

Если значение задается выражением, оно будет представлено логическим типом данных, численным типом, строковым типом или набором узлов, в зависимости от выражения. Если значение задается непустым телом шаблона, то будет создано временное дерево. В XSLT 1.0 тип данных значения – фрагмент конечного дерева; в XSLT 1.1 это будет набор узлов, содержащий корень этого дерева в качестве его единственного элемента. Все это аналогично элементу <xsl:variable>, описанному на стр. 370.

Имя параметра

Имя параметра определяется полным именем. Обычно это простое имя (типа «номер» или «список-имен»), но также оно может быть уточнено префиксом: например, «my:value». Если имя имеет префикс, этот префикс должен соответствовать пространству имен, которое действует в той точке в таблице стилей. Истинное имя параметра – для проверки наличия дублирующих имен – определяется не префиксом, а URI пространства имен, соответствующим префиксу, так что две переменные, «my:value» и «your:value», имеют одинаковые имена, если префиксы «my» и «your» относятся к одному и тому же URI пространства имен. Если имя не имеет никакого префикса, оно трактуется подобно имени атрибута: то есть оно имеет пустой URI пространства имен, так как оно не использует URI пространства имен по умолчанию.

На имя параметра ссылаются так же, как на переменные, предваряя его имя знаком доллара (например, «$num»), а все правила по уникальности и области действия имен такие, как если бы вместо элемента <xsl:param> был элемент <xsl:variable>. Единственная разница между параметрами и переменными – в том, как они получают начальное значение.

Использование

Глобальные параметры особенно полезны для выбора частей исходного документа, подлежащих обработке. Общий подход заключается в том, что XSLT-процессор выполняется в пределах веб-сервера. Запрос пользователя принимает Java Server Page или Java-сервлет, или фактически ASP-страница. Параметры запроса будут приняты, а обработка таблицы стилей начнется с использования API, определяемого каждым поставщиком. Обычно этот API обеспечивает некоторый способ передачи параметров, которые исходили из запроса HTTP, в таблицу стилей в качестве начальных значений глобальных элементов <xsl:param>.

Если API поддерживает это, глобальный параметр может принимать любое значение: строковое, численное, логическое, набор узлов или внешний объект. Поскольку это может быть набор узлов, что дает еще один способ поставки вторичных исходных документов для использования таблицей стилей, этот способ является альтернативой функции document(). Многие продукты позволяют поставлять такой параметр в виде объекта Document DOM.

Локальные параметры используются чаще с <xsl:call-template>, чем с <xsl:apply-templates>, хотя они доступны обоим, а в XSLT 1.1 они могут так-

же использоваться с <xsl:apply-imports>. Действительное значение параметра устанавливается вызывающей программой с помощью элемента <xsl:with-param>. Параметры часто необходимы рекурсивным алгоритмам, используемым в XSLT, для обработки списков пунктов: примеры таких алгоритмов приведены в разделе <xsl:call-template> на стр. 204.

Примеры

Пример: Использование <xsl:param> со значением по умолчанию

Исходный файл

Эта таблица стилей работает с любым исходным файлом XML.

Таблица стилей

Таблица стилей – вызов.xsl.

Она содержит именованный шаблон, который выводит глубину узла (определяемую как число родительских элементов). Узел может быть задан как параметр; если он не задан, то за параметр по умолчанию принимается текущий узел.

Таблица стилей включает шаблонное правило для корневого узла, которое вызывает этот именованный шаблон, используя значение параметра по умолчанию – отображать имя и глубину каждого элемента в исходном документе.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">
<xsl:for-each select="//*">
    <xsl:value-of select="concat(name(), ' -- ')"/>
    <xsl:call-template name="depth"/>;
</xsl:for-each>
</xsl:template>

<xsl:template name="depth">
    <xsl:param name="node" select="."/>
    <xsl:value-of select="count($node/ancestor::node())"/>
</xsl:template>
</xsl:transform>
```

Вывод

Если эту таблицу стилей запустить с файлом стих.xml, используемым в главе 1, вывод будет следующим:

```
стихотворение -- 1;
автор -- 2;
```

```
дата -- 2;
название -- 2;
строфа -- 2;
строка -- 3;
строка -- 3;
строка -- 3;
строка -- 3;
строфа -- 2;
строка -- 3;
строка -- 3;
строка -- 3;
строка -- 3;
строфа -- 2;
строка -- 3;
строка -- 3;
строка -- 3;
строка -- 3;
```

См. также

`<xsl:apply-templates>` на стр. 178

`<xsl:call-template>` на стр. 204

`<xsl:variable>` на стр. 370

`<xsl:with-param>` на стр. 383

xsl:preserve-space

Элемент `<xsl:preserve-space>` наряду с `<xsl:strip-space>` используется для управления обработкой узлов пробельных символов в исходном документе.

Определен в

XSLT, раздел 3.4

Формат

```
<xsl:preserve-space elements=список-КритериевИмен />
```

Расположение

`<xsl:preserve-space>` – элемент верхнего уровня, то есть он должен быть непосредственным потомком элемента `<xsl:stylesheet>`. Нет никаких ограничений на его расположение относительно других элементов верхнего уровня.

Атрибуты

Имя	Значение	Смысл
elements обязательный	Список критериев имен, разделенных пробельными символами	Определяет в исходном документе элементы, текстовые узлы которых, содержащие только пробельные символы, должны быть сохранены

Конструкция КритерийИмени (NameTest) определена в главе 5. Это может быть фактическое имя элемента, символ «*», означающий все элементы или конструкция «prefix:*», означающая все элементы в заданном пространстве имен.

Содержимое

Никакого; элемент всегда пуст.

Действие

Этот элемент, вместе с <xsl:strip-space>, определяет способ, которым обрабатываются текстовые узлы в исходном документе, состоящие только из пробельных символов. При отсутствии противоречий с элементом <xsl:strip-space> элемент <xsl:preserve-space> указывает, что текстовые узлы только из пробельных символов, являющиеся непосредственными потомками заданного элемента, должны быть сохранены в исходном дереве.

Сохранение текстовых узлов, состоящих только из пробельных символов, – действие, задаваемое по умолчанию, так что этот элемент должен использоваться только в случаях, когда необходимо нейтрализовать действие элемента <xsl:strip-space>. Взаимодействие этих двух элементов объясняется ниже.

Концепция текстовых узлов, состоящих только из пробельных символов, довольно подробно объясняется в главе 3.

Элемент <xsl:preserve-space> воздействует также на обработку текстовых узлов из пробельных символов в любом документе, загруженном с помощью функции document(). Ему не подчиняется обработка таких узлов в таблице стилей, используемой именно в роли таблицы стилей, но когда копия таблицы стилей загружается с помощью функции document(), она подчиняется тем же правилам, как любой другой документ.

Элемент не воздействует на узлы из пробельных символов в документах, возвращенных как результат функций расширения или введенных в таблицу стилей как значение глобального параметра. Кроме того, элемент не воздействует на то, что происходит с исходным документом до того, как он попадает в XSLT-процессор, поэтому если исходное дерево создается с помощью синтаксического анализатора XML, который зачищает узлы из пробельных символов (как делает это по умолчанию MSXML3 от Microsoft), а затем в таблице стилей вводится <xsl:preserve-space>, зачищенные узлы уже нельзя вернуть назад.

Текстовый узел только из пробельных символов – это текстовый узел, текст в котором целиком состоит из последовательности следующих символов: пробел, символ табуляции, символ конца абзаца и символ новой строки (#x20, #x9, #xD и #xA). Элемент <xsl:preserve-space> никак не действует на пробелы, имеющиеся в текстовых узлах, которые содержат не только пробельные символы; такие пробелы всегда сохраняются и являются частью значения текстового узла.

Прежде чем узел классифицируется как текстовый узел только из пробельных символов, дерево нормализуется, сцепляя все смежные текстовые узлы. Это включает объединение текстов, которые происходят из различных сущностей XML.

Текстовый узел только из пробельных символов может быть зачищен или сохранен. Если он зачищается, он удаляется из дерева. Это означает, что он никогда не будет сопоставляться, никогда не будет копироваться в вывод и никогда не будет подсчитываться при нумеровании узлов. Если узел сохраняется, он остается в дереве в его первоначальной форме, подверженный только нормализации концов строк, выполняемой синтаксическим анализатором XML.

Если текстовый узел только из пробельных символов имеет родителя с атрибутом xml:space, а самый близкий родитель с таким атрибутом имеет значение «xml:space="preserve"», то текстовый узел сохраняется независимо от элементов <xsl:preserve-space> и <xsl:strip-space> в таблице стилей.

Атрибут elements элемента <xsl:preserve-space> должен содержать список критериев имен, разделенных пробельными символами. Форма критерий имени (NameTest) определена на языке XPath-выражений; см. главу 5. Каждый критерий имени имеет соответствующий приоритет. Различные формы критериев имен и их значения следующие:

Синтаксис	Примеры	Значение	Приоритет
ПолноеИмя	title svg:width	Соответствует полному имени элемента, включая его URI пространства имен	0
ИмяБезДвоеточия «:*»	svg:*	Относится ко всем элементам в пространстве имен, URI которых соответствует данному префиксу	-0.25
«*»	*	Соответствует всем элементам	-0.5

Приоритет используется, когда возникают конфликты. Например, если таблица стилей определяет:

```
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="para clause"/>
```

тогда текстовые узлы только из пробельных символов, появляющиеся в пределах <para> или <clause>, будут сохранены. Хотя эти элементы соответствуют и <xsl:strip-space>, и <xsl:preserve-space>, критерий имени в последнем из них имеет более высокий приоритет (0 по сравнению с –0.5).

Элементы <xsl:strip-space> или <xsl:preserve-space>, содержащие несколько критериев имени, эквивалентны записи отдельного элемента <xsl:strip-space> или <xsl:preserve-space> для каждого критерия имени.

Текстовый узел только из пробельных символов сохраняется, если в таблице стилей нет никакого элемента <xsl:strip-space>, который соответствует его родительскому элементу.

Текстовый узел только из пробельных символов удаляется из дерева, если имеется элемент <xsl:strip-space>, который соответствует его родительскому элементу, но нет ни одного соответствующего элемента <xsl:preserve-space>.

Если родительскому элементу соответствуют и элемент <xsl:strip-space>, и элемент <xsl:preserve-space>, то решение зависит от соответствующих правил преимущества импортирования и приоритета. Принимая во внимание все элементы <xsl:strip-space> и <xsl:preserve-space>, соответствующие родительскому элементу текстового узла только из пробельных символов, XSLT-процессор выбирает из них элемент с самым высоким преимуществом импортирования (как определено в правилах для <xsl:import> на стр. 256). Если это дает более одного элемента, процессор выбирает из них элемент с самым высоким приоритетом, как определено в таблице выше. Если снова остается более одного элемента, процессор или сообщит об ошибке, или выберет элемент, который в таблице стилей встречается последним. Если выбранный элемент – <xsl:preserve-space>, то текстовый узел только из пробельных символов сохраняется в дереве; если это <xsl:strip-space> – узел удаляется из дерева.

При решении зачищать или сохранять текстовый узел только из пробельных символов, вышеупомянутые правила учитывают только его непосредственный родительский элемент. Правила для его других родительских элементов ничего не меняют. Сам элемент, конечно, никогда не удаляется из дерева: в процессе зачистки могут быть удалены только его текстовые узлы.

Если индивидуальный элемент имеет атрибут, определенный в XML, «xml:space="preserve"» или «xml:space="default"», это отменяет все, что определено в таблице стилей. Эти значения, в отличие от <xsl:preserve-space> и <xsl:strip-space>, относятся и к потомкам элемента, а не только к элементу, которому принадлежит атрибут. Если оказывается, что <xsl:strip-space> не производит никакого эффекта, одной из возможных причин может быть то, что в DTD рассматриваемый элемент объявлен как имеющий атрибут xml:space со значением «preserve» по умолчанию. Это невозможно отменить в таблице стилей.

Использование

Для многих категорий исходных документов, особенно используемых для представления структур данных, текстовые узлы только из пробельных символов не существенны, так что полезно указать:

```
<xsl:strip-space elements="*"/>
```

что удалит из дерева все подобные узлы. Зачистка этих нежелательных узлов дает два основных преимущества:

- Когда <xsl:apply-templates> используется со значением атрибута select по умолчанию, будут обрабатываться все дочерние узлы. Если текстовые узлы только из пробельных символов не зачищены, они также будут обрабатываться и, вероятно, будут скопированы в адресат вывода.
- Когда для определения положения элемента относительно одноуровневых элементов используется функция position(), текстовые узлы только из пробельных символов также включаются в подсчет. Это часто приводит к тому, что номера существенных узлов будут 2, 4, 6, 8...

В общем случае полезно зачищать текстовые узлы только из пробельных символов, принадлежащие элементам, которые имеют содержимое, то есть элементам, объявленным в DTD как содержащие дочерние элементы, но не #PCDATA.

Обычно не вредно также зачищать текстовые узлы только из пробельных символов у элементов, содержимое которых – #PCDATA; то есть у элементов, чьи потомки – только текстовые узлы. В большинстве случаев элемент, содержащий текст из одних пробельных символов, эквивалентен пустому элементу, так что логика таблицы стилей может быть упрощена, если элементы, содержащие только пробельные символы, нормализуются в пустые элементы путем удаления текстовых узлов.

В противоположность этому зачистка текстовых узлов только из пробельных символов у элементов со смешанным содержимым (элементов, объявленных в DTD как содержащие и дочерние элементы, и #PCDATA) – часто плохая идея. Например, рассмотрим такой элемент:

```
<quote>He went to <edu>Balliol College</edu> <city>Oxford</city> to read
<subject>Greats</subject></quote>
```

Пробел между элементом <edu> и элементом <city> – текстовый узел только из пробельных символов, и он должен быть сохранен, потому что иначе, когда теги удаляются приложением, которое заинтересовано только в тексте, слова «College» и «Oxford» будут рассматриваться как одно.

Примеры

Для зачистки узлов из пробельных символов у всех элементов исходного дерева:

```
<xsl:strip-space elements="*"/>
```

Для зачистки узлов из пробельных символов у выбранных элементов:

```
<xsl:strip-space elements="книга автор название цена"/>
```

Для зачистки узлов из пробельных символов у всех элементов, кроме элемента <описание>:

```
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="описание"/>
```

Для зачистки узлов из пробельных символов у всех элементов, кроме элементов в пространстве имен с URI http://mednet.org/text:

```
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="mednet:*" xmlns:mednet="http://mednet.org/text" />
```

См. также

<xsl:strip-space> на стр. 336

xsl:processing-instruction

Инструкция <xsl:processing-instruction> используется для записи узлов инструкций обработки в текущее назначение вывода.

Определен в

XSLT, раздел 7.3

Формат

```
<xsl:processing-instruction name=ИмяБезДвоеточия >
   тело шаблона
</xsl:processing-instruction>
```

Расположение

<xsl:processing-instruction> – инструкция, и всегда используется в пределах тела шаблона.

Атрибуты

Имя	Значение	Назначение
name обязательный	Шаблон значения атрибута, возвращающий ИмяБезДвоеточия	Имя (цель) генерируемой инструкции обработки

Содержимое

Тело шаблона.

Действие

Имя генерируемой инструкции обработки (в терминах XML – цель инструкции обработки, PITarget) определяется атрибутом name. Оно может быть выражено как шаблон значения атрибута. Имя должно быть действительным, поскольку цель, как определено в спецификациях XML и XSLT, налагает дополнительное условие, что это должно быть действительное ИмяБезДвоеточия (NCName), как это определено в рекомендации XML Namespaces Recommendation. Это означает, что должно оно быть XML-именем, которое не со-

держит двоеточие (чтобы быть именем без двоеточия, NCName) и которое не является именем «xml» в любом сочетании верхнего и нижнего регистров (чтобы быть целью инструкции обработки).

Из спецификаций вполне явно следует, что <xsl:processing-instruction> не может использоваться для генерирования XML-объявления в начале выходного файла. XML-объявление похоже на инструкцию обработки, но формально не является ею; и запрет на использование имени «xml» довольно явно подчеркивает это. XML-объявление в выходном файле автоматически генерируется XSLT-процессором и может, в ограниченной степени, контролироваться с помощью элемента <xsl:output>.

Данные инструкции обработки генерируются из тела шаблона. Пробел, отделяющий цель от данных, выводится автоматически. Вывод, генерируемый телом шаблона, должен содержать только текстовые узлы и не должен содержать строковый набор «?>», который завершает инструкцию обработки. В реализациях разрешается заменять «?>» на «? >» (с пробелом); к сожалению, также разрешается и сообщать об этом как об ошибке, поэтому если нужно сделать таблицу стилей переносимой, следует удостовериться, что такого не случается.

Данные инструкции обработки не могут содержать ссылки на символы (типа «₤»), поэтому будет ошибкой выводить любые символы, которые не могут быть непосредственно представлены в выбранной для выходного файла кодировке символов. Некоторые инструкции обработки могут принимать данные, которые выглядят как ссылки на символы, но приняты на уровне приложения и не имеют отношения к стандарту XML, поэтому XSLT-процессор никогда не будет генерировать такие ссылки автоматически.

Использование

Используйте эту инструкцию, когда нужно вывести инструкцию обработки.

Инструкции обработки не используются широко в большинстве приложений XML, поэтому, вероятно, инструкцию <xsl:processing-instruction> не придется использовать очень часто. В HTML она используются еще реже, хотя в HTML 4.0 рекомендовано, чтобы любые специализированные расширения осуществлялись с ее помощью. В HTML признаком конца инструкции обработки служит «>», а не «?>», и это отличие автоматически обрабатывается методом вывода HTML; см. раздел <xsl:output> на стр. 303.

Заметьте, что нельзя генерировать инструкцию обработки в выводе, записав инструкцию обработки в таблице стилей. Инструкции обработки в таблице стилей игнорируется полностью. Однако для копирования инструкций обработки из исходного дерева в конечное можно использовать <xsl:copy> или <xsl:copy-of>.

Примеры

Следующий пример выводит инструкцию обработки <?xml-stylesheet?> в начале выходного файла.

```
<xsl:processing-instruction name="xml-stylesheet">
   <xsl:text>href="housestyle.css" type="text/css"</xsl:text>
</xsl:processing-instruction>
```

При этом генерируется вывод:

```
<?xml-stylesheet href="housestyle.css" type="text/css"?>
```

Написание таблицы стилей XSLT, которая производит XML-документ, ссылающийся непосредственно на таблицу стилей CSS, – не такая абсурдная вещь, как это может показаться. Часто имеет смысл производить первую стадию обработки XML-файла на сервере, а вторую – на стороне клиента (другими словами, в броузере). На первой стадии будут извлечены данные, которые нужно отображать пользователю, и, что еще важнее, – будет удалена вся информация, которая не должна быть показана пользователям. На второй стадии будут применены детальные правила для форматирования вывода. Часто вторая стадия одинаково легко может быть выполнена как с CSS, так и с XSLT; а то, с чем CSS не может справиться, например добавление или переупорядочение текстового содержимого, можно произвести с помощью XSLT на первой стадии.

При генерировании инструкции обработки `<?xml-stylesheet?>` следует не упустить из внимания одну вещь, которая относится также и к другим инструкциям обработки. Это использование псевдоатрибутов и псевдосимволов, а также ссылок на сущности. Текст `«href="housestyle.css"»` в вышеупомянутом примере выглядит как атрибут XML, но фактически это не атрибут XML, а часть данных инструкции обработки. Она анализируется приложением, а не синтаксическим анализатором XML. Поскольку это не истинный XML-атрибут, нельзя сгенерировать его как узел атрибута, используя инструкцию `<xsl:attribute>`; скорее это необходимо генерировать как текст.

Правила для инструкции обработки `<?xml-stylesheet?>` определены в краткой рекомендации W3C, названной «Связь таблиц стилей с XML-документами» («Associating Style Sheets with XML Documents») и доступной по адресу *http://www.w3.org/TR/xml-stylesheet*. Наряду с определением данных этой инструкции обработки в форме псевдоатрибутов правила также позволяют использовать числовые ссылки на символы, например `«&#x20A4;»`, и заранее определенные ссылки на сущности, например `«>»` и `«&»`. Снова, это – не истинные ссылки на символы и сущности, которые признает синтаксический анализатор XML, и в результате они также не будут генерироваться XSLT-процессором. Если требуется включить `«&#x20A4;»` как часть данных инструкции обработки, можно записать, например, так:

```
<xsl:processing-instruction name="xml-stylesheet">
   <xsl:text>href="housestyle.css" type="text/css"</xsl:text>
   <xsl:text>title="Заголовок с символом &amp;#x20A4;"</xsl:text>
</xsl:processing-instruction>
```

xsl:script

Элемент <xsl:script> – элемент верхнего уровня, используемый для определения функций расширения, которые могут быть вызваны из XPath-выражений. Элемент <xsl:script> – новый элемент, введенный в рабочем проекте спецификации XSLT 1.1. Некоторые процессоры XSLT 1.0 обеспечивают эквивалентные механизмы, определяемые поставщиком: например, MSXML3 от Microsoft имеет элемент <msxml:script>, спецификация которого довольно похожа.

Рабочий проект XSLT 1.1 обеспечивает стандартные пути написания функций расширения на языке Java или JavaScript, хотя XSLT-процессоры не обязаны поддерживать любой из этих языков, и могут также поддерживать другие языки, привязки к которым не определены в стандарте.

Определен в

XSLT 1.1, раздел 14.4

Формат

```
<xsl:script
   implements-prefix = ИмяБезДвоеточия
   language = "ecmascript" | "javascript" | "java" | ПолноеИмя
   src = URI
   archive = список-URI >
            текст
</xsl:script>
```

Расположение

<xsl:script> – элемент верхнего уровня, поэтому он должен содержаться непосредственно в элементах <xsl:stylesheet> или <xsl:transform>.

В таблице стилей может находиться более одного элемента <xsl:script>; единственное ограничение состоит в том, что нельзя иметь два элемента <xsl:script> с одинаковыми атрибутами language или одинаковыми implements-prefix (или, точнее, префиксами, которые соответствуют одному URI пространства имен), если они не имеют различное преимущество импортирования. Порядок элементов <xsl:script> не имеет значения.

Атрибуты

Имя	Значение	Назначение
implements-prefix обязательный	Префикс пространства имен	Соответствуют префиксу, используемому в вызове функции расширения. (Считается, что префиксы соответствуют, если они связаны с одним и тем же URI пространства имен.)

Имя	Значение	Назначение
language обязательный	«ecmascript» | «javascript» | «java» | *ПолноеИмя*	Указывает язык, на котором написана функция расширения. Полное имя (которое должно иметь непустой префикс) используется для языков, привязки к которым определяются компанией-разработчиком.
src необязательный	URI	Задает местоположение кода (исходного или скомпилированного), реализующего функцию. Если опущен, исходный код может находиться в текстовом содержимом элемента <xsl:script>.
archive необязательный	Список URI, разделенных пробельными символами	Указывает места, где необходимо искать код, необходимый для поддержки функции расширения.

Содержимое

Если присутствует атрибут src, элемент должен быть пустым.

Если атрибут src опущен, элемент должен содержать код одной или более функций расширения на указанном языке. Содержимое является только текстовым, никаких элементов не должно быть. Так как в коде часто встречаются специальные символы XML типа «<» или «&», лучше записать их как секции CDATA между разделителями «<![CDATA[» и «]]>», но это необязательно.

Действие

Элемент <xsl:script> сообщает процессору, где найти реализации функций расширения, упомянутых в таблице стилей.

Функция расширения вызывается из XPath-выражения, использующего имя функции с префиксом, например «my:function()». Префикс «my» должен соответствовать URI пространства имен, которое действует в точке, где записано XPath-выражение; процессор находит реализацию функции, отыскивая элемент <xsl:script>, который имеет префикс, соответствующий этому же URI пространства имен.

Если поиск находит реализации для URI пространства имен более чем на одном языке, процессор может выбирать, какой из языков предпочтительнее. Идея здесь в том, что таблица стилей может обеспечивать альтернативные реализации одной и той же функции на разных языках, поэтому различные XSLT-процессоры могут найти версию, которую они способны обрабатывать. А проверка эквивалентности альтернативных реализаций – дело автора таблицы стилей.

Если поиск находит более одной реализации на одном и том же языке, то они должны различаться преимуществом импортирования. В идеале, процессор должен бы искать реализацию заданной функции с самым высоким преимуществом импортирования; однако спецификация признает, что это

нецелесообразно для всех языков, поэтому поведение в этом отношении зависит от языка.

Можно в одном элементе <xsl:script> определять набор функций с одним и тем же URI пространства имен. В объектно-ориентированном языке типа Java, где имена методов уникальны в пределах класса, отдельный URI пространства имен и, следовательно, отдельный элемент <xsl:script> понадобится для каждого класса, содержащего функции расширения, но все методы в пределах одного класса будут доступны через единственный элемент <xsl:script>. В языках типа JavaScript, которые позволяют, чтобы функции существовали в пределах глобального пространства имен, возможно использовать единственный элемент <xsl:script> для всех функций расширения.

Фактический код может быть или написан непосредственно в таблице стилей, или загружен из отдельного файла.

- Для интерпретируемого языка типа JavaScript применимо и то и другое. Можно записать код в пределах элемента <xsl:script> или в именованном файле, используя атрибут src, например «src="обработка-дат.js"».
- Для компилируемого языка типа Java единственная возможность – сообщить XSLT-процессору, откуда загрузить откомпилированный Java-код для соответствующих классов, и это делается указанием URI, который идентифицирует класс (например, «src="java:com.acme.toolslib.Widget"», возможно, дополненный атрибутом archive, записываемым аналогично переменной окружения CLASSPATH, определяющим список каталогов или JAR-файлов, в которых будет производиться поиск).

Элемент <xsl:script> спроектирован с расчетом на расширяемость для различных языков. Первоначально определены привязки только для JavaScript и Java (имена должны записываться в нижнем регистре, а имя «ecmascript» разрешено как синоним «javascript», – для тех, кто любит использовать официальное имя языка). Поставщиками могут предлагаться другие языки, типа B или Perl, но подробности – вне области действия стандарта.

Использование и примеры

Примеры использования этого элемента приведены в главе 8.

См. также

Дальнейшая информация о написании функций расширения на языках Java и JavaScript дана в главе 8 этой книги.

xsl:sort

Элемент <xsl:sort> используется для определения ключей сортировки и определения порядка, в котором должны обрабатываться узлы, выбранные с помощью <xsl:apply-templates> или <xsl:for-each>.

Определен в

XSLT, раздел 10

Формат

```
<xsl:sort
   select=Выражение
   order={"ascending" | "descending"}
   case-order={"upper-first" | "lower-first"}
   lang={код-языка}
   data-type={"text" | "number" | ПолноеИмя }/>
```

Расположение

<xsl:sort> – всегда непосредственный потомок элементов <xsl:apply-templates> или <xsl:for-each>. В нем может быть определено любое количество ключей сортировки в порядке от старшего к младшему. При использовании в <xsl:for-each> все элементы <xsl:sort> должны появляться раньше тела шаблона в инструкции <xsl:for-each>. При использовании в <xsl:apply-templates> элементы <xsl:sort> могут появляться как до, так и после любых элементов <xsl:with-param>.

Атрибуты

Имя	Значение	Назначение
select необязательный	выражение	Определяет ключ сортировки. Значение по умолчанию – строковое значение узла (выражение «string(.)»).
order необязательный	Шаблон значения атрибута, возвращающий: «ascending» | «descending»	Определяет, в каком порядке этого ключа обрабатываются узлы: в возрастающем или убывающем. Значение по умолчанию – «ascending».
case-order необязательный	Шаблон значения атрибута, возвращающий: «upper-first» | «lower-first»	Определяет, когда должны рассматриваться символы верхнего регистра – до или после символов нижнего. Значение по умолчанию зависит от языка.
lang необязательный	Шаблон значений атрибута, возвращающий код языка	Определяет язык, соглашения которого должны использоваться. Значение по умолчанию зависит от среды обработки.
data-type необязательный	Шаблон значения атрибута, возвращающий: «text» | «number» | ПолноеИмя	Определяет, должны ли значения рассматриваться в алфавитном порядке или в числовом, или с использованием определенного пользователем типа данных. Значение по умолчанию – «text».

Содержимое

Никакого; элемент всегда пуст.

Действие

Список элементов <xsl:sort> в пределах элемента <xsl:apply-templates> или <xsl:for-each> определяет порядок, в котором обрабатываются выбранные узлы. Сначала узлы сортируются первым ключом сортировки; любая группа узлов, имеющая дублирующиеся значения для первого ключа сортировки, сортируется вторым ключом, и так далее. Если группа узлов имеет дублирующиеся значения для всех ключей сортировки, она сортируется в порядке документа. Точно так же, когда не определено вообще никаких ключей сортировки, узлы обрабатываются в порядке документа.

Значение ключа сортировки для каждого узла в наборе узлов устанавливается вычислением выражения, заданного в атрибуте select. Они вычисляются в порядке документа, рассматривая каждый узел как текущий узел, а весь набор узлов – как текущий список узлов.

Это означает, что если требуется обработать узлы в обратном порядке документа, можно определить ключ сортировки следующим образом:

```
<xsl:sort select="position()"
          data-type="number"
          order="descending" />
```

Результат выражения всегда преобразуется в строку, а если атрибут data-type – «number», тогда строка преобразуется в число.

Способ сравнения ключей сортировки определяется другими атрибутами элемента <xsl:sort>. В спецификации XSLT некоторые из этих правил определены точно, другие даны просто как рекомендации для изготовителей, а третьи полностью определяются реализацией. Спецификация четко предупреждает, что различные реализации могут приводить к разным результатам.

Заметьте, что атрибут select должен быть выражением, которое вычисляется для каждого узла, принимая текущий узел за контекстный, что дает значение, указывающее позицию этого узла в последовательности. Нет прямого пути для указания того, что нужно использовать в различных случаях разные ключи сортировки. Иногда для этого ошибочно пишут что-то вроде:

```
<xsl:param name="ключ-сортировки" >
. . .
   <xsl:for-each select="КНИГА">
        <xsl:sort select="$ключ-сортировки"/>
```

в надежде, что если в качестве $ключ-сортировки указан «НАЗВАНИЕ», элементы будут отсортированы по названиям, и что если в качестве $ключ-сортировки указан «АВТОР», они будут отсортированы по авторам. Однако это не работает: переменная $ключ-сортировки будет иметь одно и то же значение для каждого элемента <КНИГА>, поэтому книги всегда будут выводиться в несортирован-

ном порядке. В том случае, когда ключ сортировки – всегда потомок сортируемых элементов, можно достичь нужного эффекта, написав:

```
<xsl:sort select="*[local-name()=$ключ-сортировки]">
```

Другой способ сделать сортировку условной состоит в том, чтобы использовать в качестве ключа сортировки условное выражение. Это несколько сложнее, но если число вариантов не изменяется, можно написать следующее:

```
<xsl:sort select="название[$ключ-сортировки='название'] |
                  автор[$ключ-сортировки='автор'] |
                  @isbn[$ключ-сортировки='isbn']">
```

Здесь используется выражение объединения (сконструированное с использованием оператора объединения «|»), чтобы объединить несколько наборов узлов, которые все, кроме одного, пустые, потому что предикаты – взаимоисключающие.

Атрибут `data-type` может иметь значение «text» или «number». Если значение – «text», то «10» будет выводиться раньше, чем «5», а при значении «number» – наоборот.

- Упорядочение чисел хорошо определено. В спецификации говорится, что они сортируются согласно числовому значению. Любые значения, которые являются не-числами, помещаются в начало последовательности. (Правило для сортировки значений не-чисел в оригинале спецификации XSLT 1.0 оставлено не определенным, поэтому может оказаться, что не все процессоры обрабатывают их таким способом.)
- В противоположность этому, упорядочение текстовых строк определено не очень четко. Алфавитный порядок зависим от языка, поэтому его следует регулировать атрибутом `lang`. Например, в современном немецком языке «д» идет сразу после «а» (раньше эта буква рассматривалась, как пара символов «ае»), в то время как в шведском языке «д» – отдельный символ, который стоит в конце алфавита, после «z». Спецификация XSLT рекомендует изготовителям ознакомиться с официальной рекомендацией Unicode по международной сортировке, с техническим отчетом #10 по Unicode (Unicode Technical Report #10), с Алгоритмом сравнивания Unicode (Unicode Collation Algorithm). Эти материалы см. по адресу *http://www.unicode.org/unicode/reports/tr10/index.htm*. Однако они не дают никаких обязательных правил.

Знание языка не помогает определить порядок символов верхнего и нижнего регистра (у каждого конкретного языка в мире собственные правила на этот счет), поэтому в XSLT для этого введен отдельный атрибут, `case-order`. Вообще, атрибут `case-order` используется только для упорядочения двух слов, которые считаются эквивалентными, если регистр не учитывается. Например, в немецком языке, где начальная прописная буква может изменять значение слова, некоторые словари перечисляют прилагательное *drall* (означающее «пухлый» или «эластичный») перед не связанным с ним существительным *Drall* («отклонение, кручение или смещение»), в то время

как в других языках принят противоположный порядок. Определение «case-order="lower-first"» поместило бы *drall* непосредственно перед *Drall*, в то время как при «case-order="upper-first"» сначала будет стоять *Drall*, а за ним – *drall*.

Атрибут order определяет, является ли порядок возрастающим или убывающим. Порядок по убыванию даст противоположный результат от порядка по возрастанию. Это означает, например, что не-числа окажутся последними, а не первыми, а также это означает, что эффект атрибута case-order реверсируется: если установить «case-order="upper-first"» с «order="descending"», то *drall* будет стоять **перед** *Drall*. (Опять же, это было уточнено уже после опубликования XSLT 1.0, поэтому могут встретиться процессоры, интерпретирующие эти правила иначе.)

Установка «data-type="ПолноеИмя"» была спешным добавлением к языку перед опубликованием версии 1.0, и ее поведение точно не определено. Это – зацепка, способствующая введению расширений от поставщиков или последующих расширений стандарта. Например, существует потребность в сортировке таких типов данных, как даты и время, определенных в рекомендации XML Schema Recommendation; поддержка типов данных XML Schema находится высоко в списке вещей, которые должны быть реализованы для XSLT 2.0.

Окончательный порядок отсортированных узлов определяет последовательность, в которой они обрабатываются имеющимися инструкциями <xsl:apply-templates> или <xsl:for-each>, а также значение функции position() при обработке каждого узла. Значение функции position() отразит его позицию в отсортированном списке.

Использование

XSLT предназначен для поддержки серьезных профессиональных издательских приложений, и, безусловно, это требует довольно мощных возможностей сортировки. На практике, однако, наиболее требовательные приложения почти неизменно имеют специфические правила сопоставления; например, правила для сортировки собственных имен в списке абонентов телефонной сети вряд ли подойдут для географических названий в географическом справочнике. Для этих приложений поставщики уже предоставляют зацепки, чтобы позволить введение алгоритмов сравнения, определяемых пользователем.

Примеры

Сначала будет приведена пара простых примеров, а затем – полный работающий пример, который можно загрузить и испытать самостоятельно.

Пример 1: Для обработки всех потомков <книга> текущего узла, сортируя их по значениям атрибута isbn:

```
<xsl:apply-templates select="книга">
   <xsl:sort select="@isbn"/>
</xsl:apply-templates>
```

Пример 2: Для вывода содержимого всех элементов <город> в документе в алфавитном порядке, включая каждый отдельный город только один раз:

```
<ul>
<xsl:for-each select="//город[not(.=preceding::город)]">
   <xsl:sort select="."/>
   <li><xsl:value-of select="."/></li>
</xsl:for-each>
</ul>
```

Если опустить атрибут «select="."» у элемента <xsl:sort>, эффект будет тем же самым, потому что это – значение по умолчанию; однако лучше включать его для ясности.

Пример: Сортировка по результатам вычисления

Этот пример выводит список продуктов, отсортированных по объему продаж каждого из них в убывающем порядке.

Исходный файл

Здесь приведен файл продукты.xml:

```
<продукты>
<продукт название="клубничный джем">
   <регион название="южный" продажи="20.00"/>
   <регион название="северный" продажи="50.00"/>
</продукт>
<продукт название="малиновый джем">
   <регион название="южный" продажи="205.16"/>
   <регион название="северный" продажи="10.50"/>
</продукт>
<продукт название="сливовый джем">
   <регион название="восточный" продажи="320.20"/>
   <регион название="северный" продажи="39.50"/>
</продукт>
</продукты>
```

Таблица стилей

продукты.xsl – полноценная таблица стилей, написанная с использованием упрощенного синтаксиса таблиц стилей, когда весь модуль таблицы стилей пишется как один конечный литеральный элемент. Упрощенные таблицы стилей описаны в главе 3.

Элемент <xsl:sort> сортирует выбранный набор узлов (содержащий все элементы <продукт>) по убыванию численной суммы атрибутов продажи во всех их дочерних элементах <регион>. Сумма рассчитывается с помощью функции sum() и отображается с помощью функции format-number().

```
<продукты xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:for-each select="продукты/продукт">
   <xsl:sort select="sum(регион/@продажи)" data-type="number" order="descending"/>
```

```
    <product name="{@название}"
                  sales="{format-number(sum(регион/@продажи), '$####0.00')}"/>
</xsl:for-each>
</продукты>
```

Вывод

Чтобы было понятнее, здесь добавлены переходы на новую строку:

```
<продукты>
<продукт название="сливовый джем" продажи="$359.70"/>
<продукт название="малиновый джем" продажи="$215.66"/>
<продукт название="клубничный джем" продажи="$70.00"/>
</продукты>
```

См. также

<xsl:apply-templates> на стр. 178

<xsl:for-each> на стр. 248

xsl:strip-space

Элемент <xsl:strip-space>, наряду с <xsl:preserve-space>, используется для управления способом обработки в исходном документе узлов из пробельных символов. Элемент <xsl:strip-space> задает элементы, у которых текстовые узлы только из пробельных символов являются незначимыми и могут быть удалены из исходного дерева.

Определен в

XSLT, раздел 3.4

Формат

```
<xsl:strip-space elements=список-КритериевИмен />
```

Расположение

<xsl:strip-space> – элемент верхнего уровня, то есть он всегда является непосредственным потомком элемента <xsl:stylesheet>. Нет никаких ограничений на положение его относительно других элементов верхнего уровня.

Атрибуты

Имя	Значение	Назначение
elements обязательный	Список из критериев имен, разделенных пробельными символами	Определяет элементы в исходном документе, текстовые узлы которых, состоящие только из пробельных символов, должны быть удалены

Конструкция КритерийИмени (NameTest) определена в главе 5.

Содержимое

Никакого; элемент всегда пуст.

Действие, использование и примеры

См. <xsl:preserve-space> на стр. 320. Эти два элемента, <xsl:strip-space> и <xsl:preserve-space>, тесно связаны, поэтому правила и рекомендации использования для них приведены в одном месте.

См. также

<xsl:preserve-space> на стр. 320

<xsl:text> на стр. 360

xsl:stylesheet

Элемент <xsl:stylesheet> – самый внешний элемент таблицы стилей. Как вариант, может использоваться его синоним <xsl:transform>.

Определен в

XSLT, раздел 2.2

Формат

```
<xsl:stylesheet
     id=identifier
     version="1.1"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     extension-element-prefixes=список-префиксов
     exclude-result-prefixes=список-префиксов >
        элемент-верхнего-уровня *
</xsl:stylesheet>
```

Расположение

<xsl:stylesheet> является самым внешним элементом каждого модуля таблицы стилей, кроме модуля, который использует синтаксис *конечного литерального элемента в качестве таблицы стилей*, описанный в главе 3. Он присутствует как в основной таблице стилей, так и в таблице стилей, которая импортирована или включена в другую таблицу стилей.

Кроме того, что элемент <xsl:stylesheet> является самым внешним элементом модуля таблицы стилей, для XML-документа он, как правило, является и элементом документа. Как отмечалось в главе 3, таблица стилей также может быть вложена в другой XML-документ.

Объявления пространств имен

У элемента <xsl:stylesheet>, как правило, всегда есть, по крайней мере, одно объявление пространства имен:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

Оно необходимо, чтобы идентифицировать документ как таблицу стилей XSLT. URI должен быть написан точно, как показано. Префикс «xsl» определен соглашением и используется во всей документации XSLT, включая эту книгу и сам стандарт, но можно было бы выбрать и другой префикс, например «XSLT». Тогда имя элемента было бы <XSLT:stylesheet> вместо <xsl:stylesheet>.

Можно также сделать это пространство имен пространством имен по умолчанию, используя следующее объявление:

```
xmlns="http://www.w3.org/1999/XSL/Transform"
```

В этом случае имя элемента будет просто <stylesheet>, и другие XSLT-элементы также будут именоваться без префикса, например, <template>, а не <xsl:template>. Хотя это допустимо, но обычно не рекомендуется, потому что в этом случае пространство имен по умолчанию недоступно для конечных литеральных элементов. Лучше всего резервировать пространство имен по умолчанию для конечных литеральных элементов, которые будут находиться в пространстве имен по умолчанию выходного документа.

> ***Если встречается таблица стилей, которая использует объявление пространства имен:***
>
> ```
> xmlns:xsl=http://www.w3.org/TR/WD-xsl
> ```
>
> ***тогда это вообще не таблица стилей XSLT, а таблица стилей, написанная на WD-xsl, черновом диалекте XSL, который Microsoft выпустил в 1998 с Internet Explorer 5. Он очень отличается от XSLT и не описывается в этой книге. Подробности о текущих продуктах Microsoft XSL см. в приложении A.***

Атрибуты

Имя	Значение	Назначение
Id необязательный	XML-имя	Идентификатор, используемый для идентификации элемента <xsl:stylesheet>, когда он вложен в другой XML-документ.
version обязательный	Число	Определяет версию XSLT, требуемую этой таблицей стилей. Используйте «1.1» для таблицы стилей, которой требуются возможности XSLT 1.1, или «1.0», если нужно, чтобы таблица стилей была переносимой между процессорами XSLT 1.0 и XSLT 1.1.

Имя	Значение	Назначение
extension-element-prefixes необязательный	Список имен без двоеточий, разделенных пробельными символами	Определяет любые пространства имен, использованные в этой таблице стилей для идентификации элементов расширения.
exclude-result-prefixes необязательный	Список имен без двоеточий, разделенных пробельными символами	Определяет любые пространства имен, использованные в этой таблице стилей, которые не должны копироваться в адресат вывода, если они не используются фактически в конечном документе.

Содержимое

Элемент <xsl:stylesheet> может содержать XSLT-элементы, квалифицируемые как **элементы верхнего уровня**. Это следующие элементы:

<xsl:attribute-set>
<xsl:decimal-format>
<xsl:import>
<xsl:include>
<xsl:key>
<xsl:namespace-alias>
<xsl:output>
<xsl:param>
<xsl:preserve-space>
<xsl:script> *
<xsl:strip-space>
<xsl:template>
<xsl:variable>

* Заметьте: <xsl:script> введен в XSLT 1.1.

Если есть элементы <xsl:import>, они должны располагаться перед всеми другими элементами верхнего уровня.

Элемент <xsl:stylesheet> может также содержать другие элементы при условии, что они имеют непустой URI пространства имен, который отличается от URI пространства имен XSLT. Если XSLT-процессор признает их URI пространства имен, он может интерпретировать такие элементы, как определено поставщиком, при условии, что они не воздействуют на правильное функционирование таблицы стилей. Если процессор не признает эти элементы, он должен игнорировать их.

Действие

Ниже описаны правила для каждого из атрибутов.

Атрибут id

Этот атрибут позволяет ссылаться на элемент <xsl:stylesheet>, когда он содержится внутри другого XML-документа.

Точное использование этого атрибута не определено в стандарте, но можно предположить, что атрибут id позволяет ссылаться в инструкции обработки <?xml-stylesheet?> на вложенную таблицу стилей. Пример дается в главе 3.

Атрибут version

Атрибут version определяет версию спецификации XSLT, на которую рассчитана таблица стилей. В настоящее время существует две опубликованные версии стандарта XSLT, а именно версии 1.0 и 1.1, следовательно, этот атрибут должен иметь одно из этих значений.

Реализации XSLT, даже если они реализуют только версию 1.0 стандарта, должны вести себя определенным образом, когда указанная версия отличается от 1.0.

- Если атрибут version – «1.0», процессор XSLT 1.0 должен сообщать как об ошибке о любых элементах в пространстве имен XSLT, которые он не опознает. Например, если он сталкивается с элементом <xsl:apply_template>, он посчитает это ошибкой (потому что правильное написание – <xsl:apply-templates>). Точно так же, процессор XSLT 1.1 должен отклонять неопознанные XSLT-элементы, если атрибут version установлен в «1.0» или «1.1».
- Если атрибут version имеет любое другое значение (включая даже значения ниже 1.0), процессор должен предположить, что таблица стилей рассчитана на использование особенностей, определенных в некоторой версии XSLT, которая неизвестна данной программе. Так, если атрибут version имеет значение «2.1» и встречается элемент <xsl:apply_template>, программа должна предположить, что это – новый элемент, определенный в версии 2.1 стандарта, а не ошибочное написание <xsl:apply-templates>. В спецификации XSLT это называется *обработкой в режиме совместимости с последующими версиями*. В этом режиме элемент <xsl:apply_template> будет отклонен, если только: (a) делается попытка подвергнуть его обработке, и (b) он не имеет дочерних элементов <xsl:fallback>. Если у него есть дочерний элемент <xsl:fallback>, обработан будет он, вместо неопознанного родительского элемента. Подробно это описано в разделе <xsl:fallback> на стр. 243.

Режим совместимости с последующими версиями воздействует также на обработку других очевидных ошибок. Например, при значении атрибута version «1.0» (или в случае процессора XSLT 1.1 – «1.1») любые непризнанные атрибуты или значения атрибутов у XSLT-элемента вызовут сообщение об ошибке, но в режиме опережающей совместимости такие атрибуты просто игнорируются. Точно так же в режиме опережающей совместимости несоответствующий XSLT-элемент типа <xsl:sort>, если он встречается в пределах тела шаблона, не вызовет сообщения об ошибке, пока не сделано попытки обработать его.

Атрибут version относится ко всей таблице стилей, кроме любых ее частей, содержащихся в пределах конечного литерального элемента, который имеет атрибут xsl:version. Его область действия – модуль таблицы стилей, а не все дерево таблицы стилей, созданное с помощью элементов <xsl:include> и <xsl:import>.

Атрибут extension-element-prefixes

Этот атрибут идентифицирует набор пространств имен, использованных для элементов расширения. Элементы расширения могут быть определены поставщиком, пользователем или третьим лицом. Они могут применяться в тех ситуациях, где используются инструкции, то есть в пределах тела шаблона. Если в теле шаблона встречается элемент, не принадлежащий пространству имен XSLT, то это или элемент расширения, или конечный литеральный элемент. Если его пространство имен соответствует пространству имен, указанному в атрибуте extension-element-prefixes содержащего его элемента <xsl:stylesheet>, то он будет рассматриваться как элемент расширения, в противном случае – как конечный литеральный элемент.

Значение атрибута – список префиксов, разделенных пробельными символами; каждый из этих префиксов должен идентифицировать объявление пространства имен, имеющееся у элемента <xsl:stylesheet>. Пространство имен по умолчанию (пространство имен, объявленное с помощью атрибута xmlns) может быть назначено пространством имен элемента расширения путем включения псевдопрефикса «#default».

Список пространств имен, используемых для элементов расширения, может быть применен к разделу таблицы стилей с помощью атрибута xsl:extension-element-prefixes конечного литерального элемента или элемента расширения. Это не отменяет объявлений на уровне таблицы стилей, но дополняет их.

Область действия атрибута extension-element-prefixes – модуль таблицы стилей, а не все дерево таблицы стилей, созданное с использованием элементов <xsl:include> и <xsl:import>.

Если пространство имен обозначено как пространство имен элементов расширения, то любой XSLT-процессор признает, что эти элементы – элементы расширения. Однако некоторые XSLT-процессоры могут оказаться неспособными применить их. Например, если пространство имен http://www.jclark.com/xt/extensions обозначено как пространство имен расширений, то и xt, и Xalan признают элементы расширения, но есть вероятность, что xt сумеет поддерживать их, а Xalan – нет. Если процессор знает, как обработать элемент, он сделает это; в противном случае он проверит, содержит ли этот элемент инструкцию <xsl:fallback>. Если да, процессор обработает инструкцию <xsl:fallback>, если нет – сообщит об ошибке.

Необходимо только определить пространство имен как пространство имен элементов расширения, чтобы отличать элементы расширения от конечных литеральных элементов. На верхнем уровне таблицы стилей нет риска путаницы. Любая реализация может определять собственные элементы верхнего уровня, используя собственное пространство имен, а другие реализации просто проигнорируют эти элементы, рассматривая их как данные. Таким образом, атрибут extension-element-prefixes необязателен для идентификации элементов верхнего уровня, являющихся расширениями, определяемыми поставщиком или пользователем. Например, можно использовать <msxml:script> как элемент верхнего уровня, но все процессоры, кроме Micro-

soft MSXML3, вероятно, проигнорируют его (хотя нет требования, что они должны поступать именно так).

Атрибут exclude-result-prefixes

Этот атрибут определяет набор пространств имен, которые не должны копироваться в конечное дерево.

XSLT-процессор должен сформировать корректное дерево, которое соответствует модели данных (как описано в главе 2) и правилам пространств имен XML, чтобы в выходном файле не оказалось префиксов пространств имен, которые не были объявлены. Однако в выходном файле часто остаются ненужные и нежелательные объявления пространств имен, например объявления пространств имен, которые соответствуют узлам в исходном документе, не использующимся в документе вывода, или пространств имен для функций расширения, которые используются только в таблице стилей. Эти лишние объявления пространств имен обычно не приносят вреда, так как они не воздействуют на суть выходного файла, но они загромождают его. Кроме того, они влияют на его верификацию при создании конечного документа, который соответствует специфическому DTD. Таким образом, атрибут `exclude-result-prefixes` помогает избавиться от подобных объявлений.

В особенности, спецификация XSLT требует, чтобы при обработке в таблице стилей конечного литерального элемента он копировался в конечное дерево вместе со всеми его узлами пространств имен, кроме пространства имен XSLT и любого пространства имен, определяемого элементами расширения. Элемент имеет узел пространства имен для каждого действующего пространства имен, включая пространства имен, определяемые родительскими элементами, а также непосредственно данным элементом. По этой причине копируемые пространства имен включают не только пространства имен, определенные в конечном литеральном элементе и фактически используемые в нем, но даже и такие, которые просто *доступны* для использования.

Очень часто, конечно, один конечный литеральный элемент будет прямым или дальнейшим потомком другого, и узлы его пространств имен будут включать копии всех узлов пространств имен родительского элемента. Например, рассмотрим такую таблицу стилей:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>

<xsl:template match="/">
   <acme:документ xmlns:acme="http://acme.com/xslt">
      <acme:глава>
         Давным-давно ...
      </acme:глава>
   </acme:документ>
</xsl:template>

</xsl:stylesheet>
```

Это можно представить деревом, изображенным ниже на схеме с использованием той же системы обозначений, которая была принята в главах 2 и 3.

Хотя имеется только два объявления пространств имен, они распространяются на всех потомков, поэтому, например, элемент <acme:глава> имеет два узла пространств имен, хотя сам он не имеет никаких объявлений пространств имен. (Он имеет также узел пространства имен для пространства имен «xml», не показанный на диаграмме.)

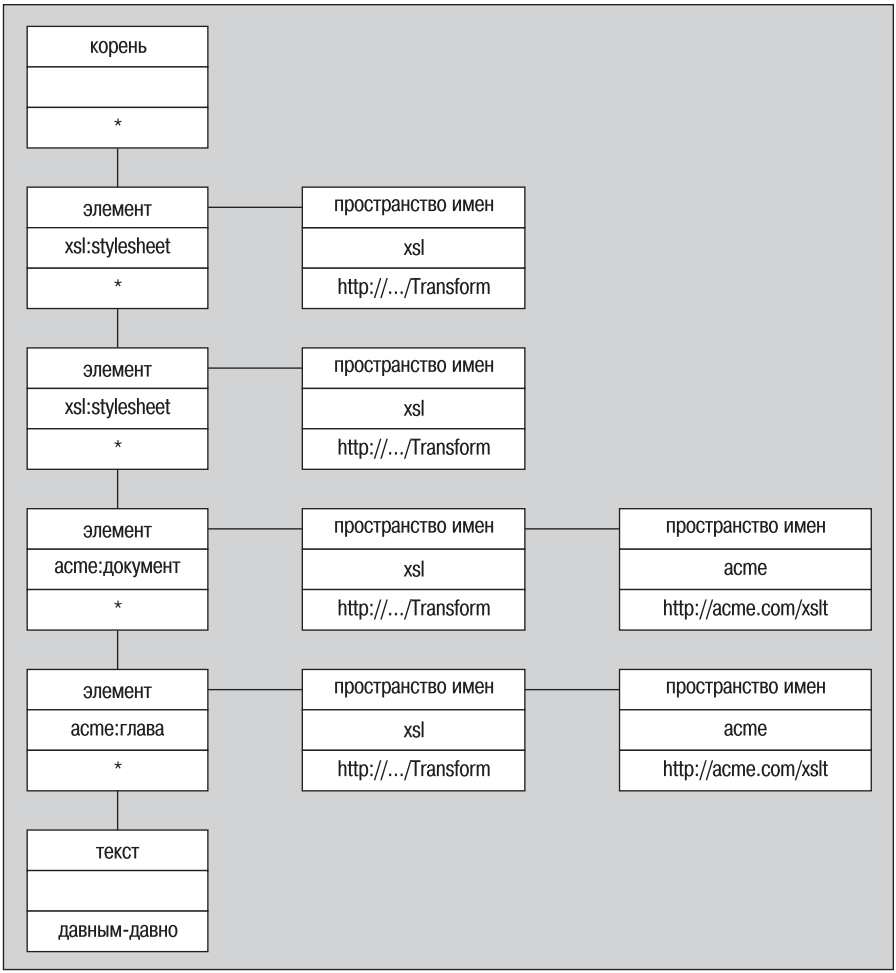


Рис. 4.8. *Древовидная структура таблицы стилей. Узлы пространств имен дочерних элементов включают и копии всех узлов пространств имен родительского элемента*

В спецификации говорится, что каждый конечный литеральный элемент копируется со всеми его узлами пространств имен (за исключением пространства имен XSLT), поэтому конечное дерево должно выглядеть так (снова опущены узлы пространства имен «xml»):

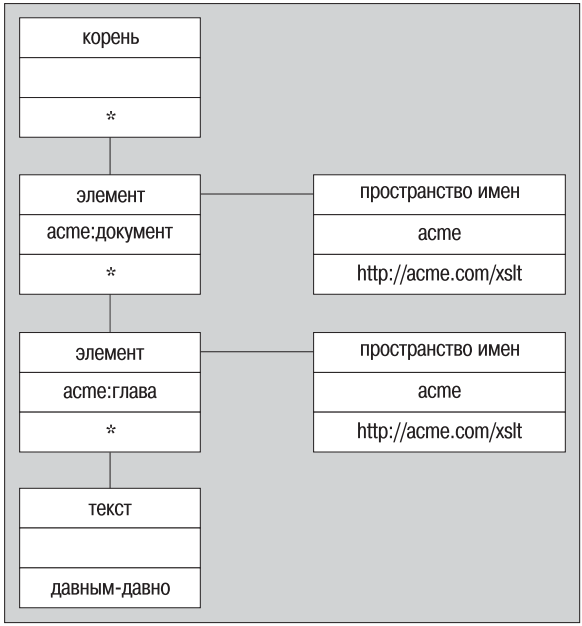


***Рис. 4.9.** Структура дерева: все конечные литеральные элементы копируются со всеми их узлами пространств имен*

Оба элемента, <acme:документ> и <acme:глава>, имеют узел пространства имен «acme». Однако это не означает, что объявление пространства имен будет излишне повторено в выходном файле: здесь речь идет о созданном абстрактном дереве, а не о конечном сериализованном файле XML. Исключение дублирующихся объявлений пространств имен – целиком задача XSLT-процессора, и большинство процессоров произведет следующий вывод, показанный для наглядности с отступами:

```
<acme:документ xmlns:acme="http://acme.com/xslt">
   <acme:глава>
      Давным-давно ...
   </acme:глава>
</acme:документ>
```

Задача атрибута exclude-result-prefixes – избавляться не от дублирующихся объявлений, а от объявлений, которые не нужны вообще, которые относятся к другим вещам. Например, предположим, что таблица стилей была такой:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:var="http://another.org/xslt"/>

<xsl:variable name="var:x" select="17"/>

<xsl:template match="/">
   <acme:документ xmlns:acme="http://acme.com/xslt">
```

```
      <acme:глава>
         Давным-давно ...
      </acme:глава>
   </acme:документ>
</xsl:template>

</xsl:stylesheet>
```

В этом случае, хотя тело шаблона не изменилось, каждый из элементов <acme:документ> и <acme:глава> теперь имеет дополнительный узел пространства имен, который будет копироваться в выходной файл, хотя он и не используется, приводя к выводу:

```
<acme:document xmlns:acme="http://acme.com/xslt"
               xmlns:var="http://another.org/xslt">
   <acme:глава>
      Давным-давно ...
   </acme:глава>
</acme:документ>
```

Почему XSLT-процессор не может просто включать в вывод только фактически используемые в элементе и именах атрибутов пространства имен и опускать остальные? Дело в том, что многие приложения XML, подобно самому XSLT, используют механизм пространств имен для создания уникальных значений в собственных данных. Например, префиксы пространства имен могут находиться не только в именах атрибутов, но и в их значениях. XSLT-процессор не может отличить их от обыкновенных значений, поэтому он вынужден перестраховываться.

Таким образом, если имеются пространства имен, нежелательные в дереве вывода, можно указать их в атрибуте exclude-result-prefixes элемента <xsl:stylesheet>. Этот атрибут – список префиксов пространств имен, разделенных пробельными символами. В него также можно включать пространство имен по умолчанию под псевдопрефиксом «#default».

Префикс, как обычно, является просто способом ссылки на соответствующий URI пространства имен: в действительности исключается не префикс, а именно URI пространства имен. Так, если тот же самый URI пространства имен объявить снова с другим префиксом, это пространство имен все равно останется исключенным.

Атрибут exclude-result-prefixes элемента <xsl:stylesheet> обеспечивает способ контроля над исключением пространств имен для всей таблицы стилей. Дополнительные префиксы, которые нужно исключить, могут быть указаны в атрибуте xsl:exclude-result-prefixes непосредственно конечного литерального элемента – это воздействует только на этот элемент и на его потомков.

Атрибуты xsl:exclude-result-prefixes и exclude-result-prefixes применяются только к узлам пространств имен, скопированным из таблицы стилей, используя конечные литеральные элементы. Они не воздействуют на узлы пространств имен, скопированные из исходного документа с помощью <xsl:copy> или <xsl:copy-of>; способа исключить эти узлы нет.

Подобно другим атрибутам элемента <xsl:stylesheet>, атрибут exclude-result-prefixes имеет отношение только к элементам в пределах модуля таблицы стилей, но не к элементам, привнесенным с использованием <xsl:include> или <xsl:import>.

Что случится, если исключить пространство имен, которое фактически необходимо, поскольку оно используется в конечном дереве? XSLT-процессор обязан генерировать вывод, который отвечает рекомендации по пространствам имен, поэтому он проигнорирует запрос на исключение этого пространства имен. (Формально, спецификация XSLT 1.0 позволяет сериализатору генерировать любое дополнительное объявление пространства имен. В XSLT 1.1 правила более жесткие: узлы пространств имен должны генерироваться в конечном дереве, только если они фактически необходимы, чтобы объявить префикс для URI пространства имен, использованного в расширенном имени узла атрибута или элемента.)

Использование и примеры

Элемент <xsl:stylesheet> – всегда самый внешний элемент таблицы стилей (хотя таблица стилей может быть вложена в другой документ). Он непременно включает:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
>
```

Различные необязательные атрибуты рассматриваются в следующих разделах.

Атрибут id

Если используемый XSLT-процессор поддерживает вложение таблиц стилей внутрь исходного документа, который подлежит преобразованию, тогда типичная компоновка будет такой:

```
<?xml version="1.0"?>
<?xml-stylesheet href="#style" type="text/xsl"?>
<данные>
...
...
   <xsl:stylesheet id="style" version="1.0"
                   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:include href="модуль1.xsl"/>
   <xsl:include href="модуль2.xsl"/>
   <xsl:template match="xsl:*"/>
   </xsl:stylesheet>
</данные>
```

Заметьте, что при использовании такой структуры таблица стилей представляется всем исходным документом, включая копию себя. Поэтому таблица стилей должна быть написана так, чтобы она могла соответствующим образом

обрабатывать свои собственные элементы, следовательно, должен быть пустой шаблон, который соответствует всем элементам в пространстве имен XSLT.

Атрибут version

Если таблица стилей использует возможности только XSLT 1.0, следует в элементе <xsl:stylesheet> указать «version="1.0"». Тогда эта таблица стилей будет работать с любым XSLT-процессором, который соответствует XSLT 1.0 или более поздней версии стандарта.

Если используются возможности, впервые определенные в XSLT 1.1, следует указать «version="1.1"» или у элемента <xsl:stylesheet>, или у конечного литерального элемента, который включает часть таблицы стилей, использующую особенности версии 1.1. Это, конечно, необязательно, если используется <xsl:script>, который является элементом верхнего уровня и поэтому никогда не включается в конечный литеральный элемент. В таком случае нужно поместить элемент <xsl:script> во включенный модуль таблицы стилей и указать «version="1.1"» в элементе <xsl:stylesheet> этого модуля.

Атрибут extension-element-prefixes

В этом атрибуте необходимо указывать список всех префиксов, которые используются в данной таблице стилей для элементов расширения. В наиболее типичном случае атрибут или опускается совсем, или включает единственный префикс пространства имен, использованного поставщиком для собственных частных расширений. У элемента <xsl:stylesheet> также всегда должно иметься объявление этого пространства имен.

Например, при использовании Saxon:

```
<xsl:stylesheet
     version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:saxon="http://icl.com/saxon"
     extension-element-prefixes="saxon">
```

Не включайте в атрибут префиксы поставщика, если частные расширения не задействованы фактически в таблице стилей. Для использования частных элементов верхнего уровня, например <msxml:script>, не требуется включать атрибут extension-element-prefixes, это нужно делать, только если частные расширения типа инструкций предполагается использовать в пределах тела шаблона, где иначе они могут быть приняты за конечные литеральные элементы.

Если использование расширений поставщика четко локализовано в таблице стилей, лучше идентифицировать их, используя атрибут xsl:extension-element-prefixes самого элемента расширения или конечного литерального элемента, который окружает тело шаблона, где фактически используются эти расширения. Это способствует сохранению переносимости таблицы стилей и облегчает возможность видеть, какие части таблицы стилей являются стандартными, а какие используют частные расширения.

Если требуется использовать расширения от нескольких разных поставщиков, можно перечислить их все в этом атрибуте. XSLT-процессор от одного поставщика спокойно воспримет пространство имен другого поставщика, обнаружив его в списке. Заминка произойдет, если только от процессора ожидают фактического применения частной инструкции, которую он не понимает. Однако даже в этом случае при наличии у незнакомого элемента расширения дочерней инструкции `<xsl:fallback>`, которая определяет поведение при сбойной ситуации, процессор выполнит эту инструкцию вместо той, которая ему непонятна.

Хотя элементы расширения от поставщиков продуктов XSLT, вероятно, будут наиболее общим случаем, в принципе, можно также устанавливать расширения от третьих лиц или писать свои собственные (правда, API у всех поставщиков разные). Все, сказанное относительно расширений от поставщиков, одинаково применимо и к собственным расширениям, и к расширениям от третьих лиц.

Для получения дополнительной информации относительно расширений, обеспечиваемых различными поставщиками в их продуктах, см. соответствующее приложение.

Оригинальное применение атрибута `extension-element-prefixes` заключается в том, чтобы пометить элементы, которые требуется зарезервировать для отладки или аннотаций. Например, если где-нибудь в таблице стилей в пределах тела шаблона включить следующий элемент (см. ниже), то он будет выведен в конечном дереве, отображая текущие значения переменных `$var1` и `$var2`.

```
<отладка:вывод var1="{$var1}" var2="{$var2}"><xsl:fallback/></отладка:вывод>
```

Когда отпадет необходимость в этой инструкции отладки, ее можно легко отключить, объявив «`отладка`» префиксом элемента расширения. В этом случае будет выполняться инструкция `<xsl:fallback/>`, потому что процессор не знает этот элемент расширения.

Атрибут exclude-result-prefixes

Простейшим способом решить, какие префиксы пространства имен нужно здесь перечислять, является метод проб и ошибок. Запустите таблицу стилей на выполнение, и если документ вывода будет содержать объявления пространств имен, которые явно не нужны, добавьте их к атрибуту `exclude-result-prefixes`. После этого снова выполните таблицу стилей.

Пространства имен, которые используются только в пределах таблицы стилей, обычно исключаются автоматически. Сюда относится непосредственно пространство имен XSLT, а также пространства имен, используемые для элементов расширения. Однако это не касается пространств имен, использованных в таблице стилей для собственных пользовательских элементов верхнего уровня.

Наиболее типичная причина попадания в выходной документ нежелательных объявлений пространств имен кроется там, где таблица стилей должна ссылаться на пространства имен, использованные в исходном документе, например в шаблоне образца match, но откуда ни один из элементов не копируется в целевой документ.

Например:

```
<xsl:stylesheet
          version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:po="http://accounting.org/xslt"
          exclude-result-prefixes="po">
<xsl:template match="po:заказ">
   <части-заказа>
   ...
   </части-заказа>
</xsl:template>
</xsl:stylesheet>
```

Если бы не атрибут exclude-result-prefixes, пространство имен «po» копировалось бы в документ результата, потому что оно является действующим, когда применяется конечный литеральный элемент <части-заказа>.

Как и в случае других атрибутов элемента <xsl:stylesheet>, необязательно производить исключения префиксов во всей таблице стилей, можно ограничиться любой ее частью, используя атрибут xsl:exclude-result-prefixes в любом конечном литеральном элементе. Это довольно удобно на практике – сохранять в одном месте и объявление пространства имен, и директиву, которая исключает его из результирующего документа.

См. также

<xsl:transform> на стр. 366

xsl:template

Элемент <xsl:template> определяет шаблон для создания вывода. Он может быть вызван путем сопоставления узлов с образцом или же явным образом – по имени.

Определен в

XSLT, раздел 5.3

Формат

```
<xsl:template
     name=ПолноеИмя
     match=Образец
```

```
      mode=ПолноеИмя
      priority=Число >
        <xsl:param> *
          тело шаблона
  </xsl:template>
```

Расположение

<xsl:template> – элемент верхнего уровня, то есть он должен быть непосредственным потомком элемента <xsl:stylesheet>.

Атрибуты

Имя	Значение	Назначение
match необязательный	образец	Образец, определяющий, какие узлы подлежат обработке данным шаблоном. Если этот атрибут отсутствует, должен иметься атрибут name.
name необязательный	полное имя	Имя шаблона. Если этот атрибут отсутствует, должен иметься атрибут match.
priority необязательный	число	Число (положительное, отрицательное, целое или действительное), которое указывает приоритет этого шаблона, используемого, когда несколько шаблонов соответствуют одному и тому же узлу.
mode необязательный	полное имя	Режим. Когда для обработки набора узлов используется <xsl:apply-templates>, единственные рассматриваемые шаблоны – это шаблоны с соответствующим режимом.

Конструкции полное имя и число определены в главе 5. Конструкция образец определена в главе 6.

Содержимое

Ни одного или более элементов <xsl:param>, сопровождаемых телом шаблона.

Действие

Элемент <xsl:template> обязательно должен иметь или атрибут match, или атрибут name, или оба эти атрибута.

- Если есть атрибут match, элемент <xsl:template> определяет шаблонное правило, которое может быть вызвано с помощью инструкции <xsl:apply-templates>.
- Если есть атрибут name, элемент <xsl:template> определяет именованный шаблон, который может быть вызван с помощью инструкции <xsl:call-template>.
- Если присутствуют оба атрибута, шаблон может быть вызван любым из этих способов.

Атрибут match

Атрибут match – это образец, определенный в главе 6. Образец используется, чтобы определить, к каким узлам применимо данное шаблонное правило.

Образец не должен содержать ссылки на переменные типа «$var». Это должно предотвратить зацикливающиеся определения: тело шаблона глобальной переменной может включать инструкцию <xsl:apply-templates>, поэтому должна быть возможность вычислить образец до того, как станут известными значения глобальных переменных.

Что могло бы случиться, если бы в атрибуте match были разрешены переменные? Рассмотрим следующее:

```
<xsl:variable name="x">
   <xsl:apply-templates select="//item"/>
</xsl:variable>
<!-- НЕВЕРНО -->
<xsl:template match="item[$x]">
   <xsl:value-of select="3"/>
</xsl:template>
<!-- НЕВЕРНО -->
```

Очевидно, в то время как переменная x вычисляется, невозможно определить, соответствует это шаблонное правило отдельному узлу <item> или нет.

Когда <xsl:apply-templates> используется для обработки отобранного набора узлов, каждый узел обрабатывается, используя наиболее подходящее для данного узла шаблонное правило, как описано в разделе <xsl:apply-templates> на стр. 178.

Шаблон рассматривается как кандидат, только если узел соответствует образцу в атрибуте match и если значение атрибута mode – такое же, как значение атрибута mode в инструкции <xsl:apply-templates>.

Если этим критериям отвечает более одного шаблонного правила, они сначала рассматриваются в порядке их преимущества импортирования (как описано в разделе <xsl:import> на стр. 256), а далее принимаются во внимание только шаблоны с самым высоким преимуществом импортирования.

Если все еще остается более одного подходящего шаблонного правила (другими словами, если два шаблонных правила, которые оба соответствуют узлу, имеют равное преимущество импортирования), сравниваются их приоритеты. Приоритет задается или значением атрибута priority, описанного ниже, или значением приоритета по умолчанию, которое определяется образцом match.

Если это оставляет один образец с численно более высоким приоритетом, чем у всех других, то образец выбран. Если самый высокий одинаковый приоритет оказывается опять у нескольких образцов, XSLT-процессор может или сообщить об ошибке, или выбрать из оставшихся шаблонов тот, который находится последним в таблице стилей. На практике некоторые процессоры просто выдают предупреждение, которое при желании можно игнори-

ровать. Однако опыт показывает, что такая ситуация часто свидетельствует о том, что автор таблицы стилей что-то упустил.

Атрибут name

Атрибут `name` – это полное имя, то есть имя, опционально уточняемое префиксом пространства имен. Если префикс есть, он должен соответствовать объявлению пространства имен, которое действует в этом элементе (это значит, оно должно быть определено или непосредственно в этом элементе, или в элементе `<xsl:stylesheet>`). Если префикса нет, то URI пространства имен пустой: пространство имен по умолчанию не используется.

Это имя используется, когда шаблон вызывается с помощью `<xsl:call-template>`. Атрибут `name` элемента `<xsl:call-template>` должен соответствовать атрибуту `name` элемента `<xsl:template>`. Два имени соответствуют друг другу, если они имеют одинаковую локальную часть и один и тот же URI пространства имен, при этом префиксы могут быть различны.

Если в таблице стилей находится более одного именованного шаблона с одинаковым именем, то используется шаблон с более высоким преимуществом импортирования. Подробнее см. в разделе `<xsl:import>` на стр. 256. Будет ошибкой, если в таблице стилей окажется два шаблона с одинаковым именем и равным преимуществом импортирования, даже если эти шаблоны не вызываются.

Атрибут priority

Атрибут `priority` – это число, например: «17», «0.5» или «-3». Иногда это – число, как определено в синтаксисе XPath-выражений, данном в главе 5, с необязательным знаком «минус» перед ним.

Атрибут `priority` используется при решении о выборе шаблона, когда вызывается `<xsl:apply-templates>` и имеется несколько возможных кандидатов. Для каждого отобранного узла шаблон выбирается согласно следующей процедуре:

- Сначала отбираются все шаблоны, имеющие атрибут `match`.
- Из них выбираются шаблоны, которые имеют такой же режим, какой используется при вызове `<xsl:apply-templates>`. Если элемент `<xsl:apply-templates>` не имеет атрибута `mode`, выбранные шаблоны тоже не должны его иметь.
- Из отобранных на предыдущей стадии шаблонов выбираются те, образец которых соответствует данному узлу.
- Если таких шаблонов больше, чем один, выбирается тот, который имеет самое высокое преимущество импортирования.
- Если все еще остается более одного шаблона, выбирается шаблон с численно самым высоким приоритетом.

Если снова остается несколько соответствующих данному узлу шаблонов и все они имеют равные преимущества импортирования и приоритеты, XSLT-

процессор может или выбрать из них шаблон, который в таблице стилей встречается последним, или сообщить об ошибке.

Если не обнаружено шаблонов, которые соответствуют данному узлу, используется встроенный шаблон для соответствующего типа узлов. Встроенные шаблоны описаны в разделе <xsl:apply-templates> на стр. 178.

Заданный по умолчанию приоритет зависит от образца и выбирается, согласно следующим правилам. Численно более высокое значение указывает на более высокий приоритет.

Синтаксис образца	Приоритет по умолчанию
`Образец1 «|» Образец2`	Это рассматривается так, словно указаны два совершенно отдельных шаблонных правила, одно – для `Образец1`, а другое – для `Образец2`, и по ним независимо вычисляется заданный по умолчанию приоритет образцов `Образец1` и `Образец2`

`ПолноеИмя`
`«@» ПолноеИмя`
`«child::» ПолноеИмя`
`«attribute::» ПолноеИмя`
`«processing-instruction» «(» Литерал «)»`
	0.0

`ИмяБезДвоеточия «:*»`
`«@» ИмяБезДвоеточия «:*»`
`«child::» ИмяБезДвоеточия «:*»`
`«attribute::» ИмяБезДвоеточия «:*»`
	-0.25

`КритерийУзла`
`«@» КритерийУзла`
`«child::» КритерийУзла`
`«attribute::» КритерийУзла`
	-0.5

В остальных случаях	0.5

Эти приоритеты по умолчанию выбраны очень тщательно. Они отражают селективность образца:

- Образцы `«node()»`, `«text()»` и `«*»` – совсем не селективны, они соответствуют любому узлу правильного типа узлов, поэтому они имеют низкий приоритет: -0.5.
- Образцы формата `«abc:*»` или `«@xyz:*»` – более селективны, они соответствуют только узлам элементов или атрибутов, принадлежащих специфическому пространству имен, поэтому они имеют более высокий приоритет, чем образцы предыдущей категории.
- Образцы, такие как `«заголовок»` или `«@isbn»`, – самые распространенные образцы, их приоритет по умолчанию 0.0 отражает то, что в смысле селективности они являются наиболее типичными.

- Образцы, более специфические, чем предыдущие: например, «книга[@isbn]» или «глава/заголовок», или «абзац[1]» — имеют более высокий приоритет, поэтому они получат предпочтение перед шаблонами, образцами в которых являются, соответственно, «книга», «заголовок» или «абзац». Стоит отметить, однако, что эта категория может также включать образцы, которые оказываются вовсе не очень селективными, например «//node()».

Все эти значения можно заменить и назначить вместо них собственные приоритеты в виде натуральных чисел, например «1», «2», «3», причем такие шаблоны всегда будут выбираться раньше, чем шаблоны с приоритетами по умолчанию, распределенными системой.

Возможно, кто-то решит, что таблицы стилей будут проще для понимания и менее подвержены ошибкам, если не полагаться на приоритеты по умолчанию, а использовать явные приоритеты всякий раз, когда более одного шаблонного правила может соответствовать одному и тому же узлу.

Атрибут mode

Атрибут mode, если он есть, должен представлять собой полное имя, то есть быть именем с необязательным префиксом пространства имен. Когда <xsl:apply-templates> используется с конкретным режимом, рассматриваются только шаблонные правила с тем же самым режимом, а когда <xsl:apply-templates> используется без атрибута mode, будут учитываться только шаблонные правила также без атрибута mode.

Режимы сравниваются с помощью обычных правил для полных имен: оба имени раскрываются, используя объявления пространств имен, действующие для соответствующих им элементов таблицы стилей (за исключением любых объявлений пространства имен по умолчанию), и они считаются соответствующими друг другу, если совпадают их локальные имена и URI пространства имен.

Режим, указанный в шаблоне <xsl:template>, **не** распространяется ни на какие элементы <xsl:apply-templates> в пределах тела шаблона. Хотя обычной практикой является обработка целого поддерева в одном режиме, а следовательно, шаблон продолжает и дальше использовать режим, в котором он был вызван, но поведением по умолчанию это является только для встроенных шаблонов.

Если шаблон имеет атрибут mode, но не имеет атрибута match, это не будет ошибкой, но будет излишеством, потому что в такой ситуации режим никогда не будет использован.

Если шаблон имеет атрибут mode, но в таблице стилей нет ни одного элемента <xsl:apply-templates> с таким же режимом, это тоже не будет ошибкой, хотя это означает, что шаблон никогда не будет выбран ни одним запросом <xsl:apply-templates>. Это можно использовать как подручный способ закомментировать шаблонное правило.

Применение шаблона

После того как элемент <xsl:template> выбран для обработки, происходит следующее:

- При вызове шаблона с помощью <xsl:apply-templates> устанавливаются, как принято, текущий узел и список текущего узла.
- Распределяется новый стековый фрейм для хранения нового экземпляра каждой локальной переменной, определенной в пределах шаблона.
- Вычисляются все параметры, перечисленные в элементах <xsl:param>, содержащихся в пределах элемента <xsl:template>. Эти элементы <xsl:param> должны находиться в теле шаблона перед всеми инструкциями. Значение каждого параметра определяется так: если оно задано вызывающей программой (с использованием элемента <xsl:with-param> с соответствующим именем), то параметру присваивается заданное значение. В противном случае непосредственно вычисляется значение элемента <xsl:param>: см. раздел <xsl:param> на стр. 316.
- Тело шаблона применяется. Это означает, что дочерние узлы элемента <xsl:template> применяются по очереди. Инструкции XSLT и элементы расширения обрабатываются по установленным для них правилам; конечные литеральные элементы и текстовые узлы записываются в текущий адресат вывода.

Когда обработка тела шаблона завершается, стековый фрейм, содержащий его локальные переменные, удаляется. Контроль возвращается вызывающему шаблону, а текущий узел и список текущего узла возвращаются к их прежним значениям.

> *Разные реализации, конечно, могут производить обработку в различном порядке, если это не влияет на результат. Некоторые программы предпочитают отложенные вычисления: параметры вычисляются, только когда они в первый раз потребуются. Это можно обнаружить, если используются функции расширения, имеющие побочные эффекты, или если используется* <xsl:message>*, чтобы следить за последовательностью выполнения.*

Использование и примеры

Сначала будет продемонстрировано применение шаблонных правил, затем будут даны некоторые советы по использованию режимов, и наконец будут обсуждены именованные шаблоны.

Использование шаблонных правил

Шаблонное правило – это элемент <xsl:template> с атрибутом match, который поэтому можно вызвать, используя инструкцию <xsl:apply-templates>.

Такой подход к обработке, основанный на правилах, – характерный способ создания таблиц стилей XSLT, хотя он ни в коем случае не является единственным способом. Самое большое его преимущество в том, что вывод для каждого типа элементов может быть определен независимо от контекста, в

котором появляется элемент. Благодаря этому, очень легко многократно использовать типы элементов в различных контекстах или добавлять новые типы элементов к существующему определению документа без необходимости перезаписывать шаблоны таблицы стилей с учетом всех возможных элементов. Классический пример подобного подхода к обработке – преобразование семантической разметки документа в разметку для отображения, что демонстрирует следующий пример.

Пример: Шаблонные правила

Исходный файл

Исходный файл – `солист.xml`.

В этом тексте, характеризующем певца, имя композитора помечено тегом `<композитор>`, название музыкального произведения – тегом `<произведение>`, а название публикации – тегом `<публикация>`. Разделы текста, относящиеся к одному и тому же выступлению, помечены тегом `<выступление>`. Таким образом, фрагмент размеченного текста выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<резюме>
<абзац>
<выступление>
   <публикация>Обзор старинной музыки</публикация>
   рассказал о дебютном концерте на сцене
   <сцена>Вигмор (Wigmore)</сцена> с
   <группа>Ансамблем Соннери</группа>
   в <дата>1998</дата> году: <цитата>Один из самых
   прекрасных концертов, которые мне доводилось слушать
   ... певец, которого нельзя пропустить</цитата>.
</выступление>
<выступление>
   В число других его ярких выступлений входят
   трансляция по телевидению произведения
   <произведение>"Страсти по Матфею" </произведение>(
   <композитор>И. С. Бах</композитор>
   ) в постановке <певец>Джонатана Миллера</певец>,
   где он исполнял партию <партия>Иуды</партия>.
</выступление>
</абзац>
</резюме>
```

Таблица стилей

Файл таблицы стилей – `солист.xsl`.

При представлении этого текста для читателей основная задача состоит в том, чтобы выбрать типографские соглашения, которые будут использованы для каждой части семантической разметки. Можно, например, отображать названия произведений курсивом, заголовки публикаций – шрифтом без засечек, а имена композиторов – обыкновенным шрифтом

основного текста. Это может быть достигнуто с помощью следующих определений таблицы стилей (предполагаемый вывод – HTML):

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html><body>
<xsl:apply-templates/>
</body></html>
</xsl:template>

<xsl:template match="абзац">
   <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="публикация">
   <font face="arial"><xsl:apply-templates/></font>
</xsl:template>

<xsl:template match="цитата">
   <xsl:text/>"<xsl:apply-templates/>"<xsl:text/>
</xsl:template>

<xsl:template match="произведение">
   <i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="партия">
   <u><xsl:apply-templates/></u>
</xsl:template>

</xsl:stylesheet>
```

Заметьте, что часть разметки игнорируется, например <певец>. Заданный по умолчанию шаблон для элементов просто отбраковывает теги и выводит именно тот текст, к которому стремились.

Вывод

Если сгенерированный HTML скопировать в текстовой процессор, он будет выглядеть так:

> Обзор старинной музыки рассказал о дебютном концерте на сцене Вигмор (Wigmore) с Ансамблем Соннери в 1998 году: «Один из самых прекрасных концертов, которые мне доводилось слушать ... певец, которого нельзя пропустить». В число других его ярких выступлений входят трансляция по телевидению произведения *«Страсти по Матфею»* (И. С. Бах) в постановке Джонатана Миллера, где он исполнял партию Иуды.

(Почему-то такой способ получения печатного вывода из таблицы стилей XSLT часто недооценивается. Он не дает таких больших возможностей форматирования, какие предполагаются в XSL Formatting, но пока этот стандарт еще не принят окончательно, и соответствующих программ еще

нет, использование HTML в качестве промежуточного формата – неплохой компромисс.)

Большим преимуществом этого подхода является то, что правила пишутся без предъявления требований к способу вложения тегов разметки в исходный документ. Очень просто добавить новые правила для новых тегов, а также многократно использовать правила, когда имеющийся тег используется в другом контексте.

В структурах документов, в которых вложение элементов осуществляется более жестко, например в некоторых файлах для обмена данными, этот гибкий стиль обработки, основанный на правилах **(форсированный)** дал бы меньше выгод. Там более предпочтителен **извлекающий** стиль программирования с использованием традиционных конструкций потока команд, подобных <xsl:for-each>, <xsl:if> и <xsl:call-template>. Более подробное обсуждение различных подходов к проектированию см. в главе 9.

Использование режимов

Классическая ситуация для использования режимов – когда одно и то же содержимое требуется обработать несколько раз различными способами. Например, в первом проходе документа можно сгенерировать оглавление, во втором проходе – основной текст, а в третьем проходе – предметный указатель.

Пример: Использование режимов

Исходный документ – биография певца, в том же формате, что и предыдущий пример. Однако на этот раз в конце биографии нужно представить список произведений, упомянутых в тексте.

Исходный файл

Исходный файл – солист.xml (см. предыдущий пример).

Таблица стилей

Файл таблицы стилей – солист+индекс.xsl.

Данная таблица стилей расширяет предыдущую, используя <xsl:import>. После вывода текста, как в прошлом примере, она еще создает таблицу, в которой перечислены выступления певца с указанием исполняемых произведений, имен композиторов, даты и места выступления.

Символы « » (авторам HTML они больше известны как неразрывные пробелы, « ») используются для гарантии того, что в каждой ячейке таблицы содержится хоть что-то: это обеспечивает более аккуратное оформление данных в броузере. Вполне можно использовать ссылку на сущность « » в таблице стилей, если она должным образом указана в начале файла в объявлении <!DOCTYPE> как XML-сущность.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:import href="солист.xsl"/>

<xsl:template match="/">
<html><body>
    <xsl:apply-templates/>
    <table bgcolor="#cccccc" border="1" cellpadding="5">
    <tr>
        <td><b>Дата</b></td>
        <td><b>Сцена</b></td>
        <td><b>Композитор</b></td>
        <td><b>Произведение</b></td>
        <td><b>Партия</b></td>
    </tr>
    <xsl:apply-templates mode="индекс"/>
    </table>
</body></html>
</xsl:template>

<xsl:template match="выступление" mode="index">
    <tr>
    <td><xsl:value-of select="дата"/> </td>
    <td><xsl:value-of select="сцена"/> </td>
    <td><xsl:value-of select="композитор"/> </td>
    <td><xsl:value-of select="произведение"/> </td>
    <td><xsl:value-of select="роль"/> </td>
    </tr>
</xsl:template>

</xsl:stylesheet>
```

Вывод

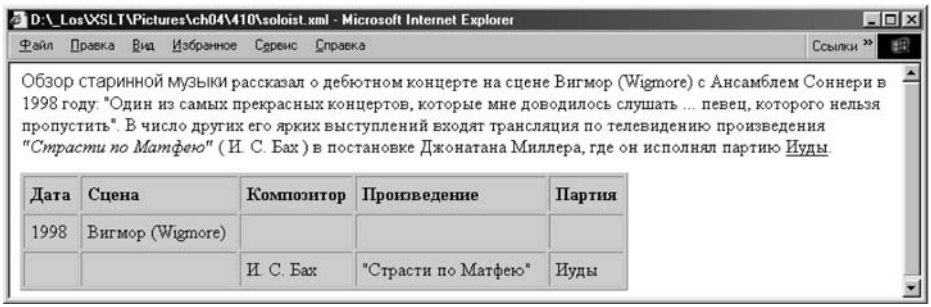


Рис. 4.10. *Полученный вывод в окне броузера*

Использование именованных шаблонов

Именованные шаблоны в XSLT служат эквивалентом механизма использования подпрограмм. Если шаблон имеет атрибут `name`, его можно вызвать с помощью инструкции `<xsl:call-template>`.

Примеры использования именованных шаблонов см. в разделе `<xsl:call-template>` на стр. 204.

См. также

<xsl:apply-templates> на стр. 178

<xsl:apply-imports> на стр. 174

<xsl:call-template> на стр. 204

Функция generate-id() в главе 7

xsl:text

Инструкция <xsl:text> используется в пределах тела шаблона для воспроизведения текстов в текущем адресате вывода.

Определен в

XSLT, раздел 7.2

Формат

```
<xsl:text disable-output-escaping="yes" | "no">
   текст ?
</xsl:text>
```

Расположение

<xsl:text> – инструкция, и всегда используется в пределах тела шаблона.

Атрибуты

Имя	Значение	Назначение
disable-output-escaping необязательный	«yes» | «no»	Значение «yes» указывает, что специальные символы (такие как «<») должны выводиться как есть, а не с использованием принятой в XML формы «<». Значение по умолчанию – «no».

Содержимое

Текстовый узел. Элемент также может быть пустым. Он не может содержать другие элементы, такие как <xsl:value-of>.

Действие

Текст, встречающийся в таблице стилей в пределах тела шаблона, копируется в текущее назначение вывода, независимо от того, включен он в <xsl:text> или нет. Единственный прямой эффект от включения текста в элемент <xsl:text> в том, что в этом случае иначе происходит обработка пробельных символов. Узел с пробельными символами, находящийся в таблице стилей

(то есть текстовый узел, который состоит только из пробельных символов) копируется в конечное дерево, если только:

- он находится в пределах элемента <xsl:text>, или
- включающий его элемент имеет атрибут «xml:space="preserve"», и действие этого атрибута не отменяется другим внутренним элементом, у которого определено «xml:space="default"».

Атрибут disable-output-escaping контролирует, нужно ли экранировать специальные символы, такие как «<» (то есть преобразовывать их в ссылки на символы или в ссылки на сущности типа «<»), если они встречаются в тексте. Значение по умолчанию – «no». Значение «yes» при некоторых обстоятельствах может игнорироваться, например, если результат преобразования записывается в DOM, а не сериализуется как файл XML.

Использование

Применение элемента <xsl:text> обеспечивает две основные функции: контроль над выводом пробельных символов и блокировку преобразования специальных символов в ссылки на символы. Обе функции обсуждаются в двух следующих разделах.

Контроль над пробельными символами

Наиболее очевидная ситуация, в которой полезен элемент <xsl:text>, – принудительный вывод пробельных символов. Пример дается в спецификации XSLT. Если написать:

```
<xsl:value-of select="имя"/> <xsl:value-of select="фамилия"/>
```

пробел между именем и фамилией будет потерян, потому что это – часть узла, который содержит только пробельные символы (единственный пробел). Чтобы этот пробел стал принудительным и переносился в вывод между именем и фамилией, следует написать так:

```
<xsl:value-of select="имя"/>
<xsl:text> </xsl:text>
<xsl:value-of select="фамилия"/>
```

Разбиение на три строки здесь сделано просто для удобства чтения, но это не влияет на вывод, потому что символы новой строки находятся в узлах только из пробельных символов, которые не будут выводиться.

Если предыдущий код выглядит многословным, можно добиться того же эффекта иначе:

```
<xsl:value-of select="concat(имя, ' ', фамилия)"/>
```

Функция concat() формирует строку путем сцепления своих параметров; это описано в главе 7.

Еще одна проблема – предотвращение вывода нежелательных пробельных символов. К счастью, в HTML лишние пробелы обычно не имеют значения,

так как броузер их игнорирует. Однако для XML или текстового вывода это может быть важно.

Если в вывод попадают излишние пробельные символы, первое, что нужно сделать – выяснить, берутся ли они из исходного документа или из таблицы стилей. Если пробельные символы соседствуют с текстом, скопированным из исходного документа, то, вероятно, они взяты оттуда; если они находятся рядом с текстом, взятым из таблицы стилей, то наиболее вероятный их источник – таблица стилей.

Если нежелательные пробельные символы взяты из исходного документа, можно использовать <xsl:strip-space>, чтобы удалить узлы, состоящие только из пробельных символов, или использовать функцию normalize-space(), чтобы удалить начальные и концевые пробелы в строках текста.

Элемент <xsl:text> можно использовать, чтобы подавить нежелательные пробельные символы, происходящие из таблицы стилей. Например, рассмотрим следующий шаблон:

```
<xsl:template match="stage-direction">
   [ <xsl:value-of select="."/> ]
</xsl:template>
```

Цель этого шаблона – вывести сценические ремарки в квадратных скобках. Однако текстовые узлы, содержащие открывающие и закрывающие квадратные скобки, содержат также символ перехода на новую строку и несколько пробелов, которые запишутся в адресат вывода вместе со скобками. Чтобы предотвратить это, проще всего добавить до и после них пустые элементы <xsl:text>, следующим образом:

```
<xsl:template match="stage-direction">
   <xsl:text/>[ <xsl:value-of select="."/> ]<xsl:text/>
</xsl:template>
```

Благодаря такому приему, лишние пробелы и символы перевода строки теперь принадлежат узлам только из пробельных символов, которые зачищаются из таблицы стилей и игнорируются.

Заметьте, что нельзя окружать элементами <xsl:text> элемент <xsl:value-of>, так как элементы <xsl:text> должны содержать только текстовые данные. Таким образом, следующий шаблон будет неправильным:

```
<!-- НЕВЕРНО -->
     <xsl:text>[ <xsl:value-of select="."/> ]</xsl:text>
<!-- НЕВЕРНО -->
```

Контроль над экранированием символов в выводе

Обычно при выводе в текстовом узле специальных символов типа «<» или «&» эти символы преобразуются в выходном файле с помощью специального механизма в XML. XSLT-процессор может выбрать любой механизм: он может записать «<» как «<» или как «<», или как «<![CDATA[<]]>», – пото-

му что все эти обозначения эквивалентны, согласно стандарту XML. Единственное, что он не напишет – «<». Не имеет значения, в какой форме «<» находится во вводе: XSLT-процессор видит «<» и преобразует этот символ в выводе.

Существует ряд ситуаций, когда такие преобразования символов нежелательны. Например:

- Выводом является не XML или HTML, а, скажем, файл данных с разделителями-запятыми.
- Выводом является HTML, и желательно воспользоваться одной из множества особенностей HTML, когда специальные символы необходимы без преобразования, например символ «<» в клиентской части JavaScript на вашей HTML-странице.
- Вывод – XML, и нужно добиться некоторого специального эффекта, которого XSLT-процессор не позволяет достичь обычным путем. Например, нужно вывести ссылку на сущность «&текущая-дата;» или объявление типа документа, содержащее объявление сущности.
- Вывод – некоторый формат, который использует синтаксис с угловыми скобками, но не чистый XML или HTML. Например, это формат ASP-страниц или Java Server Pages, которые используют в качестве разделителей «<%» и «%>». (Но при генерировании Java Server Pages имейте в виду, что они имеют альтернативный синтаксис, который является чистым XML.)

Если вывод – вообще не XML или HTML, то вместо использования `disable-output-escaping` лучше установить для элемента `<xsl:output>` атрибут «`method="text"`».

Ниже приведен пример неправильного применения экранирования символов при выводе. Задача – перенести теги разметки в документ вывода, чего не так просто достичь обычными средствами `<xsl:element>` или конечных литеральных элементов. Например, может прийти в голову такая идея:

```
<!-- НЕВЕРНО -->
<xsl:template match="маркер"/>
   <xsl:if test='not(preceding::*[self::пункт-списка])'>
      <ul>
   </xsl:if>
   <li><xsl:value-of select="."/></li>
   <xsl:if test='not(following::*[self::пункт-списка])'>
      </ul>
   </xsl:if>
</xsl:template>
<!-- НЕВЕРНО -->
```

Предполагалось, что этот шаблон будет выводить тег `<ul>`, если предшествующий элемент не является пунктом списка, и тег `</ul>`, когда следующий элемент не является пунктом списка. Конечно, эти ожидания напрасны, потому что теги `<ul>` и `</ul>` не вложены должным образом: этот шаблон будет отвергнут синтаксическим анализатором XML еще до вызова в XSLT-процессор.

Тогда можно попробовать записать теги как текст, следующим образом:

```
<xsl:template match="маркер"/>
   <xsl:if test='not(preceding::*[self::пункт-списка])'>
      <xsl:text disable-output-escaping="yes">&lt;ul&gt;</xsl:text>
   </xsl:if>
   <li><xsl:value-of select="."/></li>
   <xsl:if test='not(following::*[self::пункт-списка])'>
      <xsl:text disable-output-escaping="yes">/&lt;ul&gt;</xsl:text>
   </xsl:if>
</xsl:template>
```

Эта запись синтаксически правильна и отвечает стандарту XML и XSLT, но нет гарантии, что она сработает в любой ситуации. В частности, она может потерпеть неудачу, если существует еще какой-то контроль структуры вывода XML. Таким образом, это просто трюк, который то ли получится, то ли нет.

Если немного подумать, можно найти способ достижения желаемого вывода.

Во-первых, нужно думать о конечном дереве, содержащем узлы, а не о текстовом файле, содержащем теги. Не пытайтесь генерировать открывающий тег `<ul>` и закрывающий тег `</ul>` в два отдельных этапа; нужно генерировать узел элемента `<ul>` как единое действие, а затем генерировать его потомков.

Цель состоит в том, чтобы произвести элемент `<ul>` для каждого элемента `<маркер>` в источнике, которому не предшествует другой такой же элемент, поэтому можно начать так:

```
<xsl:template match="маркер[not(preceding-sibling::*[1][self::маркер])]">
   <ul>
      . . .
   </ul>
</xsl:template>
```

Это шаблонное правило соответствует каждому элементу `<маркер>`, который или не имеет предшествующего одноуровневого элемента, или которому предшествует одноуровневый элемент, но не `<маркер>`.

Для каждого такого маркера `<маркер>` нужно обработать список последующих одноуровневых элементов, пока не попадется следующий элемент – не маркер. Лучше всего делать это с помощью рекурсивного шаблона:

```
<xsl:template
          match="маркер[not(preceding-sibling::маркер)]">
   <ul>
      <xsl:call-template name="маркированный-список"/>
   </ul>
</xsl:template>

<xsl:template name="маркированный-список">
   <li>
      <xsl:value-of select="."/>
   </li>
```

```
    <xsl:variable name="next" select="following-sibling::*[1]"/>
    <xsl:for-each select="$next[self::маркер]">
       <xsl:call-template name="маркированный-список"/>
    </xsl:for-each>
</xsl:template>
```

Чем обрабатывать последующие маркеры по циклу, лучше воспользоваться рекурсией. Шаблон списка маркера выводит текущий элемент <маркер>, и если следующий за ним элемент – тоже <маркер>, шаблон вызывает себя для обработки и этого элемента. Заметьте, что здесь элемент <xsl:for-each> не выполняет цикл: он обрабатывает наборы узлов, в которых содержится один элемент, если следующий элемент – тоже <маркер>, или нуль элементов – в противном случае.

Примеры

1) Вывод имен и фамилий, разделенных пробелом:

```
<xsl:value-of select="имя"/>
<xsl:text> </xsl:text>
<xsl:value-of select-"фамилия"/>
```

Другой способ достичь того же эффекта состоит в использовании функции concat():

```
<xsl:value-of select="concat(имя, ' ', фамилия)"/>
```

2) Вывод списка значений, разделенных запятыми:

```
<xsl:output method="text"/>
<xsl:template match="книга">
   <xsl:value-of select="название"/>,<xsl:text/>
   <xsl:value-of select="автор"/>,<xsl:text/>
   <xsl:value-of select="цена"/>,<xsl:text/>
   <xsl:value-of select="isbn"/><xsl:text>
</xsl:text>
</xsl:template>
```

Цель пустых элементов <xsl:text/> – разбить запятую и следующий за ней символ перевода строки на отдельные текстовые узлы; это гарантирует, что символ перевода строки станет частью узла только пробельных символов, и поэтому не будет копироваться в вывод. Концевой элемент <xsl:text> гарантирует, что в конце каждой записи будет записан символ перевода строки.

3) Вывод ссылки на сущность « » :

```
<xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
```

Заметьте, что вывод символа #xa0 (или #160) обычно имеет такой же эффект; для этого можно просто написать:

```
<xsl:text> </xsl:text>
```

См. также

<xsl:value-of> на стр. 366

xsl:transform

Это синоним элемента <xsl:stylesheet>, описанного на стр. 337. Оба эти имени элемента могут использоваться взаимозаменяемо.

Почему полезно иметь два имени для одного элемента? Вероятно, потому что для комитета по стандартам так было меньше хлопот. Ну, а если серьезнее, существование этих двух имен является показателем того, что некоторые видят в XSLT, прежде всего, язык для преобразования деревьев, в то время как другие видят его основную роль в определении стилей отображения. Выбирайте любое.

Формат

```
<xsl:transform
   id=идентификатор
   version="1.0" or "1.1"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   extension-element-prefixes=список-префиксов
   exclude-result-prefixes=список-префиксов >
      элемент-верхнего-уровня *
</xsl:transform>
```

Определен в

XSLT, раздел 2.2

См. также

<xsl:stylesheet> на стр. 337

xsl:value-of

Инструкция <xsl:value-of> записывает строковое значение выражения в конечное дерево.

Определен в

XSLT, раздел 7.6.1

Формат

```
<xsl:value-of select=Выражение
              disable-output-escaping="yes" | "no"/>
```

Расположение

<xsl:value-of> – инструкция, и всегда используется в пределах тела шаблона.

Атрибуты

Имя	Значение	Назначение
select обязательный	выражение	Значение, которое должно быть выведено.
disable-output-escaping необязательный	«yes» | «no»	Значение «yes» указывает, что специальные символы (такие как «<») должны выводиться, как есть, а не с использованием принятой в XML формы «<». Значение по умолчанию – «no».

Содержимое

Никакого; элемент всегда пуст.

Действие

Выражение вычисляется как строковое значение, а результат выводится в конечное дерево как текстовый узел. В случае необходимости значение сначала преобразовывается в строку, следующим образом:

- Если значение – логический тип данных, выводом будет либо «true», либо «false».
- Если значение – численный тип данных, оно сначала преобразуется в строковое представление числа в десятичной системе счисления, например «93.7». (Для управления форматированием чисел используйте вместо этого элемент <xsl:number> или функцию format-number().)
- Если значение – набор узлов, то **все узлы кроме первого (в порядке документа)** игнорируются. Если набор узлов пуст, ничего не выводится. В противном случае выводится строковое значение первого узла. Строковое значение узла зависит от типа узла: для текстового узла – это текстовое содержимое; для узла атрибута – значение атрибута; для комментария – текст комментария; для инструкции обработки – данные инструкции обработки (исключая *цель*). Для узла элемента – сцепление значений всех потомственных текстовых узлов. Другими словами, это текст, содержащийся в элементе, без всякой разметки и без атрибутов. Аналогично, для корня строковое значение – это весь текст документа с удаленной разметкой.
- Обратите внимание на заметную разницу между набором узлов, который содержит много элементов, и набором узлов, который содержит единственный элемент, имеющий много прямых потомков. Выражение «//para» возвращает набор узлов, содержащий все элементы <para> в документе. Строковое значение этого – строковое значение первого элемента <para>, поэтому <xsl:value-of select="//para"> отобразит только один абзац. В противоположность этому, выражение «/» представляет набор узлов, содер-

жащий только один узел – корень. Его строковое значение – весь текст в документе, поэтому <xsl:value-of select="/"/> выведет весь этот текст.

- Если значение – временное дерево, выводом будет строковое значение корневого узла дерева. В действительности это означает, что выводом будет сцепление всех текстовых узлов в пределах дерева. Если в адресат вывода нужно копировать узлы дерева, а не только их строковое значение, используйте элемент <xsl:copy-of>, описанный на стр. 221.

Атрибут disable-output-escaping действует так же, как с элементом <xsl:text>. Специальные символы, такие как «<», в строковом значении выражения select будут преобразованы так же, как если бы они находились в тексте, и атрибут disable-output-escaping можно таким же образом использовать здесь для блокировки преобразования. Подробнее об этом см. в разделе <xsl:text> на стр. 360.

Использование

Элемент <xsl:value-of> обеспечивает наиболее общий путь записи текста в конечное дерево (или в текущий адресат вывода, если они различны).

Другие способы записи текстовых узлов в конечное дерево заключаются в буквальном включении текста в таблицу стилей (возможно, в пределах инструкции <xsl:text>) или в использовании <xsl:copy> или <xsl:copy-of>. Удивительно, но можно вообще обходиться без <xsl:value-of>, ведь <xsl:value-of select="X"/> всегда можно заменить на <xsl:copy-of select="string(X)"/>!

Еще одной альтернативой может быть использование в текстовом узле <xsl:apply-templates> и встроенного шаблона для текстовых узлов, что является эквивалентом <xsl:value-of select="."/>.

Инструкция <xsl:value-of> часто служит эффективным вариантом управления исходным деревом, рекурсивно использующим <xsl:apply-templates>. Например:

```
<xsl:template match="книга">
   <книга>
      <издатель><xsl:value-of select="../@название"/></издатель>
      <название><xsl:value-of select="@название"/></название>
      <автор><xsl:value-of select="@автор"/></автор>
      <isbn><xsl:value-of select="@isbn"/></isbn>
   </книга>
</xsl:template>
```

Поскольку инструкция <xsl:value-of> записывает в текущее назначение вывода, а необязательно в конечное дерево, она используется для вывода результата, полученного от вызываемого шаблона. Например, можно написать именованный шаблон, который заменяет символы в имени файла следующим образом:

```
<xsl:template name="изменить-имя-файла">
   <xsl:param name="имя-файла"/>
```

```
    <xsl:value-of select="translate($имя-файла, '\', '/')"/>
</xsl:template>
```

Этот шаблон может затем вызываться, выдавая полученный результат в переменную $новое-имя-файла:

```
<xsl:variable name="$новое-имя-файла">
   <xsl:call-template name="изменить-имя-файла">
      <xsl:with-param name="$старое-имя-файла"/>
   </xsl:call-template>
</xsl:variable>
```

Фактически значением переменной $новое-имя-файла является временное дерево, но для всех практических целей его можно рассматривать как строковое значение.

Использование disable-output-escaping

Очень часто HTML заключают внутрь XML-документа как в оболочку, например:

```
<сообщение>
  <заголовок>
    <отправлено>Отдел 178</отправлено>
    <получатель>Офис 263</получатель>
  </заголовок>
  <содержимое тело="text/html">
    <![CDATA[
       <html>
          <head><title>HTML-страница с несбалансированными тегами</title></head>
          <body>Авторы HTML-посланий часто ленивы!<p></body>
       </html>
    ]]>
  </содержимое>
</сообщение>
```

Один из способов включения HTML в XML-сообщение состоит в использовании для этого утилиты Дейва Раггетта (Dave Raggett) **html tidy** (доступной на *http://www.w3.org/*), которая конвертирует HTML в правильно построенный XHTML. Но если опасаетесь риска изменить исходный HTML, тогда остается единственный вариант – использовать секции CDATA, как показано здесь. Однако в этом случае символы «<» и «>» уже рассматриваются не как символы разметки, а как обыкновенный текст. Если выполнить XSLT-преобразование, которое выводит HTML, включенный в это сообщение, эти символы будут преобразованы в выводе, обычно, к форме «<» и «>». Если такой вывод нежелателен, можно решить проблему следующим образом:

```
<xsl:template match="содержимое[@тип='text/html']">
  <xsl:value-of select="." disable-output-escaping="yes"/>
</xsl:template>
```

Следует помнить, что этот вариант возможен, если только вывод сериализуется XSLT-процессором; если же результат записывается в DOM, это не будет работать.

Примеры

`<xsl:value-of select="."/>`	Выводит строковое значение текущего узла.
`<xsl:value-of select="название"/>`	Выводит строковое значение первого прямого потомка элемента `<название>` текущего узла.
`<xsl:value-of select="sum(@*)"/>`	Выводит сумму значений атрибутов текущего узла, преобразованную в строку. Если имеется любой нечисловой атрибут, результатом будет `"NaN"`.
`<xsl:value-of select="$x"/>`	Выводит значение переменной `$x` после преобразования его в строку.

См. также

`<xsl:copy-of>` на стр. 221

`<xsl:text>` на стр. 360

xsl:variable

Элемент `<xsl:variable>` используется в таблице стилей для объявления локальной или глобальной переменной и присваивания ей значений.

Определен в

XSLT, раздел 11

Формат

```
<xsl:variable name=ПолноеИмя select=Выражение >
   template body
</xsl:variable>
```

Расположение

Элемент `<xsl:variable>` может встречаться или как элемент верхнего уровня (то есть как прямой потомок элемента `<xsl:stylesheet>`), или как инструкция в пределах тела шаблона.

Атрибуты

Имя	Значение	Назначение
`name` обязательный	`полное имя`	Имя переменной

Имя	Значение	Назначение
select необязательный	выражение	Выражение, которое вычисляется, давая значение переменной. Если оно опущено, то значение определяется из содержимого элемента <xsl:variable>

Конструкции полное имя и выражение определены в главе 5.

Содержимое

Тело шаблона (необязательно). Если присутствует атрибут select, элемент <xsl:variable> должен быть пустым.

Действие

Элемент <xsl:variable> может находиться или на верхнем уровне таблицы стилей (когда он объявляет глобальную переменную), или, как инструкция, в пределах тела шаблона (когда он объявляет локальную переменную).

Значение переменной

Значение переменной можно задавать или выражением в атрибуте select, или в содержимом тела шаблона. Если есть атрибут select, элемент <xsl:variable> должен быть пустым. Если атрибута select нет, и тело шаблона пусто, значение переменной – пустая строка.

Если значение переменной задается выражением, тип данных будет логическим, численным, строковым или набором узлов, в зависимости от выражения. Если выражение вызывает функцию расширения, значением переменной может быть также *внешний объект*. Если значение задается непустым телом шаблона, значением переменной будет временное дерево. В XSLT 1.0 оно считается отдельным типом данных, **фрагментом конечного дерева**. Однако в XSLT 1.1 временное дерево рассматривается как значение набора узлов, причем этот набор узлов всегда содержит единственный узел: корневой узел временного дерева.

Заметьте, что если выражение используется для назначения переменной буквального строкового значения, строковый литерал должен быть заключен в кавычки, и это – дополнительные кавычки к тем, в которые заключается XML-атрибут. Так, присвоить значение «Лондон» переменной, имя которой «город», можно любым из следующих выражений:

```
<xsl:variable name="город" select="'Лондон'"/>
<xsl:variable name="город" select='"Лондон"'/>
```

Можно написать и так:

```
<xsl:variable name="город">Лондон</xsl:variable>
```

Формально значение будет скорее набором узлов, чем строкой, но оно может использоваться так, будто это строка, поскольку нет практически никакой разницы.

Типичной ошибкой является следующая запись:

```
<xsl:variable name="город" select="Лондон"/> <!-- НЕВЕРНО -->
```

Она делает значением «$город» набор узлов, содержащий всех непосредственных потомков текущего узла элемента, которые имеют имя элемента <Лондон>. Вероятно, это будет пустой набор, так как если использовать переменную как строку, ее значением будет пустая строка. При этом процессор не будет сообщать об ошибке, потому что формально такая запись правильна, но в результате таблица стилей будет производить неправильный вывод.

Такая ошибка была сделана даже в оригинале спецификации XSLT 1.0, в примерах; она должна быть уже устранена в последующих исправлениях.

Дополнительные кавычки не нужны, если требуется, чтобы значение было числовым:

```
<xsl:variable name="максимальный-размер" select="255"/>
```

Имя переменной

Имя переменной определяется с помощью полного имени. Обычно это будет простое имя, например, «город» или «сумма-продаж», но также это может быть имя, уточненное префиксом, например «мое:значение». Если имя имеет префикс, этот префикс должен соответствовать пространству имен, которое действует в этом месте таблицы стилей. Истинное имя переменной для проверки идентичности двух имен определяется не префиксом, а URI пространства имен, который соответствует данному префиксу. Таким образом, имена двух переменных, «мое:значение» и «ваше:значение», будут идентичны, если префиксы «мое» и «ваше» относятся к одному и тому же URI пространства имен. Если имя не имеет никакого префикса, оно имеет пустой URI пространства имен, так как оно не использует URI пространства имен по умолчанию.

Область действия глобальной переменной – вся таблица стилей, включая любые таблицы стилей, которые включены или импортированы. На глобальную переменную можно ссылаться даже прежде, чем она объявлена. Единственное ограничение – не допускаются циклические определения: если переменная x определена на основе y, то y не может быть определен, прямо или косвенно, на основе x.

Область действия локальной переменной имеет блочную структуру: на нее может ссылаться любой последующий равноуровневый элемент или его потомки. Это проиллюстрировано на схеме ниже.

На схеме отмечена переменная X, и показаны элементы, которые могут содержать ссылку на X: затененные элементы могут ссылаться на эту переменную, а незатененные элементы – не могут. В частности, на локальную переменную может ссылаться любой последующий одноуровневый элемент или его потомки. Собственные потомки переменной не могут ссылаться на нее. С другой стороны, область действия переменной ограничивается закрывающим тегом ее родительского элемента. В отличие от глобальных переменных, опережающая ссылка на локальную переменную недопустима. Если

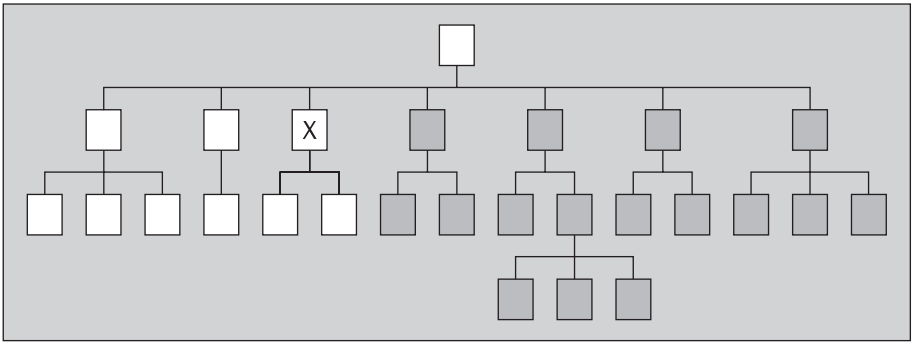


***Рис. 4.11.** Область действия локальной переменной: блочная структура*

усмотреть аналогию между инструкциями XSLT и инструкциями языков с блочной структурой – C, Java или JavaScript – и рассматривать элемент, включающий в себя другие элементы, как аналог блока, заключаемого в тех языках в фигурные скобки, правила определения области действия переменной покажутся очень знакомыми.

Две глобальные переменные могут иметь одинаковое имя, если только они имеют различное *преимущество импортирования*; другими словами, если одна из них находится в импортированной таблице стилей (дальнейшие подробности см. в разделе `<xsl:import>` на стр. 256). В этом случае доминирует определение с более высоким преимуществом импортирования (в оригинале спецификации XSLT 1.0 забыли сказать об этом, но это было отмечено в исправлениях). Заметьте, что определение с более высоким преимуществом импортирования применяется всюду, даже в импортированной таблице стилей, которая содержит определение с более низким преимуществом импортирования. Это означает, что нельзя всецело полагаться на преимущество импортирования при разрешении случайных противоречий в именах между независимо разработанными модулями. Лучше использовать для этого пространства имен.

Две локальные переменные могут иметь одинаковое имя, если только ни одна из них не определена внутри области действия другой. Однако локальная переменная может иметь такое же имя, как глобальная переменная, но в этом случае глобальная переменная недостижима в области действия локальной переменной.

Эти правила по уникальности и области действия имен применяются аналогично и к параметрам, объявленным с помощью инструкции `<xsl:param>`. Это просто еще один способ объявления переменной.

Использование

Как в любом языке программирования, переменные позволяют не производить повторно одни и те же вычисления.

Глобальные переменные удобны для определения констант, таких как цвета, которые будут использоваться во многих местах всей таблицы стилей.

В отличие от переменных во многих языках программирования, **XSLT-переменные** не могут быть модифицированы. Получив начальное значение, они сохраняют его, пока не выходят из области действия. Эта особенность имеет глубокое влияние на стиль программирования, используемый, когда таблица стилей должна производить вычисления. Вопрос о *программировании без выражений присваивания* подробно обсуждается в главе 9.

Примеры

Большинство XSLT-переменных относится к одной из трех категорий:

- Переменные, позволяющие избегать повторений общих выражений. Они вводятся, чтобы облегчить чтение кода или чтобы гарантировать, что при изменении значения придется вносить исправления только в одном месте, или, возможно, чтобы повысить эффективность обработки.
- Переменные, используемые для фиксирования контекстно-зависимой информации, что позволяет использовать переменную и после того, как контекст изменился.
- Переменные, сохраняющие временное дерево (фрагмент конечного дерева, как его обычно называют).

Во всех трех случаях переменная может быть локальной или глобальной. Здесь будут приведены примеры переменных каждого вида.

Переменные для удобства

Рассмотрим пример, который вычисляет количество мячей, забитых и пропущенных футбольной командой.

```
<xsl:variable name="за" select="sum($матчи/команда[.=$this]/@очки)"/>
<xsl:variable name="против"
              select="sum($матчи[команда=$this]/команда/@очки) - $за"/>
. . .
<td><xsl:value-of select="$за"/></td>
<td><xsl:value-of select="$против"/></td>
```

Для конструирования переменных «за» и «против», вычисляющих количество мячей, забитых и пропущенных командой, которая задается переменной «$команда», здесь использованы два довольно сложных выражения. Для лучшего понимания логики этого примера он дается как полностью работающий пример в разделе, описывающем функцию `sum()`, в главе 7.

В этом случае вполне можно обойтись без переменной «против». Выражение, которое вычисляет ее значение, можно с тем же успехом записать в том месте, где переменная используется: во второй инструкции `<xsl:value-of>`. То же самое справедливо и для переменной «за», хотя на этот раз выражение пришлось бы писать дважды: в обоих местах, где используется переменная,

а это не лучшим образом отразится на скорости обработки. Более того, эти переменные в действительности используются только для наглядности; вполне можно написать таблицу стилей и без них.

Это так, потому что ничто не может измениться между моментами определения и использования переменных. Исходный документ не может измениться, и значения переменных `$команда` и `$матчи` не могут измениться. Контекст (например, текущая позиция в исходном документе) может измениться, но в данном примере: (a) он не меняется и (b) выражения никак не зависят от контекста.

Здесь эти переменные названы *переменными для удобства*, потому что можно обойтись и без них, если потребуется (хотя могут пострадать характеристики обработки). Их можно использовать или как глобальные, или как локальные переменные. Создание глобальных переменных для удобства, которые относятся к наборам узлов в исходном документе, часто оказывается полезным приемом программирования, например:

```
<xsl:variable name="матчи-в-группе-A" select="//матч[@группа='A']"/>
```

Они действуют аналогично представлениям в базе данных SQL.

Переменные для фиксирования контекстно-зависимых значений

Эти переменные чаще всего полезны в сочетании с элементом `<xsl:for-each>`, который изменяет текущий узел. Рассмотрим следующий пример.

Пример: Использование переменной для фиксирования контекстно-зависимых значений

Исходный файл

Исходный файл – `опера.xml`. Он содержит список опер и данные о композиторах.

```
<?xml version="1.0"?>
<программа>
   <опера>
      <название>The Magic Flute</название>
      <композитор>Mozart</композитор>
      <дата>1791</дата>
   </опера>
   <опера>
      <название>Don Giovanni</название>
      <композитор>Mozart</композитор>
      <дата>1787</дата>
   </опера>
   <опера>
      <название>Ernani</название>
      <композитор>Verdi</композитор>
```

```
      <дата>1843</дата>
   </опера>
   <опера>
      <название>Rigoletto</название>
      <композитор>Verdi</композитор>
      <дата>1850</дата>
   </опера>
   <опера>
      <название>Tosca</название>
      <композитор>Puccini</композитор>
      <дата>1897</дата>
   </опера>
   <композитор имя="Mozart">
      <полное-имя>Wolfgang Amadeus Mozart</полное-имя>
      <родился>1756</родился>
      <умер>1791</умер>
   </композитор>
   <композитор имя="Verdi">
      <полное-имя>Guiseppe Verdi</полное-имя>
      <родился>1813</родился>
      <умер>1901</умер>
   </композитор>
   <композитор имя="Puccini">
      <полное-имя>Giacomo Puccini</полное-имя>
      <родился>1858</родился>
      <умер>1924</умер>
   </композитор>
</программа>
```

Таблица стилей

Таблица стилей – файл `опера.xsl`. Это полноценная таблица стилей; в ней используется упрощенный синтаксис таблицы стилей, описанный в главе 3.

Таблица стилей содержит два вложенных цикла `<xsl:for-each>`. Во внешнем цикле она устанавливает переменную «`c`» для контекстного узла (текущий композитор). Эта переменная используется в выражении, управляющем внутренним циклом. Было бы неправильно использовать «`.`» вместо «`$c`», потому что элемент `<композитор>` – больше не контекстный узел. В этом примере можно было бы воспользоваться функцией `current()` (эта функция описана в главе 7), но существуют другие случаи, когда необходима именно переменная.

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xsl:version="1.0">

<body><center>
   <h1>Программа</h1>
   <xsl:for-each select="/программа/композитор">
       <h2><xsl:value-of
            select="concat(полное-имя, ' (', родился, '-', умер, ')')"/></h2>
       <xsl:variable name="c" select="."/>
```

```
        <xsl:for-each select="/программа/опера[композитор=$c/@имя]">
            <p><xsl:value-of select="название"/></p>
        </xsl:for-each>
    </xsl:for-each>
</center></body>
</html>
```

Вывод

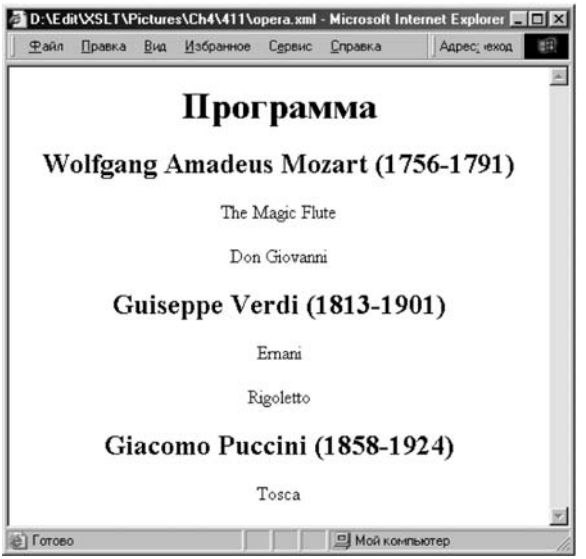


Рис. 4.12. *Полученный вывод в окне броузера*

Контекстные переменные очень полезны, когда нужно обрабатывать несколько исходных документов.

В любую таблицу стилей, предназначенную для обработки множественных исходных документов, полезно включить глобальную переменную, которая относится к корневому узлу основного исходного документа. Это делается так:

```
<xsl:variable name="корень" select="/"/>
```

Используя контекстную глобальную переменную, всегда можно обратиться к исходному документу. Если же такой переменной нет, то к данным из главного документа нет никакого доступа, когда контекстный узел находится во вторичном документе.

Например, выражение «`//item`» относится ко всем элементам `<item>` в том же документе, где находится контекстный узел. Если нужно учесть все элементы `<item>` в основном исходном документе, то (при наличии объявления глобальной переменной, приведенного выше) можно использовать выражение «`$корень//item`».

Если таблица стилей ссылается на какой-то документ, содержащий нужные данные – сообщения или налоговые ставки, например, – также полезно определить его в глобальной переменной:

```
<xsl:variable name="налоговые-ставки" select="document('налоговые-ставки.xml')"/>
```

Временные деревья

Если переменная определена при использовании содержимого элемента `<xsl:variable>`, а не атрибута `select`, то ее значение – временное дерево (или фрагмент конечного дерева, как он называется в XSLT 1.0). Во многих случаях это дерево содержит только единственный текстовый узел, тогда оно ведет себя подобно строке.

Локальная переменная часто полезна для получения значения по умолчанию атрибута. Например:

```
<xsl:variable name="ширина">
   <xsl:choose>
      <xsl:when test="@ширина">
         <xsl:value-of select="@ширина"/>
      </xsl:when>
      <xsl:otherwise>0</xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

Впоследствии переменная `$ширина` может использоваться при вычислениях вместо атрибута `@ширина`, и тогда не придется беспокоиться о том, что атрибут мог быть случайно опущен. То, что переменная формально является деревом, а не строковыми данными, не отражается на возможности ее использования.

Новички часто пишут это следующим образом:

```
<xsl:choose>
   <xsl:when test="@ширина">
      <xsl:variable name="ширина"
                    select="@ширина"/>
   </xsl:when>
   <xsl:otherwise>
      <xsl:variable name="ширина" select="0"/>
   </xsl:otherwise>
</xsl:choose>
```

Это не будет работать, потому что область действия обоих объявлений переменных заканчивается после закрывающего тега элемента `<xsl:choose>`!

Для тех, кому краткость важнее, чем ясность, есть альтернативный способ инициализировать переменную:

```
<xsl:variable name="ширина"
select="concat(@ширина, substring('0', 1, not(@ширина)))"/>
```

Как это работает, описано в разделе о функции `substring()` в главе 7.

Переменные, значением которых является дерево, также необходимы, когда требуется использовать <xsl:call-template>, чтобы вычислить значение, с которым можно будет оперировать далее. Это показано в следующем примере.

Пример: Сохранение результата <xsl:call-template> в переменной

Исходный файл

Исходный файл – список опер и композиторов, использованный в предыдущем примере, опера.xml.

Таблица стилей

Таблица стилей – файл композиторы.xsl.

Эта таблица стилей использует универсальный именованный шаблон («создать-список»), чтобы вывести список имен в формате «A; B; C и D». Таблица стилей передает этому шаблону набор узлов, содержащий имена всех композиторов в исходном документе. Результат применения шаблона «создать-список» она передает в переменную (которой является временное дерево, содержащее единственный текстовый узел), а переменную передает в функцию translate() (описанную в главе 7), чтобы преобразовать запятые в точку с запятой. Функция translate() преобразовывает ее первый параметр в строку; в данном случае результатом является значение текстового узла дерева.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="/">
    <xsl:variable name="список">
         <xsl:call-template name="создать-список">
              <xsl:with-param name="имена"
                        select="/программа/композитор/полное-имя"/>
         </xsl:call-template>
    </xsl:variable>
    На этой неделе исполняются произведения следующих композиторов:
    <xsl:value-of select="translate($список, ',', ';')"/>
</xsl:template>

<xsl:template name="создать-список">
    <xsl:param name="имена"/>
    <xsl:for-each select="$имена">
        <xsl:value-of select="."/>
        <xsl:if test="position()!=last()">, </xsl:if>
        <xsl:if test="position()=last()-1"> и </xsl:if>
    </xsl:for-each>
</xsl:template>
```

Вывод

```
На этой неделе исполняются произведения следующих композиторов: Wolfgang Amadeus
Mozart; Guiseppe Verdi; и Giacomo Puccini
```

В XSLT 1.1 все, что можно выполнить в главном исходном документе или в документе, загруженном с помощью функции `document()`, стало возможным и для временного дерева. Это значительно увеличивает полноценность временных деревьев. Ниже приведены возможные их применения:

- Выполнять преобразования в нескольких фазах. Вывод первой фазы направляется во временное дерево, а вторая фаза использует это временное дерево как свой ввод. Это позволяет разбить сложное преобразование на ряд простых шагов. Например, если требуется произвести сортировку и группирование, а затем пронумеровать отсортированные группы, можно организовать это как трехфазное преобразование. Сначала выполняется группирование, затем сортировка, и наконец производится нумерование. Для выполнения многофазных преобразований можно использовать такую структуру таблицы стилей:

```
<xsl:template match="/">
  <xsl:variable name="вывод-первой-фазы">
      <xsl:apply-templates select="." mode="фаза1"/>
  </xsl:variable>
  <xsl:variable name="вывод-второй-фазы">
      <xsl:apply-templates select="$вывод-первой-фазы" mode="фаза2"/>
  </xsl:variable>
  <xsl:apply-templates select="$вывод-второй-фазы" mode="фаза3"/>
</xsl:template>
```

- Полезно не только использовать отдельный режим для каждой фазы преобразования, но и сохранять шаблонные правила для каждой фазы в отдельном модуле таблицы стилей, связанном с главным модулем таблицы стилей с помощью `<xsl:include>`.
- Использовать временное дерево для хранения готовых поисковых таблиц. Пример этого показан в разделе по функции `document()` главы 7. В XSLT 1.0 такие данные можно хранить в пределах таблицы стилей в определяемом пользователем элементе верхнего уровня, обращаясь к нему с помощью «`document('')`», что делает таблицу стилей доступной в качестве вторичного исходного документа. В XSLT 1.1 вместо использования определяемого пользователем элемента верхнего уровня, такие данные могут храниться в глобальной переменной дерева, и к ним можно обращаться прямо.
- Использовать временное дерево для сохранения сложных рабочих структур данных, которые могут передаваться именованному шаблону как параметры для обработки. Например, в главе 10 один из демонстрационных примеров вычисляет маршрут коня по шахматной доске, и для этого требуется именованный шаблон, который берет в качестве входного параметра текущее состояние доски. В XSLT 1.0 (без привлечения расшире-

ний от поставщика) необходимо кодировать эту информацию в виде строки. В XSLT 1.1 на текущий момент это можно представить как древовидную структуру, гораздо лучше поддающуюся обработке. (Но имейте в виду, это не изменяет основное правило, что переменные нельзя модифицировать: временное дерево остается таким, каким его сформировали. Единственное, что можно делать – копировать его в новое дерево, производя во время этого нужные изменения.)

- Формировать временное дерево, представляющее список пунктов, чтобы можно было использовать <xsl:for-each> для итерации по этому списку. Например, если нужно произвести выходной файл, перечисляющий все цвета радуги, можно определить переменную следующим образом:

```
<xsl:variable name="радуга">
  <цвет>красный</цвет>
  <цвет>оранжевый</цвет>
  <цвет>желтый</цвет>
  <цвет>зеленый</цвет>
  <цвет>голубой</цвет>
  <цвет>синий</цвет>
  <цвет>фиолетовый</цвет>
</xsl:variable>
```

а затем выполнять итерации по ней:

```
<xsl:for-each select="$радуга/цвет">
   <xsl:document href="{.}.xml">
     . . .
   </xsl:document>
</xsl:for-each>
```

Точно так же можно создать, например, список всех цветов выпускаемых изделий.

См. также

<xsl:param> на стр. 316

xsl:when

Элемент <xsl:when> всегда является непосредственным потомком элемента <xsl:choose>. Он определяет условие, которое должно быть проверено, и действие, которое нужно произвести, если условие выполняется, то есть истинно.

Определен в

XSLT, раздел 9.2

Формат

```
<xsl:when test=Выражение>
   тело шаблона
</xsl:when>
```

Расположение

<xsl:when> всегда является непосредственным потомком элемента <xsl:choose>, причем у <xsl:choose> обязательно должен иметься, по крайней мере, один дочерний элемент <xsl:when>.

Атрибуты

Имя	Значение	Назначение
test обязательный	выражение	Логическое условие, которое должно быть проверено

Содержимое

Тело шаблона.

Действие

Элемент <xsl:choose> применяется следующим образом:

- Выбирается первый элемент <xsl:when>, выражение test которого истинно; последующие элементы <xsl:when> игнорируются, независимо от значения их выражений.
- Если ни один из элементов <xsl:when> не имеет выражения test, которое оказывается истинным, выбирается инструкция <xsl:otherwise>. Если таких инструкций нет, не выбирается никакой элемент.
- Выбранный дочерний элемент (если он выбран) выполняется путем применения его тела шаблона к текущему контексту: это равносильно тому, как если бы его тело шаблона находилось на месте инструкции <xsl:choose>.

Точно не определено, вычисляется ли выражение test в элементах <xsl:when> после уже выбранного или нет, поэтому, если оно вызывает функции, имеющие побочные эффекты, или если оно содержит ошибки, результат непредсказуем.

Любое значение XPath-выражения может быть преобразовано в логический тип данных. Вкратце, правила следующие:

- Если выражение – набор узлов, оно считается истинным, когда набор узлов не пустой
- Если выражение – строковое или является фрагментом конечного дерева, оно считается истинным, когда строковое значение не пустое

- Если выражение – численное, оно считается истинным, когда число – не нуль

Использование и примеры

См. <xsl:choose> на стр. 213

См. также

<xsl:choose> на стр. 213

<xsl:otherwise> на стр. 302

<xsl:if> на стр. 253

xsl:with-param

Элемент <xsl:with-param> используется для установки значений параметров при вызове шаблона или при использовании <xsl:call-template>, или при использовании <xsl:apply-templates>. В XSLT 1.1 он также может использоваться с <xsl:apply-imports>.

Определен в

XSLT, раздел 11.6

Формат

```
<xsl:with-param name=ПолноеИмя select=Выражение >
   тело шаблона
</xsl:with-param>
```

Расположение

<xsl:with-param> – всегда непосредственный потомок элемента <xsl:apply-templates> или <xsl:call-template>, или (только в XSLT 1.1) <xsl:apply-imports>.

Атрибуты

Имя	Значение	Назначение
name обязательный	Полное имя	Имя параметра.
select необязательный	выражение	Значение параметра, которое будет установлено в вызываемом шаблоне.

Конструкции полное имя и выражение определены в главе 5.

Содержимое

Тело шаблона (необязательно). Если присутствует атрибут select, элемент <xsl:with-param> должен быть пустым.

Действие

Элемент <xsl:with-param> может встречаться только как непосредственный потомок элементов <xsl:call-template>, <xsl:apply-templates> или (только в XSLT 1.1) инструкции <xsl:apply-imports>.

Элемент <xsl:with-param> задает значения параметрам. Значение параметра может использоваться в пределах вызываемого шаблона.

Значение параметра устанавливается точно так же, как в случае элемента <xsl:variable>. То есть значение берется из выражения select, если оно есть, или путем применения тела шаблона, если атрибут select отсутствует.

Если в вызываемом шаблоне есть элемент <xsl:param>, имя которого соответствует имени элемента <xsl:with-param>, то значение, установленное для элемента <xsl:with-param>, доступно в пределах шаблона. Если вызываемый шаблон не имеет такого параметра, значение параметра игнорируется, причем это не рассматривается как ошибка. В случае элемента <xsl:apply-templates> значение параметра доступно для каждого из вызываемых шаблонов (по одному шаблону на каждый выбранный узел). Параметр оценивается только один раз, он имеет одно и то же значение для каждого из этих шаблонов.

Имя параметра определяет полное имя. Обычно это – простое имя, например «город» или «сумма-продаж», но также это может быть имя, уточненное префиксом, например «мое:значение». Если имя имеет префикс, этот префикс должен соответствовать пространству имен, которое действует в этом месте таблицы стилей. Истинное имя параметра, для сопоставления его с элементом <xsl:param> в вызываемом шаблоне, определяется не префиксом, а URI пространства имен, который соответствует данному префиксу. Таким образом, имени «мое:значение» будет соответствовать параметр, объявленный как «ваше:значение», если префиксы «мое» и «ваше» относятся к одному и тому же URI пространства имен. Если имя не имеет никакого префикса, оно имеет пустой URI пространства имен, так как оно не использует URI пространства имен по умолчанию.

Считается ошибкой, если два равноуровневых элемента <xsl:with-param> имеют одинаковое имя, после того как их префиксы пространств имен заменены на URI пространств имен. Другими словами, нельзя задавать два значения для одного и того же параметра.

Элемент <xsl:with-param> фактически не объявляет переменную, поэтому допустимо, если его имя идентично имени переменной, в области действия которой находится элемент. Вполне можно задавать параметр следующим образом:

```
<xsl:with-param name="текущий-пользователь" select="$текущий-пользователь"/>
```

Это гарантирует, что переменная "$текущий-пользователь" в вызываемом шаблоне имеет такое же значение, как переменная "$текущий-пользователь" в вызывающем шаблоне.

Использование и примеры

Параметры для шаблонов имеют существенное значение в XSLT, потому что переменные не могут быть модифицированы. Это означает, что решение многих задач, которое в традиционных языках программирования достигается путем изменения значения переменных в цикле, в XSLT требует вместо этого использование рекурсивных вызовов и параметров. Подробнее это обсуждается в главе 9, а в главе 10 приведены примеры некоторых детальных методик.

Примеры рекурсивных вызовов есть также в данной главе в разделе <xsl:call-template> на стр. 204.

См. также

<xsl:apply-imports> на стр. 174

<xsl:apply-templates> на стр. 178

<xsl:call-template> на стр. 204

<xsl:param> на стр. 316

Резюме

Это была длинная глава, но автор надеется, читатели согласятся, что в ней нет лишних страниц! Здесь подробно обсуждены все элементы XSLT и приведены действующие примеры, помогающие лучше понять возможности использования каждого элемента. Далее таким же образом будут рассмотрены *выражения*.

5

Выражения

В этой главе определяются синтаксис и значение **выражений XPath**. Выражения XPath используются во многих местах таблиц стилей XSLT для выборки и манипуляции данными исходного документа. С их помощью формируется выходной документ. XPath определен консорциумом W3C как самостоятельный язык и поэтому может использоваться и в других контекстах, например, в определении ссылок из одного XML-документа в другой (см. спецификации W3C XLink и XPointer); однако все наши примеры будут посвящены способу использования его в XSLT.

Так же как и в других языках программирования, синтаксис определяется набором **порождающих правил**. Каждое правило определяет структуру конкретной конструкции как набор альтернатив, последовательностей или повторений. В данной главе каждому порождающему правилу соответствует отдельный раздел.

Официальные порождающие правила из спецификации XPath, находящейся по адресу *http://www.w3.org/TR/xpath*, здесь скопированы практически без изменений. Они только переупорядочены в алфавитном порядке для упрощения поиска и в них сделаны незначительные поправки в оформлении для упрощения чтения. Также здесь скопированы использующиеся в синтаксисе XPath правила из стандартов XML и XML Namespaces. В них вставлены ссылки, позволяющие, если это необходимо, найти соответствующее правило в исходной спецификации. Однако вся необходимая информация оттуда уже размещена здесь, так что это может понадобиться только в том случае, если необходимо увидеть точную формулировку стандарта.

В этой главе находится большая часть информации, необходимой для написания выражений XPath. Ключевые концепции были объяснены во второй главе, в частности древовидная модель и понятие контекста выражения, – обе концепции совершенно необходимы для полного понимания данной гла-

вы. Во второй главе также была объяснена система типов данных. В выражения XPath также можно включать вызовы функций: стандартные функции, определенные в рекомендациях XPath и XSLT, описаны в главе 7 этой книги, а способ добавить собственные – в главе 8.

Текущая версия спецификации XPath во время написания этой книги все еще 1.0, хотя уже опубликован Рабочий проект спецификации XSLT 1.1, он не требует изменений в XPath. Впрочем, автор включил в книгу самый последний список ошибок в XPath 1.0 (хотя обычный пользователь никогда не столкнется с большинством этих ошибок и упущений).

Система обозначений

Порождающие правила XPath неявно определяют приоритет операторов. Например, правило ВыражениеИЛИ определяет его как последовательность операндов ВыражениеИ, разделенных операторами «or». Это удобный способ определения того, что у оператора «and» приоритет выше, чем у оператора «or».

Одно из последствий использования такого стиля определений – то, что простейшее выражение ВыражениеИЛИ состоит из одного выражения ВыражениеИ без единого оператора «or». Это создает проблему, так как когда мы говорим о выражении, не использующем оператор «or», мы не можем назвать его ВыражениеИЛИ. Так что мы будем называть выражение ВыражениеИЛИ без оператора «or» **тривиальным** ВыражениемИЛИ, а настоящее выражение (с одним или более операторами «or») **нетривиальным** ВыражениемИЛИ. В разделе, описывающем действие ВыраженияИЛИ, мы всегда будем рассматривать нетривиальный вариант. Эта же ситуация возникает и со многими другими конструкциями.

Хотя порождающие правила и определяют приоритет операторов, они не обеспечивают проверку типов. Этого было бы сложно достигнуть, потому что переменные не типизированы, к тому же в большинстве контекстов значение одного типа может быть преобразовано в значение требуемого типа, так что, например, «3 or 'хлеб'» вполне допустимое выражение (равное true). Есть контексты, требующие значение, являющееся набором узлов, а значения других типов не могут быть преобразованы в наборы узлов; тем не менее, проектировщики языка решили не указывать это ни в одном порождающем правиле. Это означает, что выражение «3|'хлеб'» (где «|» – оператор объединения) также синтаксически верно в соответствии с порождающими правилами, хотя на самом деле это очевидная ошибка, потому что нарушено правило, по которому оба операнда «|» должны быть наборами узлов. Подумайте об аналогии с русским языком – можно придумать грамматически корректное, но бессмысленное предложение: «Удобное яблоко вчера лишь ревело».

С чего начать

Некоторые предпочитают представлять синтаксис языка вверх ногами, начиная с самых простых конструкций, таких как числа и имена, а другие

предпочитают начинать с самого верхнего уровня – с таких конструкций, как Программа или Выражение. Поскольку ни один порядок представления не удовлетворит всех читателей и поскольку они, скорее всего, будут переходить по ссылкам между главами, мы решили перечислять порождающие правила в алфавитном порядке. Если вы относитесь к людям, любящим непредсказуемость произвольного листания энциклопедии, вы можете наслаждаться последовательным прочтением разделов или с тем же успехом начать читать с конца, продвигаясь к началу. При подходе «сверху вниз» начинайте с правила Выражение на стр. 394, так как это порождающее правило самого высокого уровня для выражения XPath. При подходе «снизу вверх» взгляните на синтаксическое дерево, показанное на следующей странице, и выберите элементы, находящиеся дальше всего от корня дерева. Или, если вы хотите начать с середины, взгляните на правило Шаг (стр. 465), являющееся одной из ключевых концепций, придающих XPath его силу.

Многие языки различают лексические правила, определяющие формат основных лексем, таких как имена, числа и операторы, и синтаксические правила, определяющие, как эти лексемы объединяются для образования выражений и других конструкций высокого уровня.

В спецификации XPath есть и синтаксические и лексические порождающие правила, но они разделены не столь четко, как в других языках. Так как некоторые конструкции фигурируют в обоих видах правил, я оставил их вместе, приводя лексические правила в алфавитном порядке, так же как и синтаксические, но выделяя их в тексте. Основное отличие между этими двумя видами правил в том, что пробельные символы могут свободно использоваться между лексемами, но не внутри лексем. Лексическое правило самого верхнего уровня – ЭлементВыражения.

Синтаксическое дерево

Чтобы помочь вам найти свою дорогу в синтаксисе, автор составил нижеприведенные иерархические схемы, показывающие все синтаксические правила (но не исключительно лексические правила). Непосредственные потомки каждой синтаксической конструкции – это конструкции, на которые она ссылается в своем синтаксическом правиле.

На первой схеме находятся все выражения вплоть до ВыраженияПути. Это очень простая часть иерархии – каждая конструкция, кроме ОператорУмножения, определяет тип выражения, используя операнды и операторы, а сама иерархия отражает приоритет операторов. Например, эта схема демонстрирует, что ВыражениеИ может содержать ВыражениеРавенства, из чего следует, что операторы, используемые в ВыраженииРавенства (такие как «=»), имеют более высокий приоритет, чем оператор «and», используемый в ВыраженииИ.

```
Выражение
    ВыражениеИЛИ
        ВыражениеИ
            ВыражениеРавенства
```

```
            ВыражениеОтношения
               АддитивноеВыражение
                  МультипликативноеВыражение
                     ОператорУмножения
                     УнарноеВыражение
                        ВыражениеОбъединения
                           ВыражениеПути
```

На следующей схеме раскрывается ВыражениеПути**. Звездочка, следующая после имени, означает, что оно раскрывается в иерархии.**

```
ВыражениеПути
   МаршрутПоиска
      ОтносительныйМаршрутПоиска *
      АбсолютныйМаршрутПоиска
         ОтносительныйМаршрутПоиска *
         СокращенныйАбсолютныйМаршрутПоиска
            ОтносительныйМаршрутПоиска *
   ФильтрующееВыражение
      ПервичноеВыражение
         ССылкаНаПеременную
         Выражение *
         Литерал
         Число
            Цифры
         ВызовФункции
            ИмяФункции
               ПолноеИмя
            Аргумент
               Выражение *
      Предикат
         ПредикативноеВыражение
            Выражение *
   ОтносительныйМаршрутПоиска
      Шаг
         СпецификаторОси
            ИмяОси
            СокращенныйСпецификаторОси
         КритерийУзла
            КритерийИмени
               ИмяБезДвоеточия
               ПолноеИмя
            ТипУзла
            Литерал *
         Предикат *
         СокращенныйШаг
      СокращенныйОтносительныйМаршрутПоиска
      ОтносительныйМаршрутПоиска *
```

Заметьте, что как только вы сталкиваетесь с ВыражениемПути**, синтаксис становится намного более сложным. Пожалуй, это неудивительно для языка, на-**

зываемого XPath, поскольку основное назначение этого языка – определение путей в XML-документе, а это именно то, чем занимается ВыражениеПути. Ниже этого уровня синтаксис становится запутанным из-за того, что он полон сокращений, упрощающих написание часто используемых выражений пути, но с трудом определяемых формально. Подробности находятся в разделе ВыражениеПути на стр. 407.

В следующих разделах в алфавитном порядке описываются синтаксические конструкции.

АбсолютныйМаршрутПоиска (AbsoluteLocationPath)

АбсолютныйМаршрутПоиска – это маршрут поиска, начинающийся с корневого узла.

Выражение	Синтаксис
АбсолютныйМаршрутПоиска	«/» ОтносительныйМаршрутПоиска ? | СокращенныйАбсолютныйМаршрутПоиска

Определено в

XPath, раздел 2, правило 2

Используется в

МаршрутПоиска

Справка по использованию

Простейший экземпляр АбсолютногоМаршрутаПоиска – «/» – выбирает корневой узел. Точнее, при использовании нескольких входных документов, он выбирает корневой узел документа, содержащего контекстный узел.

Такой синтаксис знаком каждому, кто работал с файловой системой UNIX, хотя на самом деле он не очень логичен. Символ «/» является как выражением для ссылки на корневой узел, так и оператором для разделения частей пути. Я представляю себе экземпляр АбсолютногоМаршрутаПоиска «/X» как сокращение для некоего воображаемого выражения «®/X», а «/» как сокращение для «®», где «®» – корневой узел документа.

Конструкция СокращенныйАбсолютныйМаршрутПоиска, принимающая форму «//X», обсуждается в отведенном ей разделе на стр. 390.

Если вы пишете таблицу стилей, загружающую несколько исходных документов, используя функцию `document()`, то не существует способа выбрать корень основного исходного документа, когда контекстный узел находится в другом документе. Для решения этой проблемы можно включить в табли-

цу стилей объявление глобальной переменной вида <xsl:variable name="корень" select ="/"/>. Затем вы сможете в любой момент обратиться к корню основного документа с помощью выражения «$корень».

Примеры

Выражение	Описание
/	Выбирает корневой узел документа, содержащего контекстный узел.
/price-list	Выбирает элемент документа при условии, что он называется <price-list>.
/*	Выбирает элемент документа, независимо от его типа.
/child::node()	Выбирает все узлы, являющиеся непосредственными дочерними узлами корня документа, то есть элемент документа и любые комментарии или инструкции обработки, следующие в документе до или после него. (Заметьте, однако, что строка <?xml version="1.0"?> в начале документа **не** является инструкцией обработки; на самом деле, она вообще не является узлом и к ней невозможно получить доступ средствами XPath.)
/*/xsl:*	Выбирает все узлы элементов с именами из пространства имен, связанного с префиксом «xsl:», являющиеся непосредственными дочерними узлами элемента документа. (Если этот пример применить к таблице стилей XSLT, то будут выбраны все элементы XSLT самого верхнего уровня.)
//рисунок	Этот экземпляр СокращенногоАбсолютногоМаршрутаПоиска выбирает в документе все элементы <рисунок>.

АддитивноеВыражение (AdditiveExpr)

Нетривиальные АддитивныеВыражения используются для сложения или вычитания числовых значений.

Выражение	Синтаксис
АддитивноеВыражение	МультипликативноеВыражение | АддитивноеВыражение «+» МультипликативноеВыражение | АддитивноеВыражение «-» МультипликативноеВыражение

Определено в

XPath, раздел 3.5, правило 25

Используется в

ВыражениеОтношения

Справка по использованию

Нетривиальное АддитивноеВыражение состоит из одного или более операндов МультипликативноеВыражение, разделенных операторами сложения «+» или вычитания «-». Тривиальное выражение АддитивноеВыражение состоит из единственного выражения МультипликативноеВыражение без оператора сложения или вычитания.

При использовании оператора вычитания позаботьтесь о том, чтобы не спутать его с дефисом в имени. Если он следует непосредственно за именем, используйте пробел, чтобы разделить их. Запомните, что «цена-скидка» (без пробелов) – это имя, в то время как выражение «цена - скидка» (с пробелами) – осуществляет вычитание.

В XSLT всегда используются числа с плавающей точкой двойной точности, так что вычисления выполняются с использованием арифметики с плавающей точкой, в соответствии с правилами IEEE 754. См. описание числового типа данных в соответствующем разделе главы 2.

Если операнд АддитивногоВыражения не является числом, то он преобразуется в число, как если бы для этого была использована функция number(). Если же значение не может быть преобразовано в обычное число, то оно преобразуется в специальное значение NaN (не-число) и в этом случае результатом сложения или вычитания также будет NaN.

Примеры в контексте

Выражение	Описание
`$X + 1`	Результат добавления единицы к значению переменной $X.
`last()-1`	Значение, на единицу меньшее, чем положение последнего узла в контекстном списке.
`@цена - @скидка`	Отнять от преобразованного в число значения атрибута контекстного узла цена преобразованное в число значение атрибута контекстного узла скидка.
`count($список) mod 5 + 1`	Остаток от деления числа узлов в наборе узлов на 5, плюс единица. Результат будет в диапазоне от 1 до 5. (За дополнительной информацией об операторе получения остатка от деления обращайтесь к стр. 428.)
`42`	Число 42 является тривиальным АддитивнымВыражением и поэтому может быть использовано везде, где допускается появление АддитивногоВыражения.

Аргумент (Argument)

Конструкция Аргумент используется для представления значения, передаваемого конструкции ВызовФункции.

Выражение	Синтаксис
Аргумент	Выражение

Определено в

XPath, раздел 3.2, правило 17

Используется в

ВызовФункции

Справка по использованию

В качестве аргумента функции может использоваться любое выражение XPath. Для конструкции Аргумент порождающее правило определено исключительно ради удобства чтения.

Имейте в виду, что использование некоторых выражений в качестве аргументов может выглядеть довольно странно с точки зрения обычных языков программирования. Например, выражение «string-length(..)» возвращает длину строкового значения родителя контекстного узла, а выражение «document(@*,/)» возвращает набор узлов, содержащий корневые узлы документов, URI которых находятся в атрибутах контекстного узла, используя базовый URI корня исходного документа для разрешения относительных URI.

Спецификация XPath использует термин Arguments (аргументы) для входных данных выражения ВызовФункции. Термин **параметры** (parameters) имеет другое значение – он используется в XSLT для значений, передаваемых шаблону.

Для внешних функций, например, написанных на Java или JavaScript, может возникнуть необходимость преобразовать аргументы из типов XPath в типы JavaScript. Правила таких преобразований (определенные в Рабочем проекте спецификации XSLT 1.1) даны в главе 8.

Большинство функций автоматически преобразуют передаваемые им аргументы в нужный тип данных, например, функция concat() автоматически преобразует каждый свой аргумент в строку. Это не относится к тому случаю, когда требуемый тип аргумента – набор узлов (например, у функций count() и sum()), потому что значения никаких других типов не могут быть преобразованы в наборы узлов. Некоторые функции, такие как id() и key(), полиморфны: их поведение меняется в зависимости от типа переданного аргумента. В этом случае вам может понадобиться совершить явное преобразование типа (используя одну из функций: string(), number() или boolean()) для достижения желаемого результата. Все эти функции описаны в главе 7.

Примеры в контексте

В качестве аргумента функции может использоваться любое выражение XPath. Вот несколько примеров, демонстрирующих эту гибкость:

Выражение	Описание
`count(@*)`	Возвращает число атрибутов контекстного узла.
`id(string(@idref))`	Возвращает набор узлов, содержащий элемент (если такой есть) с ID равным значению атрибута `idref` контекстного узла.
`not(isbn)`	Возвращает истину, если у контекстного узла нет дочернего узла элемента с именем `<isbn>`. В этом примере аргументом является `ОтносительныйМаршрутПоиска`, эквивалентный выражению `«./child::isbn»`.
`generate-id(/)`	Возвращает строку с уникальным идентификатором корневого узла документа, содержащего контекстный узел.

ВызовФункции (FunctionCall)

`ВызовФункции` вызывает либо системную функцию, либо функцию расширения, определенную пользователем или поставщиком ПО.

Выражение	Синтаксис
`ВызовФункции`	`ИмяФункции «(» ( Аргумент («,» Аргумент )* )? «)»`

Определено в

XPath, раздел 3.2, правило 16

Используется в

`ПервичноеВыражение`

Справка по использованию

Существует два вида вызовов функций: вызовы стандартных функций, перечисленных в главе 7, которые должны присутствовать в любом XSLT-продукте, соответствующем стандарту, и вызовы функций расширения, которые могут подключаться поставщиком продукта или пользователем, как это описано в главе 8. Оба вида легко отличить друг от друга, потому что при вызове стандартной функции имя функции всегда будет простым именем, без двоеточия, а имя функции расширения всегда будет находиться в каком-либо пространстве имен, отличном от пространства имен «по умолчанию», и узнается по виду `«префикс:локальное-имя»`. Префикс должен ссылаться на объявление пространства имен, находящееся в области действия выражения, содержащего вызов функции.

С технической точки зрения существует два вида стандартных функций: основные функции, определенные непосредственно в спецификации XPath, и

дополнительные функции, определенные в спецификации XSLT. При использовании XPath только в таблицах стилей XSLT это деление на два вида не имеет значения.

Вызов функции без аргументов имеет вид `ИмяФункции «(» «)»`, а если при вызове передаются аргументы, то они, как и следует ожидать, разделяются запятыми. Аргументы могут быть любым выражением XPath, подвергаемым всем правилам проверки типов или другим ограничениям, определенным для конкретной рассматриваемой функции. Сюда входят такие выражения, как «/», «.» и «@*», которым нельзя найти соответствие в традиционных языках программирования.

Синтаксис вызова функции похож на синтаксис критерия узла, такого как «`node()`» или «`comment()`», но легко различается по использованию набора зарезервированных имен для различных типов узлов.

При использовании стандартных функций передаваемые значения всегда приводятся к требуемому типу, если это необходимо. Многие стандартные функции принимают фиксированное число аргументов. У некоторых функций есть необязательный аргумент; как правило, его значением по умолчанию является набор узлов, состоящий из контекстного узла. Функция `concat()`, сцепляющая строки, отличается тем, что число ее аргументов не ограничено.

Вызов функции всегда возвращает значение одного из четырех типов данных XPath: логического, числового, строкового или набор узлов. Начиная с версии спецификации XSLT 1.0, функция теоретически может возвращать фрагмент конечного дерева, но ни одна из стандартных функций этого не делает. Возвращаемым значением функции расширения может также быть *внешний объект*, описываемый далее. Нигде не сказано, что одна и та же функция всегда будет возвращать значение одного и того же типа: например, функция `system-property()` иногда возвращает число, а иногда – строку.

Рабочий проект стандарта XSLT 1.1 определяет дополнительный тип данных для использования в функциях расширения: внешний объект. Это дескриптор или обертка для объекта, созданного в другом языке, таком как Java или JavaScript. Внешние объекты могут возвращаться одним вызовом функции расширения и передаваться в качестве аргумента другому, у них нет какого-либо другого применения в таблицах стилей; их нельзя даже преобразовать в строку для последующего вывода, если, конечно, для этого нет другой функции расширения.

Ни у одной стандартной функции нет побочных эффектов; они не изменяют значения переменных, не выводят ничего на экран или в файл и не изменяют настройки броузера или операционной системы. Однако ничто не препятствует созданию функции расширения с побочными эффектами. Например, функция расширения может выводить на экран сообщение или увеличивать счетчик, она может даже изменять исходный документ или саму таблицу стилей. Однако функции расширения с побочными эффектами будут действовать довольно непредсказуемо, так как не существует определенного по-

рядка, в котором они выполняются. Например, нельзя сказать, что глобальные переменные вычисляются в каком-то определенном порядке или что они вычисляются только один раз; а глобальная переменная, к которой нет ни одного обращения, может вообще не вычисляться.

Вызов недоступной функции является ошибкой. Однако если в таблице стилей есть выражение, ссылающееся на недоступную функцию, – это не обязательно ошибка. Доступность конкретной функции можно проверить при помощи системной функции `function-available()` и не вызывать недоступные функции. С помощью `function-available()` можно определить доступность как функций расширения, так и системных функций: это полезная возможность, так как, если в будущих версиях стандартов XSLT и XPath появятся новые функции, может потребоваться разрабатывать таблицы стилей, совместимые с продуктами, реализующими только предыдущую версию.

Примеры

Выражение	Описание
`true()`	Вызов стандартной функции, всегда возвращающей логическое значение истина.
`string-length($x)`	Вызов стандартной функции, преобразующей свой аргумент в строку и возвращающей число символов в получившейся строке.
`count(*)`	Вызов стандартной функции, вычисляющей значение набора узлов «`*`» (набор всех дочерних элементов контекстного узла) и возвращающей число узлов, находящихся в этом наборе узлов.
`xt:intersection($x, $y)`	Вызов функции расширения. Префикс «`xt:`», который должен соответствовать объявлению пространства имен, находящемуся в текущей области действия, указывает на то, что это функция расширения. Правила нахождения реализации этой функции расширения определяются ее разработчиком.
`function-available ('xt:intersection')`	Вызов стандартной функции `function-available()`, проверяющей доступность функции расширения «`xt:intersection`». Заметьте, что положительный ответ не означает, что вызов этой функции будет успешно выполнен; например, не существует способа узнать число аргументов функции.

Выражение (Expr)

Выражение – это порождающее правило самого высокого уровня, представляющее выражение XPath.

Выражение	Синтаксис
Выражение	ВыражениеИЛИ

Определено в

XPath, раздел 3.1, правило 14

Используется в

Выражение — правило самого высокого уровня для синтаксиса выражения XPath.

Вложенные Выражения используются также в порождающих правилах Аргумент, ПервичноеВыражение и ПредикативноеВыражение.

В таблице стилей XSLT выражения XPath используются в следующих контекстах:

Контекст	Получающийся тип данных
Шаблоны значений атрибутов. Выражение записывается между фигурными скобками «{» и «}» в любом атрибуте, где допускается использование шаблонов значений атрибутов. Список этих атрибутов приведен в разделе «Шаблоны значений атрибутов» главы 3.	строковый
`<xsl:apply-templates select="">`	набор узлов
`<xsl:copy-of select="">`	набор узлов
`<xsl:for-each select="">`	набор узлов
`<xsl:if test="">`	логический
`<xsl:key use="">`	набор узлов или строка
`<xsl:number value="">`	число
`<xsl:param select="">`	любой
`<xsl:sort select="">`	строковый
`<xsl:value-of select="">`	строковый
`<xsl:variable select="">`	любой
`<xsl:when test="">`	логический
`<xsl:with-param select="">`	любой

Справка по использованию

Выражение — это конструкция самого высокого уровня для порождающих правил, определенных в данной главе. Синтаксическое правило говорит, что каждое Выражение является ВыражениемИЛИ (и наоборот, каждое ВыражениеИЛИ является Выражением). Так что оно просто говорит о том, что правила те же самые, что и для ВыраженияИЛИ, конструкция OrExpr, описанного далее в этой главе.

Архитекторы XPath решили задать синтаксис языка таким образом, чтобы приоритет различных операторов был заложен в порождающие правила. Так

что это правило говорит нам, что самый низкий приоритет у оператора «or». Порождающее правило для ВыраженияИЛИ определено с использованием ВыраженияИ, а это говорит о том, что у оператора «and» более высокий приоритет, чем у «or», и так далее. Одним из последствий такого стиля определений является то, что выражениеИЛИ не обязательно содержит оператор «or», это просто выражение, появляющееся в контексте, где будет воспринят оператор «or».

В XSLT выражения (то есть предложения, соответствующие порождающему правилу для конструкции Выражение) находятся внутри таблиц стилей, а контекст таблицы стилей накладывает не заложенные в синтаксис XPath ограничения на Выражения. Это и правила типов данных (например, в некоторых контекстах таблицы стилей требуется набор узлов, как показано в таблице выше), и правила об области действия переменных и назначении префиксов пространств имен. Например, «2+2» является вполне допустимым выражением XPath, однако если вы попытаетесь написать <xsl:for-each select="2+2"/>, XSLT-процессор выведет сообщение об ошибке, говорящее, что в данном контексте требуется набор узлов.

Контекст таблицы стилей накладывает ограничения на синтаксическую допустимость выражения, а также, когда приходит время вычислять выражение, предоставляет контекст времени выполнения. Таким образом, мы можем рассматривать контекст выражения как состоящий из двух частей: статической и динамической части.

Статический контекст выражения, который можно определить путем рассмотрения таблицы стилей, включает в себя:

- Список имен переменных, находящихся в текущей области действия в момент появления выражения.
- Список префиксов пространств имен, находящихся в текущей области действия в момент появления выражения, а также URI пространства имен для каждого префикса. Это влияет на действительность и, на самом деле, на значение любого ПолногоИмени (грубо говоря, имени, уточненного пространством имен), встречающегося в выражении, как описано в разделе «ПолноеИмя» этой главы.
- Требуемый тип данных (например, инструкции <xsl:if> нужен логический тип данных, инструкции <xsl:for-each> – набор узлов, а выражение, находящееся в шаблоне значения атрибута, должно возвращать строку).
- Набор функций расширения (внешние функции, определенные пользователем или поставщиком ПО), доступные в текущем контексте таблицы стилей.
- URI таблицы стилей, в которой находится выражение, или, точнее, базовый URI элемента, в котором находится выражение. Обычно это URL сущности XML, содержащей данную часть таблицы стилей (которая может использовать ссылки на внешние общие сущности или состоять из нескольких отдельных сущностей, соединенных директивами <xsl:include> и <xsl:import>). В XSLT 1.1 также разрешается изменять базовый URI при помощи атрибута xml:base, как это описано в главе 2. Фактически этот ба-

зовый URI нужен лишь для одной цели – разрешения относительных URI, используемых в функции document(), которая описана в главе 7.

Динамический контекст выражения, который может быть определен только лишь в процессе использования таблицы стилей для обработки определенного исходного документа, включает в себя:

- Текущие значения всех переменных, на которые ссылается выражение.
- Контекстный узел, контекстная позиция и размер контекста, вместе задающие множество узлов исходного документа, обрабатываемых в данный момент таблицей стилей. Эти важные концепции описаны в главе 2.

XSLT-процессору во время выполнения также может понадобиться информация о статическом контексте. Например, определенные функции (такие как system-property() и key()) могут возвращать ПолноеИмя (имя в виде «prefix:localname») как результат вычисления выражения, и, также как и для любого другого ПолногоИмени, написанного непосредственно в выражении, любые используемые им префиксы пространств имен должны находиться в области действия.

Как объяснялось в первой главе, отсутствие побочных эффектов при вычислении выражения является общим принципом XPath; вычисление выражения не изменит значения каких-либо переменных, не запишет информацию в файл журнала и не попросит у пользователя номер его кредитной карточки. Поэтому вычисление одного и того же выражения в одном и том же контексте более одного раза не должно изменить конечный результат, равно как и смена порядка вычисления выражений не должна повлиять на результат. Поэтому в спецификациях XSLT и XPath обычно ничего не говорится о порядке вычисления. Единственным исключением являются конструкции ВыражениеИЛИ и ВыражениеИ, для которых специально определен порядок вычисления слева направо.

Побочные эффекты могут возникнуть только лишь в результате вычисления выражения, вызывающего написанную пользователем (или поставщиком ПО) функцию расширения, потому что в спецификации XPath не заданы ограничения на действия функций расширения. Равным образом, не гарантируется, что вызовы функций расширения будут производиться в определенном порядке.

Примеры

Множество примеров выражений встречается в тексте данной главы. Вот выборка примеров, собранных вместе, чтобы продемонстрировать разнообразие конструкций, подходящих для этого раздела:

Выражение	Описание
$x + ($y * 2)	Возвращает результат умножения $y на два, сложенный со значением переменной $x.

Выражение	Описание
`//книга | //журнал`	Возвращает набор узлов, содержащий все элементы <книга> и <журнал> того же документа, что и контекстный узел.
`substring-before(автор, ' ')`	Находит строковое значение первого дочернего узла <автор> контекстного узла и возвращает часть этого значения, предшествующую первому пробелу.
`глава and строфа`	Возвращает логическую истину, если у контекстного узла есть дочерние узлы элементов <глава> и <строфа>.
`93.7`	Возвращает числовое значение 93.7.

ВыражениеИ (AndExpr)

Нетривиальное ВыражениеИ предназначено для проверки того, что каждое из двух или более логических условий истинно.

Выражение	Синтаксис
ВыражениеИ	ВыражениеРавенства | ВыражениеИ «and» ВыражениеРавенства

Определено в

XPath, раздел 3.4, правило 22

Используется в

ВыражениеИЛИ

Справка по использованию

Нетривиальное ВыражениеИ состоит из двух или более ВыраженийРавенства, разделенных оператором «and». Тривиальное ВыражениеИ состоит из одного ВыраженияРавенства без оператора «and».

ВыражениеИ вычисляется путем вычисления по очереди каждого ВыраженияРавенства, слева направо, и приведения результата к логическому типу до тех пор, пока не будет получен ложный результат. Как только находится ложный результат, вычисление заканчивается и результатом вычисления выражения становится ложь. Если все ВыраженияРавенства истинны, то результатом вычисления становится истина.

Правила преобразования значения к логическому типу такие же, как и для функции boolean(), описанной в соответствующем разделе главы 7. Например, значение строки нулевой длины или пустого набора узлов – ложь.

В спецификации XPath порядок вычисления определен довольно точно, однако обычно это не имеет значения, так как выражения XPath свободны от

побочных эффектов. Это может влиять на результат в том случае, если один из операндов вызывает определенную пользователем функцию, имеющую побочные эффекты. Кроме того, порядок имеет значение, если при вычислении одного из операндов возникает ошибка. Это означает, что вы можете без опаски написать следующее:

```
<xsl:if test="function-available('my:trace') and my:trace('Вот мы где!')"/>
```

Этот пример вызовет функцию `my:trace()`, если она доступна, в противном случае она не будет вызвана. Что будет возвращено функцией `my:trace()` не имеет значения, так как элемент `<xsl:if>` пустой.

Примеры

Выражение	Описание
`$x > 3 and $x < 8`	Истинно, если значение переменной `$x` больше 3 и меньше 8.
`@имя and @адрес`	Истинно, если у контекстного узла есть атрибуты `имя` и `адрес`. (Оба операнда – наборы узлов, которые преобразуются в логическую истину, если они содержат хотя бы один узел, и в логическую ложь, если они пусты.)
`string(@имя) and string(@адрес)`	Истинно, если у контекстного узла есть атрибуты `имя` и `адрес`, являющиеся строками ненулевой длины. (Оба операнда являются строками, которые преобразуются в логическую истину, если их длина отлична от нуля. Если атрибут отсутствует, набор узлов будет пустым и, следовательно, его строковое значение также будет пустым.)
`true()`	Тривиальное `ВыражениеИ`, состоящее из одного вызова функции.

ВыражениеИЛИ (OrExpr)

Нетривиальный вариант `ВыраженияИЛИ` представляет логическое выражение, истинное, если истинен любой из его операндов.

Выражение	Синтаксис
`ВыражениеИЛИ`	`ВыражениеИ | ВыражениеИЛИ «or» ВыражениеИ`

Определено в

XPath, раздел 3.4, правило 21

Используется в

Выражение

Справка по использованию

Нетривиальный вариант ВыраженияИЛИ состоит из двух или более операндов ВыраженияИ, разделенных оператором «or». Тривиальный вариант ВыраженияИЛИ состоит из одного ВыраженияИ без оператора «or».

Оператор «or» – это обычный оператор «ИЛИ» математической логики: если один или оба операнда истинны, результатом будет истина. Если это необходимо, операнды сначала приводятся к логическому типу неявным вызовом функции `boolean()`.

Спецификация языка утверждает, что правый операнд оператора «or» не вычисляется, если результатом вычисления левого операнда является истина. (Это одно из немногих мест, где явно определяется порядок вычисления.)

Заметьте, что в XPath нет значений, подобных NULL в SQL, и поэтому нет необходимости в трехзначной логике для ситуаций, когда данные отсутствуют. Вместо этого можно явно проверить наличие данных, как показано в некоторых из нижеследующих примеров.

Примеры

Выражение	Описание
`$x = 5 or $x = 10`	Истина, если преобразованное в число значение переменной `$x` равно 5 или 10.
`@name or @id`	Истина, если у контекстного узла есть атрибут `name` или `id` или они оба.
`not(@id) or @id=""`	Истина, если у контекстного узла нет атрибута `id` или его значение – пустая строка.
`//абзац[position()=1 or position()=last()]`	Выбирает элементы `<абзац>`, являющиеся первыми или последними дочерними элементами `<абзац>` своего родительского узла.

ВыражениеОбъединения (UnionExpr)

Нетривиальный вариант ВыраженияОбъединения образует объединение двух наборов узлов; другими словами, в результат входит каждый узел, который есть в одном из входных наборов узлов.

Выражение	Синтаксис
ВыражениеОбъединения	ВыражениеПути | ВыражениеОбъединения «|» ВыражениеПути

Определено в

XPath, раздел 3.3, правило 18

Используется в

УнарноеВыражение

Справка по использованию

Нетривиальный вариант ВыраженияОбъединения состоит из двух или более Выражений Пути, разделенных оператором объединения «|». Этот оператор образует объединение двух или более наборов узлов: в получающийся набор узлов входят все узлы, которые есть хотя бы в одном из входных наборов узлов, без дубликатов.

Тривиальный вариант ВыраженияОбъединения состоит из одного выражения пути, и в этом случае его значением является значение выражения пути.

Оба операнда оператора объединения должны быть наборами узлов. Синтаксис не навязывает это ограничение, он не только допускает выражения типа «$a | $b», где во время выполнения у переменных может оказаться неверный тип, но также допускает совершенно бессмысленные выражения, такие как «2.0 | "Лондон"». Это считается семантической, а не синтаксической ошибкой.

Примеры

Выражение	Описание
*/рисунок | */таблица	Возвращает набор узлов, в котором находятся все внуки контекстного узла, являющиеся элементами <рисунок> или <таблица>.
книга[not(@издательство)] | книга[@издательство='Wrox']	Возвращает все дочерние элементы <книга> контекстного узла, у которых нет атрибута издательство или он равен строке «Wrox». Заметьте, что этот же результат может быть достигнут более эффективно использованием в предикате оператора «or».
(.|..)/заголовок	Возвращает все элементы <заголовок>, которые являются непосредственными дочерними узлами контекстного узла или родителя контекстного узла.
sum((книга|журнал)/@продажи)	Возвращает сумму атрибутов продажи для всех дочерних элементов <книга> и <журнал> контекстного узла.
(//* | //@*) [.='нимбус2000']	Возвращает набор узлов, в котором находятся все узлы элементов и атрибутов документа, строковое значение которых равно «нимбус2000».

В XPath 1.0 не существует операторов, эквивалентных остальным операциям над множествами, таким как пересечение и разность. Некоторые продукты предоставляют функции расширения для заполнения этого пробела, подробности находятся в соответствующих приложениях.

Если вы хотите найти пересечение двух наборов узлов $p и $q, это можно сделать при помощи следующего не совсем очевидного выражения:

```
$p [ count( . | $q) = count( $q ) ]
```

Это выражение выбирает узлы из $p, которые также находятся в $q. Они должны присутствовать в $q, потому что в их объединении с $q столько же узлов, сколько и в самом $q.

Подобным образом, следующее выражение находит узлы, которые есть в $p, но не в $q:

```
$p [ count( . | $q) != count( $q ) ]
```

ВыражениеОтношения (RelationalExpr)

Нетривиальный вариант ВыраженияОтношения сравнивает величину двух чисел. Он предоставляет обычную четверку операторов: меньше, больше, меньше или равно, больше или равно. Когда операндом является набор узлов, сравниваются числовые значения отдельных узлов из набора.

Выражение	Синтаксис
ВыражениеОтношения	АддитивноеВыражение | ВыражениеОтношения «<» АддитивноеВыражение | ВыражениеОтношения «>» АддитивноеВыражение | ВыражениеОтношения «<=» АддитивноеВыражение | ВыражениеОтношения «>=» АддитивноеВыражение

Определено в

XPath, раздел 3.4, правило 24

Используется в

ВыражениеРавенства

Справка по использованию

Нетривиальный вариант ВыраженияОтношения в XPath состоит из двух или более операндов – аддитивных выражений, разделенных одним из операторов: «<», «>», «<=» или «>=».

Тривиальный вариант сравнивающего выражения состоит из одного аддитивного выражения без единого оператора.

XPath определяет язык выражений, который может использоваться в различных контекстах. Основной интересующий нас контекст – таблицы стилей XSLT, в которых выражения всегда находятся в атрибутах XML-элементов. В значении атрибута не может находиться символ «<», поэтому он должен записываться либо как ссылка на численную символьную сущность, ли-

бо как ссылка на сущность «<». На практике многие разработчики предпочитают записывать знак «>» как ссылку на сущность «>», хотя это совершенно необязательно. Эти операторы всегда выполняют численное сравнение. В XPath не существует механизма для лексикографического сравнения строковых значений.

Синтаксически выражение «10 < $x < 30» является допустимым сравнивающим выражением, однако его значение отличается от ожидаемого. На самом деле, оно означает следующее: вычислить значение выражения «(10 < $x)», преобразовать результат логического типа в число, а затем сравнить его с числом 30.

Если ни один из операндов не является набором узлов, то оба оператора преобразуются в числа при помощи функции number() и численно сравниваются по правилам, определенным в IEEE 754. Для обычных чисел это дает ожидаемый результат, например, сравнение «1.0 < 3.5» вернет истину, а «1.0 > 3.5» – ложь. Если одно из чисел – плюс или минус бесконечность, или отрицательный ноль, результат будет совпадать с интуитивно ожидаемым, например, сравнение «1.0 < (2.0 div 0.0)» вернет истину, так как результатом выражения «2.0 div 0.0» является положительная бесконечность. Однако, если один или оба операнда являются не-числами (NaN), результатом всегда будет ложь. Например, сравнение «'ноль' <= 'два'» вернет ложь, потому что преобразование обоих операндов в числа вернет NaN, а сравнение NaN <= NaN вернет ложь.

Если одним из операндов является набор узлов, то результат зависит от типа другого операнда, как показано в следующей таблице. Я буду использовать термин «верно сравнивается с» в значении «меньше», «меньше или равен», «больше» или «больше или равен», в зависимости от того, какой используется оператор: <, <=, > или >=. Предположим, что набором узлов является первый операнд.

Тип второго операнда	Результат
Логический	Сложно представить, зачем кому-нибудь может понадобиться сравнение набора узлов со значением логического типа, но если это действительно необходимо, то вот правила, по которым оно происходит. Пусть P будет результатом преобразования набора узлов в логический тип, а затем в число: результатом будет 0, если набор узлов пустой, или 1, если в нем находится один или более узлов. Пусть Q будет результатом преобразования операнда логического типа в число: 0 для ложного значения и 1 для истинного значения. Конечным результатом будет истина, если P верно сравнивается с Q.
Числовой	Результатом будет истина, если в наборе узлов есть узел, строковое значение которого после преобразования в число верно сравнивается с числовым операндом. Например, сравнение «//цена < 5.00» вернет истину, если в документе есть элемент <цена>, значение которого, преобразованное в число, меньше 5.

Тип второго операнда	Результат
Строковый	Результатом будет истина, если в наборе узлов есть узел, строковое значение которого после преобразования в число верно сравнивается с результатом преобразования строкового операнда в число. Например, сравнение «//цена < '5.00'» вернет истину, если в документе есть элемент <цена>, значение которого, преобразованное в число, меньше 5. Если строка не может быть преобразована в число, результатом всегда будет ложь.

Если набор узлов является вторым операндом, то применяйте правила из таблицы после переворачивания выражения; так, например, выражение «$p < $q» переписывается как «$q > $p».

Наконец, рассмотрим ситуацию, когда оба операнда являются наборами узлов. В этом случае результатом будет истина, только если в первом наборе узлов есть узел P, а во втором – Q, такие, что number(P) верно сравнивается с number(Q). Можно определить результат такого сравнения другим способом, при помощи следующей таблицы, где min() и max() обозначают максимальное и минимальное числовые значения узлов из набора.

Выражение	Результат
M < N	Истина, когда min(M) < max(N).
M <= N	Истина, когда min(M) <= max(N).
M > N	Истина, когда max(M) > min(N).
M >= N	Истина, когда max(M) >= min(N)

На практике чаще всего сравниваются наборы узлов, состоящие из одного узла. Например, выражение «@цена > 5.00» истинно, если атрибут цена существует и его числовое значение больше 5.00; оно ложно, если атрибут цена не существует или его значение меньше либо равно 5.00.

Когда один или оба операнда являются деревьями, дерево рассматривается как набор узлов, состоящий из единственного узла – корневого узла дерева. Так как в наборе всегда находится только один узел, правила похожи на правила сравнения строк или чисел, например, переменную, объявленную следующим образом:

```
<xsl:variable name="limit">142</xsl:variable>
```

можно использовать в сравнивающем выражении так, как будто бы она была объявлена как число:

```
<xsl:variable name="limit" select="142"/>
```

или как строка:

```
<xsl:variable name="limit" select="'142'"/>
```

Примеры

Выражение	Описание
`count(*) > 10`	Истина, если у контекстного узла более 10 дочерних элементов.
`sum(ПРОДАЖИ) < 10000`	Истина, если сумма числовых значений дочерних элементов <ПРОДАЖИ> контекстного узла меньше десяти тысяч.
`position() < last() div 2`	Истина, если контекстная позиция меньше половины размера контекста, то есть если узел находится в первой половине списка обрабатываемых узлов.
`not(//@temp <= 0.0)`	Истина, если все значения атрибута temp в документе больше нуля.

ВыражениеПути (PathExpr)

ВыражениеПути — это выражение для выбора набора узлов путем прохождения маршрута (последовательности из одного или более шагов), начиная с заданной начальной точки. Начальной точкой пути может быть контекстный узел, корневой узел или произвольный набор узлов, являющийся, например, значением переменной или результатом вызова функции.

Выражение	Синтаксис
ВыражениеПути	МаршрутПоиска | ФильтрующееВыражение | ФильтрующееВыражение «/» ОтносительныйМаршрутПоиска | ФильтрующееВыражение «//» ОтносительныйМаршрутПоиска

Определено в

XPath, раздел 3.3, правило 19

Используется в

ВыражениеОбъединения

Справка по использованию

Нетривиальный вариант ВыраженияПути задает набор узлов. Однако поскольку тривиальный вариант ФильтрующегоВыражения может состоять из одного лишь числа или литерала, тривиальный вариант может задавать значение любого типа. Нетривиальный вариант ВыраженияПути может задавать набор узлов несколькими способами:

- Маршрутом поиска, представляя набор узлов, выбранных проходом последовательности шагов, начинающейся с корня документа или с контекстного узла.

- ФильтрующимВыражением, которое может быть ссылкой на переменную, задающую набор узлов, вызовом функции, возвращающим набор узлов, выражением, заключенным в скобки, таким как «($a | $b)», или любой из вышеперечисленных конструкций, за которой следует один или более предикатов.
- ФильтрующимВыражением, за которым следует оператор пути «/» и относительный маршрут поиска, который задает последовательность шагов, начинающуюся с набора узлов заданных ФильтрующимВыражением, а не с корневого узла или контекстного узла, как для обычного маршрута поиска.
- ФильтрующимВыражением, за которым следует оператор сокращенного пути «//» и относительный маршрут поиска: как и в других контекстах, оператор «//» является сокращением для выражения «/descendant-or-self::node()/».

Примеры

Выражение	Описание
абзац	Это выражение пути является маршрутом поиска, выбирающим все дочерние элементы <абзац> контекстного узла.
абзац[@id]	Это выражение пути является маршрутом поиска, выбирающим все дочерние элементы <абзац> контекстного узла, у которых есть атрибут id.
абзац/@id	Это выражение пути является маршрутом поиска, выбирающим атрибуты id всех дочерних элементов <абзац> контекстного узла. Отличие от предыдущего примера заключается в том, что результатом является набор узлов атрибутов, а не набор узлов элементов.
/*/абзац	Это выражение пути является маршрутом поиска, выбирающим все дочерние элементы <абзац> элемента документа (самого внешнего элемента документа).
$абзацы	Данное выражение пути состоит из ФильтрующегоВыражения, состоящего, в свою очередь, из ссылки на переменную. Оно не обязательно является набором узлов.
$абзацы[23]	Данное выражение пути состоит из ссылки на переменную, фильтрованную предикатом. Оно выбирает 23-й узел в наборе узлов, являющемся значением переменной $абзацы.
document('lookup.xml')	Данное выражение пути состоит из ФильтрующегоВыражения, состоящего, в свою очередь, из вызова функции. Оно выбирает корневой узел XML-документа, заданного URI lookup.xml.
key('имясотр', 'Иван Петров') [@местоположение='Москва']	Данное выражение пути состоит из ФильтрующегоВыражения, состоящего из ссылки на переменную, фильтрованную предикатом. В предположении, что ключ «имясотр» был определен очевидным образом, оно выбирает всех сотрудников с именем Иван Петров, находящихся в Москве.

Выражение	Описание
(//раздел|//подраздел)[заголовок='Введение']	Данное выражение пути состоит из ФильтрующегоВыражения, состоящего из заключенного в скобки и фильтрованного предикатом ВыраженияОбъединения. Оно выбирает все элементы <раздел> и <подраздел>, у которых есть дочерний элемент <заголовок>, содержимым которого является строка «Введение».
$разделы/тело	Это выражение пути выбирает все дочерние элементы <тело> узлов из набора, заданного переменной $разделы.
$разделы[3]/тело	Это выражение пути выбирает все дочерние элементы <тело> третьего узла из набора узлов, заданного переменной $разделы.
(//раздел|//подраздел)//абзац	Это выражение выбирает всех потомков <абзац> элементов <раздел> и <подраздел>.
((//раздел|//подраздел)//абзац)[last()]	Данное выражение пути выбирает последний (в порядке документа) элемент <абзац>, являющийся потомком элемента <раздел> или <подраздел>.

Из схемы, находящейся в начале главы, очевидно, что большинство сложностей в синтаксисе XPath связано с конструкцией ВыражениеПути. Фактические порождающие правила довольно запутанны, и их нелегко соблюдать, но они существуют для облегчения написания выражений, особенно, когда вы хорошо знакомы с тем, как записываются пути к каталогам и файлам в UNIX.

Если мы на время забудем о сокращении «//» и представим себе функцию root(), находящую корневой узел, то становится возможной запись любого выражения пути в виде ФильтрующегоВыражения («/» Шаг)*, значительно упрощающая порождающие правила. Остальные варианты позволяют обозначать корневой узел как «/», начинать абсолютные выражения пути с «/» и опускать начальные символы «./» в относительных выражениях пути.

ВыражениеРавенства (EqualityExpr)

Нетривиальный вариант ВыраженияРавенства используется для определения равенства или неравенства двух значений.

Замечание: информацию обо всех других операторах сравнения («<», «<=», «>» и «>=») можно найти в разделе ВыражениеОтношения на стр. 404.

Выражение	Синтаксис
ВыражениеРавенства	ВыражениеОтношения | ВыражениеРавенства «=» ВыражениеОтношения | ВыражениеРавенства «!=» ВыражениеОтношения

Определено в

XPath, раздел 3.4, правило 23

Используется в

ВыражениеИ

Справка по использованию

Нетривиальный вариант ВыраженияРавенства состоит из одного или более операндов ВыражениеОтношения, разделенных операторами «=» («равно») или «!=» («не равно»). Тривиальный вариант ВыраженияРавенства состоит из одного ВыраженияОтношения без операторов «=» или «!=».

Результатом проверки на равенство (или неравенство) всегда является логическая истина или ложь. Тем не менее, во многих ситуациях результат вычисления выражения неочевиден, поэтому я подробно рассмотрю различные случаи.

Безобидно выглядящий оператор «=» может преподнести много сюрпризов в XPath, поэтому стоит подробно изучить правила его употребления, даже если они выглядят запутанными. Ниже приводится список наиболее распространенных ошибок:

- Нельзя рассчитывать на то, что выражение «$X=$X» всегда истинно. Обычно это так, но если «$X» – пустой набор узлов, то это выражение вернет ложь.
- Нельзя предполагать, что выражение «$X!=3» означает то же самое, что и «not($X=3)». Если «$X» – набор узлов, то первое выражение будет истинным, если любой узел из набора узлов не равен 3, а второе – если в наборе узлов нет узла, равного 3.
- Нельзя предполагать, что если «$X=$Y and $Y=$Z», то «$X=$Z». В этом случае также виноваты наборы узлов. Два набора узлов считаются равными, если в них есть одинаковое значение, то есть результат сравнений {2, 3} = {3, 4} и {3, 4} = {4, 5} – истина, но сравнения {2, 3} = {4, 5} – ложь.

В этом странном мире Оруэлла, где некоторые значения равнее других, есть только одно утешение: «$X=$Y» всегда означает то же самое, что и «$Y=$X».

Левым операндом ВыраженияРавенства может быть другое ВыражениеРавенства. Это означает, что выражение «$A = $B = $C» является допустимым. Однако от него мало проку и, скорее всего, оно не произведет желаемого действия. Оно означает то же, что и выражение «($A=$B)=$C», и проверяет, совпадает ли приведенное к логическому типу значение $C с результатом сравнения $A и $B. Это означает, например, что результат вычисления выражения «2=1=0» – истина, потому что «2=1» – ложь, и приведение нуля к логическому типу также дает ложь, поэтому выражение «2=1» эквивалентно выражению «boolean(0)».

Сравнение простых значений

Сначала рассмотрим случай, когда оба операнда являются простыми значениями: логическими значениями, числами или строками. Если у двух значений разный тип, то:

- если одно из них логического типа, то и другое приводится к логическому типу, и они сравниваются как значения логического типа
- в противном случае, если одно из них – число, то и другое преобразуется в число, и они сравниваются как числа
- в остальных случаях они сравниваются как строки

Действия этих правил сведены в следующей таблице:

«=»	Логический	Числовой	Строковый
Логический	Истина, если оба операнда истинны или оба ложны. Ложь, если один из операндов – истина, а другой – ложь.	Истина, если операнд логического типа истинен, а число отлично от нуля и от NaN; или если операнд логического типа – ложь, а число равно нулю или NaN.	Истина, если операнд логического типа истинен и строка не пуста, или если операнд логического типа – ложь, а строка пуста.
Числовой	Истина, если операнд логического типа истинен, а число отлично от нуля и NaN; или если операнд логического типа – ложь, а число равно нулю или NaN.	Истина только в том случае, если оба операнда численно равны по определению IEEE 754.	Истина только в том случае, если строка, преобразованная в число по правилам функции number(), численно равна числу по определению IEEE 754.
Строковый	Истина, если операнд логического типа истинен и строка не пуста, или если операнд логического типа – ложь, а строка пуста.	Истина только в том случае, если строка, преобразованная в число по правилам функции number(), численно равна числу по определению IEEE 754.	Истина только в том случае, если оба операнда содержат совпадающую последовательность символов Unicode.

Подробности определения численного сравнения по IEEE 754 см. в разделе «Числовые значения» главы 2. Например, положительный ноль и отрицательный ноль считаются равными, а два не-числа – нет.

Для простых значений (включая особые числовые значения, такие как не-число) результат применения оператора «!=» (не равно) всегда противоположен результату применения оператора «=»: если оператор «=» возвращает истину, то оператор «!=» возвращает ложь, и наоборот.

Сравнения с наборами узлов

Рассмотрим теперь случай, когда по крайней мере одним из операндов является набор узлов. (И помните, что даже такое невинное выражение, как «НА-

ЗВАНИЕ» или «@HREF», на самом деле является набором узлов, даже если он содержит всего лишь один узел.) Результаты сравнения операторами «=» или «!=» показаны в нижеследующей таблице. Выберите строку таблицы, соответствующую типу одного из операндов, а другой операнд – всегда набор узлов.

Тип второго операнда	= (равно)	!= (не равно)
Логический	Истина, если логическое значение – истина, а в наборе узлов находится по меньшей мере один узел, или если логическое значение – ложь, а набор узлов пуст.	Истина, если логическое значение – истина, а набор узлов пуст, или если логическое значение – ложь, а в наборе узлов находится по меньше мере один узел.
Число	Истина, если в наборе узлов есть узел, строковое значение которого, преобразованное в число функцией `number()`, численно равно числовому операнду по определению IEEE 754.	Истина, если в наборе узлов есть узел, строковое значение которого, преобразованное в число функцией `number()`, численно не равно числовому операнду по определению IEEE 754.
Строка	Истина, если в наборе узлов есть узел, строковое значение которого равно строковому операнду.	Истина, если в наборе узлов есть узел, строковое значение которого не равно строковому операнду.
Набор узлов	Истина, если существуют два узла, взятые по одному из каждого набора узлов, с одинаковым строковым значением. Заметьте, это означает, что если один или оба набора узлов пусты, результатом всегда будет ложь. Если один и тот же узел находится в обоих наборах узлов, то результатом всегда будет истина.	Истина, если существуют два узла, взятые по одному из каждого набора узлов, с различающимся строковым значением. Заметьте, это означает, что если один или оба набора узлов пусты, результатом всегда будет ложь. Это также означает, что если в любом наборе узлов находится более одного узла и если не у всех них совпадающие строковые значения, то результатом всегда будет истина.

Так что, если набор узлов «$N» сравнивается со строкой «'мэри'», проверка «$N='мэри'» – это просто короткий вариант записи следующей последовательности действий: проверить, есть ли в наборе узлов $N такой узел n, что string-value(n)='мэри'. Аналогично, проверка «$N!='мэри'» – короткий вариант записи следующей последовательности действий: проверить, есть ли в наборе узлов $N такой узел n, что string-value(n)!='мэри'. Если в наборе узлов $N находятся два узла со строковыми значениями "мэри" и "джон", то оба выражения «$N='мэри'» и «$N!='мэри'» вернут истину, потому что есть узел как, равный строке 'мэри', так и не равный ей. Если набор узлов $N пуст, то выражения «$N='мэри'» и «$N!='мэри'» вернут ложь, потому что в нем нет узла, равного строке 'мэри', но также нет и узла, не равного ей.

Учтите, что когда мы говорим об узлах в наборе узлов, мы имеем в виду только узлы, по праву являющиеся членами набора узлов. Их дочерние узлы не являются членами набора узлов.

Пример: Сравнения с наборами узлов

Рассмотрим, например, следующий фрагмент XML:

```
<книги>
  <книга><автор>Адам</автор><название>Пингвины</название></книга>
  <книга><автор>Бетти</автор><название>Жирафы</название></книга>
</книги>
```

Предположим, что мы хотим создать переменную, значением которой будет набор узлов со всеми элементами <книга>, следующим образом:

```
<xsl:variable name="все-книги" select="//книга"/>
```

И предположим, что у нас есть следующее условие:

```
<xsl:if test="$все-книги = 'Адам'"/>
```

Результатом будет ложь, так как в наборе узлов $все-книги находятся два узла <книга>, и строковые значения обоих отличны от строки "Адам". Строковое значение первого элемента <книга> – "АдамПингвины", а второго – "БеттиЖирафы". Тот факт, что у одного из них есть дочерний узел со строковым значением "Адам", не имеет значения: дочерний узел не является членом набора узлов.

Строковое значение узла зависит от типа узла. Для текстового узла это символы, находящиеся внутри элемента. Для узла элемента это сумма строковых значений всех его дочерних текстовых узлов. Для узла атрибута это значение атрибута. Более подробные правила находятся во второй главе.

У правил сравнения наборов узлов есть интересное следствие: если $N – пустой набор узлов, то результатом проверки условия «$N=$N» будет ложь, так как в первом наборе нет узла со строковым значением, равным строковому значению какого-либо узла из второго набора узлов.

На этих правилах можно запросто споткнуться, предполагая, к примеру, что условие <xsl:if test="@имя!= 'Джеймс'"> означает то же самое, что и <xsl:if test="not(@имя = 'Джеймс')">. Это не одно и то же: если атрибут имя не существует, то проверка первого условия вернет ложь, в то время как проверка второго – истину.

Вообще говоря, избегайте оператора «!=», если вы не до конца уверены в том, что вы делаете. Используйте вместо него выражение «not(x=y)», оно лучше соответствует интуитивному представлению.

Однако существует ситуация, в которой может пригодиться оператор «!=», используемый с наборами узлов: проверка совпадения значений всех узлов из набора. Например, выражение <xsl:if test="not($документы//версия!=1.0)">

проверяет, есть ли хоть один узел в наборе узлов «$документы//версия» с числовым значением, отличным от 1.0.

Важно помнить, что проверка на равенство сравнивает строковые значения узлов, а не проверяет их тождественность. Например, выражение «../=/» может показаться естественным способом проверки того, что родитель контекстного узла – корневой узел. На самом деле, результатом проверки этого условия будет истина также и в том случае, если родительским узлом будет элемент документа, потому что в правильно построенном дереве строковое значение элемента документа совпадает со строковым значением корневого узла. Это не только неверная проверка, но также и очень ресурсоемкая: строковое значение корневого узла состоит из всего текста документа, так что понадобится создать две строки, каждая из которых может быть в миллион символов длиной, а затем сравнить их.

Более приемлемым способом проверки, не является ли родительский узел текущего узла корневым, будет выражение «parent::*[not(..)]»: оно истинно только для узла, у которого есть родитель, но нет «дедушки». Другая возможность, которая подходит для сравнения любых наборов узлов, состоящих из единственного узла, состоит в использовании выражения «generate-id(..)=generate-id(/)». Этот способ основывается на функции generate-id(), описанной в главе 7, возвращающей строку, уникальным образом идентифицирующую узел. Наконец, проверку тождественности узла можно произвести при помощи оператора «|» и функции count(): выражение «count(../ /)=1 and ..» будет истинным только в том случае, если родительский и корневой узел являются одним и тем же узлом. Вторая часть выражения – «and ..» – нужна, потому что без нее выражение будет истинным также и для узла, у которого нет родителя, то есть для корневого узла.

Сравнения с деревьями

Когда одним из операндов ВыраженияРавенства является корень дерева, то действуют приведенные выше правила для наборов узлов. Однако в таком наборе узлов всегда будет находиться ровно один узел, так что во многих отношениях такое сравнение будет напоминать сравнение строк.

Деревья могут возникать в результате различных действий. Выражение «/» ссылается на дерево, так же как и возвращаемое значение функции document() (хотя в последнем случае оно также может быть и набором узлов с корневыми узлами нескольких деревьев). Кроме того, существует возможность создать временное дерево при помощи элементов <xsl:variable> или <xsl:param> без атрибута select. Одно из отличий от исходного дерева документа состоит в том, что исходное дерево служит представлением правильно построенного XML-документа, у корневого узла которого есть только один дочерний узел элемента и нет дочерних текстовых узлов, а временное дерево должно быть лишь сбалансированным: дочерними узлами корня дерева может быть любое число узлов элементов и текстовых узлов.

Эти правила означают, что когда сравнивается переменная, значением которой является дерево, со строкой, результат будет всегда истиной, если стро-

ковое значение корневого узла дерева совпадает со строкой. Так что обычная переменная, хранящая дерево и объявленная следующим образом:

```
<xsl:variable name="город">Осака</xsl:variable>
```

ведет себя так же, как и строка, объявленная следующим образом:

```
<xsl:variable name="город" select="'Осака'"/>
```

В обоих случаях выражение «$город='Осака'» будет истинным, а «$город='Токио'» – ложным.

Переменная, которой присвоено дерево, может иметь более сложную структуру, например:

```
<xsl:variable name="дерево">Ну <emph>очень</emph> важный господин</xsl:variable>
```

Дерево этого примера состоит из корневого узла с тремя дочерними узлами: текстовый узел для строки «Ну®» (где ♦ обозначает символ пробела), узел элемента <emph> и текстовый узел для строки «®важный®господин». Строковым значением этой переменной является строка «Ну♦очень♦важный♦господин». Сравнение при помощи операторов «=» и «!=» с переменной $дерево будет давать такой же результат, как и сравнение со строкой.

Подытожим все вышесказанное: когда один из операндов оператора сравнения «=» или «!=» – дерево, сравнение производится таким же образом, как если бы этот операнд был набором узлов с единственным узлом – корнем дерева. Результаты сравнений показаны в нижеследующей таблице:

	= (равно)	!= (не равно)
Логический	Истина, если значением операнда логического типа является истина, в противном случае ложь; содержимое дерева значения не имеет, так как в нем всегда есть хотя бы один узел.	Истина, если значением операнда логического типа является ложь, в противном случае результатом сравнения будет ложь; содержимое дерева значения не имеет, так как в нем всегда есть хотя бы один узел.
Число	Истина, если строковое значение дерева, преобразованное в число функцией number(), численно равно числовому операнду по определению IEEE 754.	Истина, если строковое значение дерева, преобразованное в число функцией number(), численно не равно числовому операнду по определению IEEE 754.
Строка	Истина, если строковое значение дерева равно строковому операнду.	Истина, если строковое значение дерева не равно строковому операнду.
Набор узлов	Истина, если в наборе узлов есть узел, строковое значение которого равно строковому значению дерева.	Истина, если в наборе узлов есть узел, строковое значение которого не равно строковому значению дерева.

Эти правила означают, что для всех практических применений переменная, определенная следующим образом:

```
<xsl:variable name="город">Йоханнесбург</xsl:variable>
```

будет вести себя точно так же, как и следующая переменная строкового типа:

```
<xsl:variable name="город" select="'Йоханнесбург'"/>
```

На самом деле результат отличается только в одном случае: а именно, при сравнении со значением логического типа. Результатом сравнения «$город=false()» будет истина, если $город – строка нулевой длины, но он всегда будет ложным, если $город – дерево, независимо от его значения.

Заметьте, что Рабочий проект стандарта XSLT 1.1 не изменяет это; хотя фрагменты конечного дерева не являются в нем отдельным типом данных, семантика проверки равенства остается в нем такой же.

Примеры

Выражение	Описание
`@ширина = 3`	Проверяет, равняется ли преобразованное в число значение атрибута ширина контекстного узла числу 3. Результатом будет истина, если значением атрибута ширина является строка «3», или, скажем, «3.00». Если у контекстного узла нет атрибута ширина – результатом будет ложь.
`@ширина = @высота`	Проверяет, не совпадает ли строковое значение атрибутов ширина и высота контекстного узла. Если значение атрибута ширина – «3», а высота – «3.00», результатом будет ложь. Он также будет ложью, если один или оба атрибута отсутствуют. Если вы хотите сравнивать значения численно, используйте функцию number(), описанную в главе 7, для принудительного преобразования значений в числа.
`@ширина != $x`	Если в переменной $x находится числовое значение, то выполняется численное сравнение; если в ней находится строковое значение, то выполняется строковое сравнение. Результат всегда будет истинным, если значения отличаются. Он будет ложным, если отсутствует атрибут ширина. Если в переменной $x находится набор узлов, результатом будет истина, если в наборе узлов есть узел со строковым значением, равным значению атрибута ширина; результатом будет ложь, если набор узлов пуст или атрибут ширина отсутствует.

ИмяБезДвоеточия (NCName) и СимволИмениБезДвоеточия (NCNameChar)

ИмяБезДвоеточия – это имя или часть имени без двоеточия. XPath заимствует это определение из спецификации XML Namespaces Recommendation.

Не существует формальной расшифровки английского сокращения NCName, однако приемлем вариант «no-colon-name» («имя-без-двоеточия»).

Правило СимволИмениБезДвоеточия определяет символы, которые могут присутствовать в ИмениБезДвоеточия.

Выражение	Синтаксис
ИмяБезДвоеточия	(Буква | «_») (СимволИмениБезДвоеточия) *
СимволИмениБезДвоеточия	Буква | Цифра | «.» | «-» | «_» | КомбинирующийСимвол | Расширитель

Правила Буква (Letter), Цифра (Digit), КомбинирующийСимвол (CombiningChar) и Расширитель (Extender) находятся в спецификации XML. Определения даны в виде длинных списков символов Unicode, и повторение их здесь имело бы небольшую пользу. Основной принцип заключается в том, что если имя допустимо в XML, то оно также допустимо и в XPath.

Неформально говоря, ИмяБезДвоеточия начинается с буквы или значка подчеркивания, за которым следует ноль или более символов СимволИмениБезДвоеточия, которые могут быть буквами, цифрами, точкой, дефисом или символом подчеркивания. Категории Буква и Цифра включают в себя широкий выбор символов и идеограмм национальных (не латинских) алфавитов и латинские буквы с акцентами, а категории КомбинирующийСимвол и Расширитель включают в себя акценты и диакритические знаки многих языков.[1]

Определено в

Спецификация XML Namespaces Recommendation

Используется в

ПолноеИмя

КритерийИмени

Справка по использованию

ИмяБезДвоеточия используется в синтаксисе XPath в двух случаях: как часть ПолногоИмени, подробно обсуждаемого на стр. 436, и как часть КритерияИмени, описанного на стр. 420.

В обоих контекстах имя используется для сопоставления с именами, встречающимися в исходном документе XML, и по этой причине синтаксис должен соответствовать синтаксису имен XML.

[1] Здесь уместно напомнить, что XSLT-процессоры обрабатывают документы в кодировке Unicode, что и позволяет корректно обрабатывать символы национальных алфавитов и многоязычные документы. – *Примеч. ред.*

Кроме правил имен XML, в XPath включены дополнительные правила, определенные в спецификации XML Namespaces. Архитекторы языка XPath решили, что манипулировать XML-документом средствами XPath будет невозможно, если XML-документ не соответствует правилам, определенным в спецификации XML Namespaces, например, правилу, что имя не может содержать более одного двоеточия.

Как и в XML, имена чувствительны к регистру и совпадают только в том случае, если они состоят из в точности совпадающей последовательности символов. Это верно даже тогда, когда по стандартам Unicode символы являются эквивалентными, например, существуют разные способы написания букв с акцентами.

Примеры

Вот примеры допустимых `ИменБезДвоеточия`:

```
A
a
π
ℵ
_system-id
iso-8859-1
счет.адрес
StraЯenьberfьhrung
ΕΛΛΑΣ
_···_---_···_
```

ИмяОператора (OperatorName)

Оператор, записываемый как имя.

Выражение	Синтаксис
`ИмяОператора`	`«and» | «or» | «mod» | «div»`

Определено в

XPath, раздел 3.7 (Lexical Rules), правило 33

Используется в

`ЭлементВыражения`

Справка по использованию

Порождающее правило для ИмениОператора является частью лексических правил XPath; имя ИмяОоператора – это экземпляр конструкции ЭлементВыражения, то есть лексема. Так как имена операторов являются лексемами, то за и перед ними, но не внутри их, могут находиться пробельные символы. На практике пробельные символы могут понадобиться, чтобы отделить имя оператора от соседней лексемы, если она также состоит из букв и цифр.

Четыре имени операторов не являются зарезервированными словами. В исходном XML-документе может находиться элемент с именем (и в самом деле, элемент с таким именем определяется в XHTML), и на него можно ссылаться обычным способом, используя выражение пути «ancestor::div» или просто «div». Решение того, является ли «div» (или «and», «or», «mod») именем без двоеточия или именем оператора, принимается в зависимости от контекста: эта лексема распознается как ИмяБезДвоеточия, если она первая в выражении, или если перед ней находится одна из следующих лексем: «@», «::», «,», «(», «[» или оператор.

Примеры в контексте

Выражение	Описание
`$x = 5 or $x = 10`	Истина, если преобразованное в число значение переменной $x равно 5 или 10.
`position() mod 2`	Ноль, если контекстная позиция является четным числом, в противном случае – единица.
`floor(` `string-length(@name) div 2)`	Половина длины значения атрибута name округленная вниз.
`@name and @id`	Истина, если у контекстного узла есть атрибуты name и id.

ИмяФункции (FunctionName)

ИменемФункции может быть любое допустимое имя, кроме имени, зарезервированного для типа узла.

Выражение	Синтаксис
ИмяФункции	ПолноеИмя - ТипУзла

Знак «минус» в порождающем правиле означает «кроме»; другими словами, именем функции может быть любое имя, кроме используемого для ТипаУзла. На практике выбор имен еще более ограничен, так как имена в пространстве имен по умолчанию (без двоеточий) могут использоваться только для стандартных функций, список которых определен в спецификациях XPath и XSLT.

Определено в

XPath, раздел 3.7 (Lexical Structure), правило 35

Используется в

ПервичноеВыражение

Справка по использованию

С точки зрения синтаксиса, именем функции может быть любое имя, кроме одного из четырех названий типов узлов: «comment», «text», «processing-instruction» или «node». Синтаксически имя функции является именем, за которым следует открывающая скобка.

Только в именах системных функций (перечисленных в главе 7) нет префикса пространства имен. В имена функций расширения, определенных пользователем или поставщиком ПО, всегда включается префикс пространства имен. Интерпретация префикса зависит от объявлений пространств имен, находящихся в области действия для элемента таблицы стилей, содержащего выражение, в котором находится имя функции.

Примеры

Выражение	Определение
concat string-length namespace-uri	Имена системных функций.
saxon:intersection xt:node-set irs:get-tax-rate	Имена функций расширения.

КритерийИмени (NameTest)

КритерийИмени является или именем, или обобщенным именем, заданным при помощи символов подстановки.

Определено в

XPath, раздел 3.7 (Lexical Rules), правило 37

Выражение	Синтаксис
КритерийИмени	«*» | ИмяБезДвоеточия «:» «*» | ПолноеИмя

Заметьте, что КритерийИмени является лексемой, а это означает, что в нем не могут находиться пробельные символы. Поэтому второй вариант в порождающем правиле можно записать более естественным образом: ИмяБезДвоеточия «:*».

Используется в

КритерийУзла

КритерийИмени используется не только в выражениях XPath, но также и в некоторых контекстах XSLT. Например, у элементов <xsl:preserve-space> и <xsl:strip-space> есть атрибут, значением которого является список критериев имен, разделенных пробельными символами.

Справка по использованию

В общем случае критерий имени будет соответствовать одним именам и не соответствовать другим.

Критерий имени «*» соответствует любому имени. (Но при использовании в качестве самостоятельного выражения символ «*» является сокращенным вариантом записи пути «child::*», выбирающего все дочерние элементы контекстного узла. Результат ограничен только узлами элементов, потому что символ «*» при использовании в Шаге выбирает только узлы, соответствующие основному типу оси, а для всех типов осей, кроме осей attribute и namespace, основным типом узлов являются узлы элементов.)

> ***Неожиданным результатом этого правила является то, что нельзя написать:***
>
> ```
> <xsl:copy-of select="@*[not(self::название)]"/>
> ```
>
> ***чтобы скопировать все атрибуты элемента, кроме атрибута*** название***. Почему? Потому что основным типом узла для оси self является узел элемента, так что если контекстный узел является атрибутом*** название***, выражение*** «self::название» ***не выберет его. Чтобы избежать этого, пишите так:***
>
> ```
> <xsl:variable name="название" select="generate-id(@название)"/>
> <xsl:copy-of select="@*[generate-id(.)!=$название]"/>
> ```

Критерий имени «xyz:*» выбирает любое имя с пространством имен, связанным с префиксом «xyz». Проверяемое имя необязательно должно иметь такой же префикс, достаточно того, чтобы префикс ссылался на тот же самый URI пространства имен.

Критерий имени «xyz:item» выбирает любое имя с пространством имен, связанным с префиксом «xyz» и локальной частью «item». Проверяемое имя необязательно должно иметь такой же префикс, достаточно того, чтобы префикс ссылался на тот же самый URI пространства имен.

Критерий имени «item» (без префикса пространства имен) выбирает любое имя с локальной частью «item» и с пустым URI пространства имен. Пространство имен по умолчанию не используется.

Если в документе используется объявление пространства имен по умолчанию, такое как «xmlns="some.uri"»***, то элемент*** <item> ***исходного документа не будет выбран таким выражением XPath как*** «//item»***, даже если в таблице стилей находится такое же объявление пространства имен*** «xmlns="some.uri"»***. Это происходит потому, что в выражениях XPath игнорируется пространство имен по умолчанию. Вы должны явно объявить пространство имен, например*** «xmlns:x="some.uri"»***, и ссылаться на элемент, используя префикс:*** «//x:item»**.**

Примеры

Выражение	Описание
*	Соответствует любому имени. Если «*» используется сам по себе, он является относительным маршрутом поиска «child::*», выбирающим все дочерние элементы контекстного узла, независимо от их имени.
xt:*	Соответствует любому имени в пространстве имен, связанном с префиксом «xt». Если выражение «xt:*» используется само по себе, оно является относительным маршрутом поиска «child::xt:*», выбирающим все дочерние элементы контекстного узла из пространства имен, связанного с префиксом «xt».
заголовок	Соответствует любому имени с локальной частью «заголовок» и с пустым URI пространства имен.
wrox:заголовок	Соответствует любому имени с пространством имен, связанным с префиксом «wrox» и локальной частью «заголовок».

КритерийУзла (NodeTest)

КритерийУзла проверяет соответствие узла заданным ограничениям на его тип или его имя.

Выражение	Синтаксис
КритерийУзла	КритерийИмени | ТипУзла «(» «)» | «processing-instruction» «(» Литерал «)»

Заметьте, что второй вариант уже допускает выражение «processing-instruction()»; третий вариант нужен только в том случае, когда необходимо задать имя инструкции обработки.

Третья форма необычна из-за того, что это единственное место в XPath, где имя узла исходного документа пишется в кавычках. Причина этого, вероятно, в том, что архитекторы языка хотели сохранить возможность использовать в будущих версиях спецификации конструкции, подобные node() и comment(), как вызовы функций; если бы на месте литерала стояло ИмяБезДвоеточия, это привело бы к непоследовательности семантики относительно правил вызова функций.

Определено в

XPath, раздел 2.3, правило 7

Используется в

Шаг

Справка по использованию

КритерийУзла используется в Шаге для задания типа и/или имени узлов, которые должны быть им выбраны.

В общем случае задается либо имя узлов, либо их тип. Если вы задаете КритерийИмени, неявно происходит выборка узлов того же типа, что и основной тип узлов для оси, используемой в Шаге. Для оси attribute выбираются узлы атрибутов; для оси namespace выбираются узлы пространств имен; для всех остальных осей выбираются узлы элементов.

Задание в качестве ТипаУзла выражения «node()» выбирает все узлы на оси. Нужно задать тип «node()», если необходимо, чтобы Шаг выбирал узлы более чем одного типа.

Указание типа «processing-instruction()», «comment()» или «text()» в качестве типа узла выбирает узлы указанного типа. Нет смысла задавать эти типы узлов на оси attribute или namespace, потому что они на этих осях не встречаются. У этих узлов нет имен, кроме узлов инструкций обработки, для этого единственного случая и существует вариант синтаксиса, позволяющий задать как тип узла, так и его имя.

Не существует критерия узла для корневого узла. Если на оси (например, ancestor или parent) находится корневой узел, он будет выбран тогда и только тогда, если критерием узла является «node()». Если вы ищете именно корневой узел, то вам не нужна ось, чтобы найти его, потому что для этого существует маршрут поиска «/».

Примеры

Выражение	Описание
ЗАГОЛОВОК	Данный критерий имени выбирает все элементы <ЗАГОЛОВОК>, если он не используется с осью attribute (в выражениях вида «attribute::ЗАГОЛОВОК» или «@ЗАГОЛОВОК»), на которой он выберет атрибут ЗАГОЛОВОК, или осью namespace (например, «namespace::ЗАГОЛОВОК»), на которой он выберет пространство имен с префиксом ЗАГОЛОВОК.
новости:сообщение	Данный критерий имени выбирает все узлы с локальным именем «сообщение» в пространстве имен «новости». Это могут быть узлы атрибутов или элементов, в зависимости от используемой оси. В таб-

Выражение	Описание
	лице стилей должен быть объемлющий элемент, объявляющий префикс «новости», используя для этого атрибут «xmlns:новости="uri"». Узел в исходном документе должен иметь имя с тем же URI пространства имен, но необязательно совпадающим префиксом.
MathML:*	Этот критерий имени выбирает все узлы, имена которых находятся в пространстве имен MathML. Это могут быть узлы атрибутов или элементов, в зависимости от используемой оси. В таблице стилей должен быть объемлющий элемент, объявляющий данный префикс, используя для этого атрибут «xmlns:MathML="*uri*"».
*	Данный критерий имени выбирает все элементы, если он не используется с осью attribute (в выражениях вида «attribute::*» или «@*»), на которой он выберет все атрибуты, или осью namespace (например, «namespace::*»), на которой он выберет все пространства имен.
text()	Этот критерий узла выбирает все текстовые узлы на соответствующей оси.
processing-instruction()	Этот критерий узла выбирает все узлы инструкций обработки на соответствующей оси. Заметьте, что XML-объявление в начале документа *не* является инструкцией обработки, несмотря на внешнее сходство.
processing-instruction('ckpt')	Этот критерий узла выбирает все инструкции обработки с именем (в спецификации XML оно называется целью инструкции обработки, PITarget) «ckpt»: например, инструкцию обработки <?ckpt периодичность=ежедневно?>.
node()	Этот критерий узла выбирает все узлы на соответствующей оси.

Литерал (Literal)

Литерал представляет собой константную строку.

Выражение	Синтаксис
Литерал	«"» [^"]* «"» | «'» [^']* «'»

Если вы незнакомы с регулярными выражениями, чтение этого порождающего правила может оказаться затруднительным, но на самом деле то, что в нем говорится, довольно просто: Литерал – это или последовательность любых символов, кроме двойных кавычек, заключенных в двойные кавычки, или последовательность любых символов, кроме одинарных кавычек, заключенных в одинарные кавычки.

Литералы являются лексемами. Внутри литералов допускается наличие пробельных символов, которые в этом случае имеют значение – они являются частью значения литерала. Требуется осторожность при использовании в

литерале символов табуляции, возврата каретки и перевода строки, потому что анализатор XML должен заменить их пробелами в процессе *нормализации значений атрибутов* еще до того, как их увидит анализатор выражений XPath. Для предотвращения этого используйте ссылки на символы, такие как «	», «
» и «
».

Определено в

XPath, раздел 3.7 (Lexical Structure), правило 29

Используется в

ПервичноеВыражение

КритерийУзла

Литералы используются в синтаксических правилах для КритерияУзла для проверки имени инструкции обработки: за дополнительной информацией об этой непоследовательности обращайтесь к разделу КритерийУзла на стр. 422.

Справка по использованию

Литерал представляет собой неизменное строковое значение (заметьте, что в отличие от некоторых других языков, числовые константы не считаются литералами). Он заключается в двойные или одинарные кавычки. Если используются одинарные кавычки, то внутри строки не должно быть одинарных кавычек, а если используются двойные кавычки, то внутри строки не должно быть двойных кавычек.

В литерале не могут одновременно находиться одинарные и двойные кавычки; если это все же необходимо, воспользуйтесь системной функцией `concat()`, описанной в главе 7. Использование символьных сущностей или ссылок на сущности не решает эту проблему, так как их значения подставляются анализатором XML до того, как они передаются анализатору XPath.

На практике способность использовать двойные и одинарные кавычки ограничивается еще и тем, что в контексте XSLT, выражения XPath всегда записываются в XML-атрибутах таблицы стилей. Если атрибут XML записывается с использованием одинарных кавычек, литерал XPath должен заключаться в двойные кавычки и не может содержать одинарную кавычку, потому что она завершает значение атрибута; по этой же причине, если атрибут записывается с использованием двойных кавычек, литерал XPath должен заключаться в одинарные кавычки и не может содержать двойную кавычку.

Лучший способ избежать этой проблемы – использовать вместо литералов переменные, значениями которых являются деревья. Например, вместо того, чтобы писать выражение «`message[text="I won't"]`», объявите переменную следующим образом:

```
<xsl:variable name="msg-text">I won't</xsl:variable>
```

а выражение запишите так: «message[text=$msg-text]».

Там, где выражение XPath целиком состоит из литерала, необходимы две пары кавычек:

```
<xsl:variable name="диалект-html" select="'netscape'"/>
```

Если не писать вторую пару кавычек, строка «netscape» будет воспринята как выражение, находящее все дочерние элементы <netscape> контекстного узла. Это означает, что вы не получите вразумительное сообщение об ошибке, если забудете написать вторую пару кавычек, наоборот, это скорее всего приведет к тому, что таблица стилей будет давать неверный результат. Следующий вариант записи более ясен:

```
<xsl:variable name="диалект-html">netscape</xsl:variable>
```

Хотя с технической точки зрения это не одно и то же (потому что в данном случае значением является временное дерево, а не строка), на практике везде вместо строки можно использовать дерево. (Единственным исключением является приведение к логическому типу.)

Примеры

Вот примеры допустимых в XPath литералов:

```
'Лондон'
"Лос-Анжелес"
"O'Коннор"
'Никогда не говори "никогда"'
```

Заметьте, что последние два примера вполне допустимы в XPath, но чтобы записать их в таблице стилей XSLT, понадобится использовать символьные ссылки или ссылки на сущности для одинарных или двойных кавычек, в зависимости от того, какой тип кавычек используется для ограничения значения XML-атрибута. Например:

```
<xsl:value-of select='"O&apos;Коннор"'/>
<xsl:value-of select="'Никогда не говори &quot;никогда&quot;'"/>
```

Вы не можете использовать символьные сущности или ссылки на сущности для кавычек, в которые заключается сам литерал, например:

```
<xsl:value-of select='"Никогда не говори &quot;никогда&quot;"'/> <!-- Ошибка -->
```

потому что после того, как анализатор XML разрешит ссылки на сущности, выражение XPath, полученное XSLT-процессором, будет выглядеть так:

```
"Никогда не говори "никогда""
```

а это очевидная ошибка. Обычно проще использовать переменные, которым присваиваются деревья, и функцию concat(), как это предлагалось выше.

МаршрутПоиска (LocationPath)

МаршрутПоиска является ВыражениемПути, выбирающим узлы, проходя по маршруту, начинающемуся либо с корневого узла (АбсолютныйМаршрутПоиска), либо с контекстного узла (ОтносительныйМаршрутПоиска).

Выражение	Синтаксис
МаршрутПоиска	ОтносительныйМаршрутПоиска | АбсолютныйМаршрутПоиска

Определено в

XPath, раздел 2, правило 1

Используется в

ВыражениеПути

Справка по использованию

Когда ОтносительныйМаршрутПоиска используется в качестве МаршрутаПоиска, он выбирает набор узлов, основываясь на их положении относительно контекстного узла.

АбсолютныйМаршрутПоиска используется для выбора набора узлов, основанного на их положении в исходном документе относительно корневого узла. (Если используется несколько исходных документов, это всегда будет корневой узел документа, в котором находится контекстный узел, так что слово «абсолютный» выбрано не совсем удачно.)

Примеры

Выражение	Описание
/контракт/пункт[3]/подпункт[2]	Абсолютный маршрут поиска выбирает второй <подпункт> третьего <пункт>а <контракт>а, являющегося элементом документа. Если элемент документа не <контракт> или если отсутствуют другие компоненты пути, результатом этого поиска будет пустой набор узлов.
//рисунок	Абсолютный маршрут поиска, выбирающий в документе все элементы <рисунок>. (О плохой производительности таких конструкций читайте в разделе АбсолютныйМаршрутПоиска.)
@заголовок	Этот относительный маршрут поиска выбирает все атрибуты «заголовок» контекстного узла. Результат может оказаться или пустым, или содержать один узел атрибута.

Выражение	Описание
книга/автор/имя	Данный относительный маршрут поиска выбирает дочерние элементы <имя> элементов <автор>, являющихся дочерними элементами элементов <книга>, которые, в свою очередь, являются дочерними элементами контекстного узла.
город[not(@название=preceding-sibling::город/@название)]	Этот относительный маршрут поиска выбирает все дочерние элементы <город> контекстного узла, у которых нет атрибута «название» с тем же значением, что и у атрибута «название» предыдущего одноуровневого элемента <город>. Таким образом, этот маршрут выбирает набор элементов <город> с несовпадающими названиями. Это выражение очень похоже на оператор SQL SELECT DISTINCT.

МультипликативноеВыражение (MultiplicativeExpr)

Нетривиальный вариант МультипликативногоВыражения выполняет операции умножения, деления или нахождения остатка от деления двух числовых операндов.

Определено в

XPath, раздел 3.5, правило 26

Синтаксис

Выражение	Синтаксис
МультипликативноеВыражение	УнарноеВыражение | МультипликативноеВыражение ОператорУмножения УнарноеВыражение | МультипликативноеВыражение «div» УнарноеВыражение | МультипликативноеВыражение «mod» УнарноеВыражение

Используется в

АддитивноеВыражение

Справка по использованию

Нетривиальный вариант МультипликативногоВыражения состоит из двух или более операндов УнарноеВыражение, разделенных операторами умножения «*», деления «div» или нахождения остатка от деления «mod», которые подробно описаны ниже. Тривиальный вариант МультипликативногоВыражения состоит из одного УнарногоВыражения без единого оператора.

Причина, по которой только для одного ОператораУмножения из всех операторов XPath определено собственное синтаксическое правило, заключается в

том, что значение символа «*» в XPath очень сильно зависит от контекста: кроме того, что он используется в качестве оператора умножения, он также является и символом подстановки в критериях узлов.

Символ «/» не используется для оператора деления, так как он уже используется в качестве оператора пути и выражения, представляющего корневой узел.

Хотя XPath не предназначен для выполнения расчетов, простые арифметические операции могут пригодиться, например, для вычисления положения объектов на выходном носителе.

Операнды `МультипликативногоВыражения`, не являющиеся числами, преобразуются в числа, как если бы для этого использовалась функция `number()`. Если значение не может быть преобразовано к обычному числу, оно преобразуется в специальное значение `NaN` (не-число), и в этом случае результатом вычислений также будет `NaN`.

В XSLT всегда используются числа с плавающей точкой двойной точности, поэтому вычисления производятся с использованием арифметики с плавающей точкой. Операции умножения и деления соответствуют правилам IEEE 754, приведенным в главе 2. Операция нахождения остатка от деления возвращает остаток от целочисленного деления, то есть деления не использующего дробную часть числа. Так как во многих случаях сложно понять, что это значит, в спецификации находятся полезные поясняющие примеры:

Выражение	Результат
5 mod 2	1
5 mod -2	1
-5 mod 2	-1
-5 mod -2	-1

Для целочисленного деления используйте выражение «`floor($X div $Y)`».

Арифметические операции в XPath никогда не вызывают ошибки. Например, деление на ноль – не ошибка, результатом этой операции будет бесконечность (или минус бесконечность).

Примеры

Выражение	Описание
`ceil(count(item) div 3)`	Результат округления вверх числа дочерних элементов <item> контекстного узла, деленного на три. (Например, это может быть выражение из таблицы стилей, располагающей элементы в три колонки.)
`@поле*2`	Удвоенное значение атрибута «поле» контекстного узла.
`item[position() mod 2 = 0]`	Выбирает четные дочерние элементы <item> контекстного узла.

НазваниеОси (AxisName)

Конструкция НазваниеОси используется в выражениях Шаг для задания пути следования от данного узла к другим родственным узлам.

Выражение	Синтаксис
НазваниеОси	«ancestor» | «ancestor-or-self» | «attribute» | «child» | «descendant» | «descendant-or-self» | «following» | «following-sibling» | «namespace» | «parent» | «preceding» | «preceding-sibling» | «self»

Определено в

XPath, раздел 2.2, правило 6

Используется в

СпецификаторОси

Справка по использованию

Ось (axis) – это путь через дерево документа, начинающийся с данного узла (называемого в этой книге началом отсчета) и следующий по узлам, связанным определенным отношением.

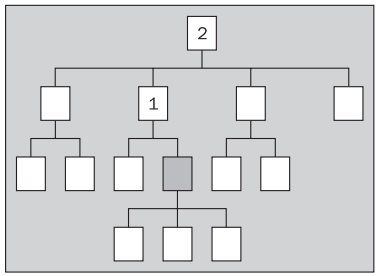
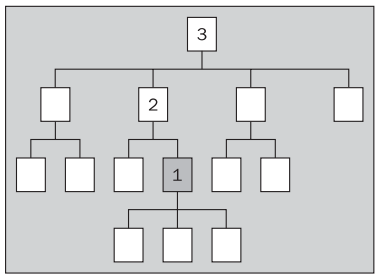
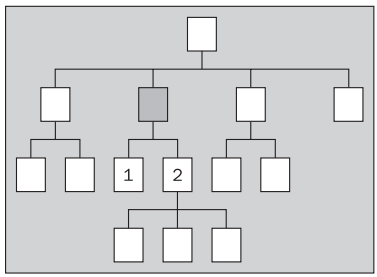
Различные названия осей означают следующее.

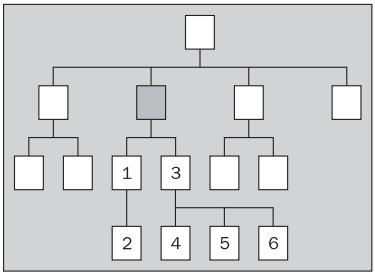
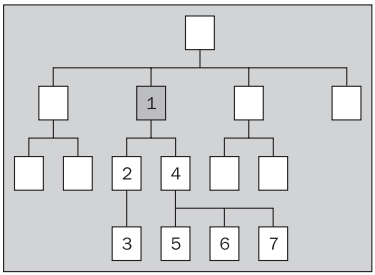
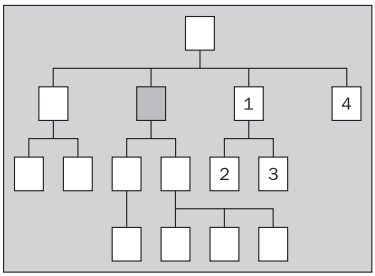
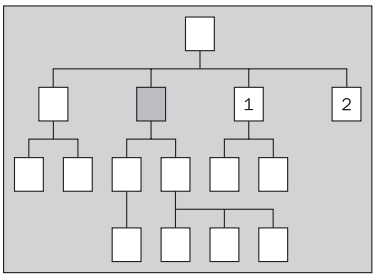
Ось	Описание
ancestor	Выбирает все узлы предков начального узла в порядке, обратном порядку документа. Первый узел на оси – родитель узла начала отсчета, второй – его дедушка, и так далее; последним узлом на этой оси является корень документа.
ancestor-or-self	Выбирает те же самые узлы, что и ось ancestor, но начиная с узла начала отсчета, а не его родителя.
attribute	Если узел начала отсчета – элемент, то эта ось выбирает все его узлы атрибутов, в произвольном порядке. В противном случае не выбирает ничего.

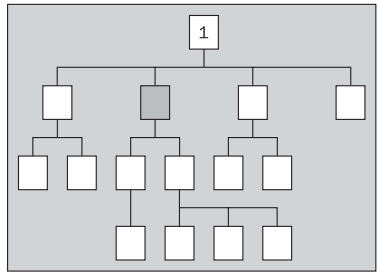
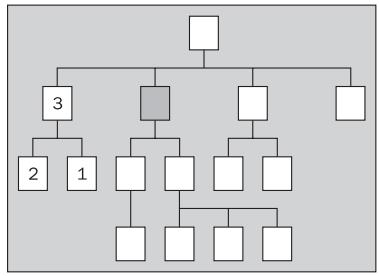
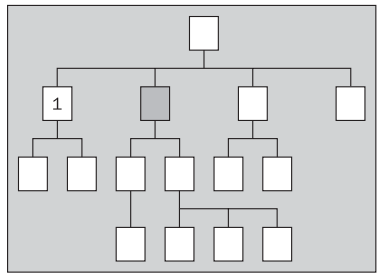
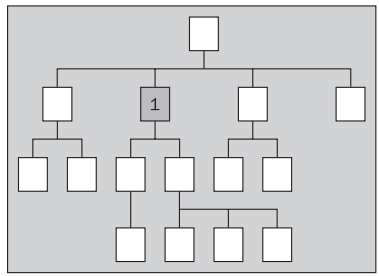
Ось	Описание
child	Выбирает все дочерние узлы начального узла в порядке документа. Для любого узла, кроме корневого или узла элемента, эта ось не выбирает ничего. Запомните, что в число дочерних узлов узла элемента не входят узлы его атрибутов или пространств имен, а только текстовые узлы, узлы элементов, инструкций обработки и комментариев, из которых состоит его содержимое.
descendant	Рекурсивно выбирает все дочерние узлы начала отсчета, их дочерние узлы и так далее. Узлы результата располагаются в порядке документа. Если началом отсчета является элемент, это означает, что ось descendant содержит все текстовые узлы, узлы элементов, комментариев и инструкций обработки, находящиеся в исходном документе между начальным и конечным тегами элемента, в их исходной последовательности.
descendant-or-self	То же самое, что и ось descendant, за исключением того, что первым выбирается узел начала отсчета.
following	Выбирает все узлы, находящиеся после узла начала отсчета, кроме его потомков, в порядке документа. Если началом отсчета является элемент, это означает, что ось содержит все текстовые узлы, узлы элементов, комментариев и инструкций обработки, расположенные в исходном документе сразу за закрывающим тегом элемента начала отсчета. На этой оси никогда не находятся узлы атрибутов или пространств имен.
following-sibling	Выбирает все узлы, следующие за узлом начала отсчета и являющиеся дочерними узлами его родительского узла, в порядке документа. Если началом отсчета является корневой узел, узел атрибута или пространства имен, то ось following-sibling будет всегда пустой.
namespace	Если узел начала отсчета – элемент, то эта ось выбирает все узлы пространств имен, находящиеся в области действия этого элемента; в противном случае она пуста. Порядок следования узлов пространств имен не определен. Узлы пространств имен соответствуют объявлениям пространств имен (`xmlns="x"` или `xmlns:y="z"`) самого элемента или одного из его предков, за исключением любых объявлений пространств имен, неприменимых к элементу, поскольку они скрыты другим объявлением с тем же префиксом пространства имен. За дополнительной информацией о пространствах имен, включая правила создания узлов пространств имен в конечном дереве, обращайтесь к главе 2.
parent	Эта ось выбирает единственный узел – родителя узла начала отсчета. Если узел начала отсчета – корневой, ось parent пуста.
preceding	Выбирает все узлы, находящиеся перед узлом начала отсчета, кроме его предков, в порядке, обратном порядку документа. Если началом отсчета является элемент, это означает, что ось содержит все текстовые узлы, узлы элементов, комментариев и инструкций обработки, заканчивающиеся перед начальным тегом эле-

Ось	Описание
preceding	мента начала отсчета. На этой оси никогда не находятся узлы атрибутов или пространств имен.
preceding-sibling	Выбирает все узлы перед узлом начала отсчета и являющиеся дочерними узлами его родительского узла в порядке, обратном порядку документа. Если началом отсчета является корневой узел, узел атрибута или пространства имен, то ось preceding-sibling будет всегда пустой.
self	Эта ось выбирает единственный узел – узел начала отсчета. Эта ось всегда не пуста.

Эти оси также можно изобразить схематично. На схемах для каждой оси нижеследующей таблицы узел начала отсчета выделен темным затенением, а узлы, находящиеся на оси, пронумерованы в порядке их появления на оси. На схемах не показаны узлы атрибутов и пространств имен, и поэтому осей атрибутов и пространств имен в таблице нет.

Ось	Схема
ancestor	
ancestor-or-self	
child	

Ось	Схема
descendant	
descendant-or-self	
following	
following-sibling	

Ось	Схема
parent	
preceding	
preceding-sibling	
self	

Дополнительную информацию об использовании осей в маршрутах поиска и примеры можно найти в описании выражения `Шаг` на стр. 465.

Примеры в контексте

Конструкция НазваниеОси всегда используется как часть Шага в маршруте поиска. Вот несколько примеров.

Выражение	Описание
ancestor::*	Выбирает элементы, являющиеся предками контекстного узла.
child::node()	Выбирает дочерние узлы контекстного узла. Обычно это выражение записывается в сокращенном виде: «node()».
attribute::цвет	Выбирает атрибут цвет контекстного узла. Обычно это выражение записывается в сокращенном виде: «@цвет». Если контекстный узел – не элемент, то результат будет пустым.
namespace::*	Выбирает все узлы пространств имен из области действия, в которой находится контекстный узел. Если контекстный узел – не элемент, то результат будет пустым.
self::item	Выбирает контекстный узел, если он является элементом <item>, в противном случае пустой набор узлов. Эта ось обычно используется в предикатах, например, выражение «*[not(self::item)]» выбирает все дочерние элементы контекстного узла, кроме элементов <item>. Работа этого выражения основывается на неявном преобразовании набора узлов в логическое значение, посредством которого пустой набор узлов преобразуется в ложь.

Оператор (Operator)

Оператор – это символ или имя, обозначающее операцию.

Выражение	Синтаксис
Оператор	ИмяОператора | ОператорУмножения | «/» | «//» | «|» | «+» | «–» | «=» | «!=» | «<» | «<=» | «>» | «>=»

Определено в

XPath, раздел 3.7 (Lexical Rules), правило 32

Используется в

Каждый оператор используется в отдельном порождающем правиле, например оператор «=» в правиле ВыражениеРавенства.

Справка по использованию

Порождающее правило для оператора является частью лексических правил XPath; оператор – это экземпляр конструкции ЭлементВыражения, то есть лексема. Так как операторы являются лексемами, то за и перед ними, но не внутри их, могут находиться пробельные символы.

В некоторых случаях требуется добавить пробельные символы до или перед оператором, чтобы анализатор мог гарантированно его распознать. Это относится не только к именованным операторам (таким как «and» или «or»), но также и к знаку «минус», который без предшествующего пробела может быть спутан с дефисом.

Операторы сравнения чисел записаны здесь в том виде, в каком они попадают к анализатору XPath; в таблице стилей XSLT специальные символы «<» и «>» могут записываться как «<» и «>» соответственно. Символы «/» и «//» считаются операторами, хотя у них есть одна отличительная черта: находящийся справа операнд является не выражением, а Шагом.

Примеры

```
and  or  div
*  /  =  !=  <
```

ОператорУмножения (MultiplyOperator)

ОператорУмножения – это знак умножения: звездочка.

Выражение	Синтаксис
ОператорУмножения	«*»

Причина, по которой только для одного ОператораУмножения из всех операторов XPath определено собственное порождающее правило, заключается в том, что значение символа «*» в XPath очень сильно зависит от контекста: кроме того, что он используется в качестве оператора умножения, он также является и символом подстановки в КритерииУзла.

Определено в

XPath, раздел 3.7 (Lexical Structure), правило 34

Используется в

МультипликативноеВыражение

Справка по использованию

См. раздел МультипликативноеВыражение на стр. 428.

Лексические правила XPath говорят, что символ «*» распознается как оператор умножения, если перед ним находится лексема и эта лексема не является одной из перечисленных: «@», «::», «,», «(», «[» или Оператор. В противном случае он считается частью КритерияИмени.

Примеры

Выражение	Описание
`item[$x * 2]`	Выбирает элемент <item>, положение которого равно удвоенному значению переменной $x.
`***`	Преобразует строковое значение первого дочернего элемента контекстного узла в число и умножает его на самого себя. (По приведенным выше правилам, только одна из трех звездочек воспринимается как оператор умножения. Остальные две являются сокращенной записью выражения «child::*».)

ОтносительныйМаршрутПоиска (RelativeLocationPath)

При использовании относительного маршрута поиска в качестве самостоятельного выражения он действует как маршрут поиска, выбирающий узлы путем прохода шагов, начиная с контекстного узла.

Относительный маршрут поиска также может использоваться в других конструкциях для представления последовательности шагов, начинающейся с какой-либо иной точки отсчета.

Выражение	Синтаксис
ОтносительныйМаршрутПоиска	Шаг | ОтносительныйМаршрутПоиска «/» Шаг | СокращенныйОтносительныйМаршрутПоиска

Определено в

XPath, раздел 2, правило 3

Используется в

МаршрутПоиска
АбсолютныйМаршрутПоиска
СокращенныйАбсолютныйМаршрутПоиска
ВыражениеПути
СокращенныйОтносительныйМаршрутПоиска

Справка по использованию

При использовании относительного маршрута поиска в качестве самостоятельного выражения он представляет собой последовательность шагов, начинающихся с контекстного узла. При использовании его в качестве части выражения пути он является последовательностью шагов, начинающейся с узлов из данного набора узлов, а при использовании в качестве части абсолютного маршрута поиска он является последовательностью шагов, начинающейся с корневого узла.

Относительный маршрут поиска состоит из одного или более шагов, разделенных оператором пути «/» или оператором сокращенного пути «//». В порождающем правиле используется рекурсия для определения этого, однако оно также может быть записано следующим образом:

```
ОтносительныйМаршрутПоиска   ⇒   Шаг ( ( «/» | «//» )  Шаг )*
```

Для простоты можно считать, что оператор «//» выбирает потомков узла. Он определяется в отдельном синтаксическом правиле, СокращенныйОтносительныйМаршрутПоиска, потому что на самом деле является сокращением. Точнее, если в относительном маршруте поиска одним из шагов (но не первым и не последним) является выражение «descendant-or-self::node()», то он может быть опущен. Например, выражение «A/descendant-or-self::node()/B» может быть заменено на «A//B». Подробности и примеры находятся в разделе СокращенныйОтносительныйМаршрутПоиска на стр. 450.

Описание работы оператора пути «/» находится в разделе Шаг на стр. 465.

Примеры

Выражение	Описание
ancestor::ГЛАВА	Это относительный маршрут поиска, состоящий из одного шага. Он выбирает предков контекстного узла, являющихся элементами с именем <ГЛАВА>.
ЗАГОЛОВОК	Это относительный маршрут поиска, состоящий из одного шага: на этот раз шаг является сокращенным шагом. Он выбирает дочерние узлы контекстного узла, являющиеся элементами с именем <ГЛАВА>.
descendant::АБЗАЦ/@стиль	Это относительный маршрут поиска, состоящий из двух шагов. Первый шаг выбирает потомков контекстного узла, являющихся элементами <АБЗАЦ>; второй шаг является сокращенным шагом, выбирающим атрибуты стиль этих элементов.
раздел[1]/пункт[3]	Это относительный маршрут поиска, состоящий из двух шагов, каждый из которых включает в себя позиционирующий предикат. Первый шаг выбирает первый элемент <раздел> из дочерних элементов контекстного узла; второй шаг выбирает третий элемент <пункт>, являющийся дочерним элементом выбранного элемента <раздел>.

Выражение	Описание
глава/раздел/абзац/предложение	Данный относительный маршрут поиска выбирает каждый элемент <предложение>, являющийся дочерним элементом <абзац>, который, в свою очередь, является дочерним элементом элемента <раздел>, являющегося дочерним элементом элемента <глава>, являющегося дочерним элементом контекстного узла.
.//предложение	Данный сокращенный относительный маршрут поиска выбирает каждый элемент <предложение>, являющийся потомком контекстного узла.

ПервичноеВыражение (PrimaryExpr)

ПервичноеВыражение – это, в сущности, выражение, в котором нет операторов. Оно также может быть заключенным в скобки подвыражением.

Выражение	Синтаксис
ПервичноеВыражение	СсылкаНаПеременную | «(» Выражение «)» | Литерал | Число | ВызовФункции

Определено в

XPath, раздел 3.1, правило 15

Используется в

ФильтрующееВыражение

Справка по использованию

Порождающее правило первичного выражения описывает несколько разных видов выражений, которые могут быть использованы в качестве строительных блоков для составления более сложного выражения.

У этих разных видов первичного выражения общим является только контекст, в котором они могут использоваться.

В соответствии с синтаксическими правилами, любое первичное выражение может заканчиваться предикатом для образования фильтрующего выражения, например выражения «17[1]» и «'Берлин'[3]» вполне допустимы с точки зрения синтаксиса. Однако семантические правила говорят, что предикат может применяться только к значению, являющемуся набором узлов, поэтому имеет смысл использовать в фильтрующем выражении предикаты только с такими первичными выражениями, как ссылка на переменную, выражение в скобках или вызов функции.

В данном порождающем правиле заметно отсутствие выражения пути: оно не является первичным выражением. Это гарантирует корректное распознавание выражения, подобного «абзац[1]», как выражения пути с предикатом «[1]», являющимся частью шага пути, а не как фильтрующего выражения, состоящего из первичного выражения «абзац», за которым следует предикат «[1]». Возможно превращение выражения пути в первичное выражение путем помещения его в скобки, например, выражение «(абзац)[1]» является фильтрующим выражением. В данном случае выражения означают одно и то же, но так бывает не всегда.

Например:

Выражение	Описание
ancestor::*[1]	Возвращает первого предка контекстного узла относительно оси ancestor, другими словами, родителя контекстного узла.
(ancestor::*)[1]	Возвращает первый узел (по порядку следования в документе) из набора узлов, образованного выражением «ancestor::*», то есть элемент документа.
//раздел/абзац[1]	Возвращает все элементы <абзац>, являющиеся первыми дочерними элементами <абзац> элемента <раздел>.
(//раздел/абзац)[1]	Возвращает первый элемент документа, являющийся дочерним элементом <абзац> элемента <раздел>.

Примеры

Выражение	Описание
23.5	Число является первичным выражением.
'Колумб'	Литерал является первичным выражением.
$var	Ссылка на переменную является первичным выражением.
contains(@название, '#')	Вызов функции является первичным выражением.
(position() + 1)	Выражение в скобках является первичным выражением.

ПолноеИмя (QName)

ПолноеИмя — это имя, которое может быть уточнено префиксом пространства имен.

Выражение	Синтаксис
ПолноеИмя	(Префикс «:»)? ЛокальнаяЧасть
Префикс	ИмяБезДвоеточия
ЛокальнаяЧасть	ИмяБезДвоеточия

Определено в

Спецификация XML Namespaces Recommendation

Используется в

КритерийИмени

Полные имена также используются во многих других контекстах таблиц стилей XSLT, вне выражений XPath. Они используются для ссылок на элементы из исходного документа (например, в элементах `<xsl:preserve-space>` и `<xsl:strip-space>`), а также для именования объектов в самой таблице стилей, включая переменные, шаблоны, режимы и наборы инструментов, и обращения к ним.

Существуют ситуации, когда полные имена могут создаваться динамически, в результате вычисления выражения. Эта возможность используется, например, в элементах `<xsl:element>` и `<xsl:attribute>`, чтобы создавать имена в конечном документе, а также в функциях `key()` и `format-number()` для обращения к объектам (ключам и десятичным форматам), определенным в таблице стилей. Динамически создаваемые полные имена никогда не используются для сопоставления с именами из исходного документа, а также для ссылок на шаблоны, переменные, режимы и наборы атрибутов таблицы стилей – все эти ссылки должны быть неизменными именами.

Справка по использованию

Полные имена используются в XPath для сопоставления с именами узлов исходного документа.

Если у имени есть префикс, то должно существовать объявление соответствующего ему пространства имен в каком-либо из окружающих его элементов таблицы стилей.

Например:

```
<xsl:apply-templates select="math:formula" xmlns:math="http://math.org/"/>
```

В этом примере пространство имен объявляется в элементе, использующем префикс, но это также может быть и любой предок этого элемента.

У фактического элемента в исходном документе необязательно должен быть тег «`math:formula`», можно использовать любой понравившийся префикс (или даже пространство имен по умолчанию), при условии, что имя элемента из исходного документа находится в пространстве имен с URI «`http://math.org`».

Если у полного имени нет префикса, то у совпадающего с ним имени должен быть пустой URI пространства имен. Это означает, что у элемента в исходном документе не должно быть никакого префикса, потому что в соответствии со спецификацией XML Namespaces не разрешается существование не-

пустых префиксов, связанных с пустым URI. Однако обратное неверно: имя без префикса из исходного документа может находиться в пространстве имен по умолчанию с непустым URI, и в этом случае для сопоставления с этим элементом необходимо использовать префикс в таблице стилей.

В выражении XPath полное имя без префикса использует пустой URI пространства имен, а не URI пространства имен по умолчанию.

Незнание этого факта является распространенной ошибкой. Представим, что исходный документ начинается так:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

а таблица стилей – так:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns="http://www.w3.org/1999/xhtml"
        version="1.0">

<xsl:template match="html">
```

Почему шаблонное правило «`match="html"`» не срабатывает, когда в документе встречается элемент `<html>`? Потому что пространство имен по умолчанию (объявленное при помощи атрибута «`xmlns="…"`») относится только к полным именам без префикса из исходного документа, но не к полным именам без префикса, находящихся в выражениях и образцах соответствия таблицы стилей. Фрагмент таблицы стилей необходимо переписать следующим образом:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xhtml="http://www.w3.org/1999/xhtml"
        version="1.0">

<xsl:template match="xhtml:html">
```

Примеры

Выражение	Описание
`ТАБЛИЦА`	Соответствует узлу в исходном документе с локальным именем `ТАБЛИЦА`, находящемуся в пространстве имен по умолчанию, при условии, что у пространства имен по умолчанию пустой URI.
`HTML:ТАБЛИЦА`	Соответствует узлу в исходном документе с локальным именем `ТАБЛИЦА`, находящемуся в пространстве имен, URI которого совпадает с URI пространства имен префикса `HTML` в таблице стилей.

Предикат (Predicate)

Предикат – это уточняющее выражение, служащее для выбора подмножества узлов из набора узлов или шага пути.

Выражение	Синтаксис
Предикат	«[» ПредикативноеВыражение «]»

Определено в

XPath, раздел 2.4, правило 8

Используется в

Шаг

ФильтрующееВыражение

Справка по использованию

Предикат может находиться в двух местах: как часть шага пути в выражении пути или как часть ФильтрующегоВыражения. В обеих ситуациях он имеет сходное назначение, и поэтому его легко в них использовать, не задумываясь о разнице между ними.

Например:

Выражение	Описание
абзац[3]	В данном примере предикат «[3]» применяется к шагу пути «абзац», который является сокращением для «./child::абзац». Он выбирает третий дочерний элемент <абзац> контекстного узла.
$абзац[3]	В данном примере предикат «[3]» применяется к ссылке на переменную «$абзац». В предположении, что переменная ссылается на набор узлов, данное выражение выбирает третий узел в этом наборе узлов, в порядке документа.

В обоих случаях предикат выбирает подмножество узлов из набора узлов. Разница появляется, когда предикат используется в выражении пути, состоящем из более чем одного шага. Например:

Выражение	Описание
глава/абзац[1]	В данном примере предикат «[1]» применяется к шагу пути «абзац», который является сокращением для «./child::абзац». Он выбирает первый элемент <абзац> каждого дочернего элемента <глава> контекстного узла.

Выражение	Описание
(глава/абзац)[1]	Это фильтрующее выражение, в котором предикат «[1]» применяется к набору узлов, выбранному выражением пути «глава/абзац». Оно выбирает единственный элемент <абзац>, первый дочерний элемент <абзац> элемента <глава>, являющегося дочерним элементом контекстного узла.

В действительности у оператора «[ ]» более высокий приоритет, чем у оператора пути «/». Другое различие между двумя случаями состоит в том, что в случае с фильтрующим выражением при вычислении предиката узлы рассматриваются в порядке документа. В случае с шагом пути, узлы рассматриваются в порядке соответствующей оси. Более подробно это обсуждается ниже.

Предикат может быть логическим выражением или числовым выражением. Синтаксически они не отличаются, так как XPath не является строго типизированным языком; например, предикат «[$p]» может относиться к любому типу из двух возможных. Отличие проявляется только во время выполнения. Если значением предиката является число, оно рассматривается как числовой предикат, в противном случае оно преобразуется в логический тип при помощи функции boolean() и рассматривается как логический предикат. Так, например, предикат «[@последовательный-номер]» истинен, если у контекстного узла есть атрибут последовательный-номер, и ложен в противном случае. Фактическое значение атрибута последовательный-номер не имеет значения: значением выражения «@последовательный-номер» является набор узлов, поэтому оно рассматривается как выражение «[boolean(@последовательный-номер)]». Если вы хотите использовать атрибут последовательный-номер в качестве числового предиката, перепишите его как «[number(@последовательный-номер)]».

Числовой предикат «[P]» — это просто сокращение для логического предиката «[position()=P]», так что можно достичь желаемого эффекта выражением «[position()=@последовательный-номер]».

Как было объяснено в главе 2, каждое выражение вычисляется в конкретном контексте. Контекст, в котором вычисляется предикат, отличается от контекста выражения, частью которого он является.

Предикат по очереди применяется к каждому узлу в наборе узлов. При каждом его вычислении:

- Контекстным узлом (узлом, выбираемым выражением «.») становится узел, к которому применяется предикат.
- Контекстной позицией (значение функции position()) становится позиция этого узла в наборе узлов *относительно какой-либо оси*.
- Размером контекста (значение функции last()) становится число узлов в наборе узлов.

Фраза «*относительно какой-либо оси*» означает, что позиция узла в шаге пути зависит от направления оси, используемой в этом шаге. Некоторые оси (child, descendant, descendant-or-self, following, following-sibling) направлены вперед, поэтому функция position() нумерует узлы в порядке документа.

Другие оси (ancestor, ancestor-or-self, preceding, preceding-sibling) направлены назад, поэтому функция position() нумерует узлы в порядке, обратном порядку документа. Оси self и parent возвращают только один узел, поэтому порядок не имеет значения. Направление осей attribute и namespace не определено.

В спецификации говорится, что контекстная позиция является позицией относительно оси child, когда предикат используется как часть фильтрующего выражения (а не частью шага пути). Это довольно туманное утверждение означает, что при вычислении значения функции position() в предикате узлы рассматриваются в порядке документа, независимо от порядка, в котором они были получены. Это не означает, что узлы должны быть дочерними узлами какого-нибудь общего родителя или вообще чьими-нибудь дочерними узлами. Вполне допустимо следующее выражение:

```
(document('a.xml') | document('b.xml'))[1]
```

В этом примере образуется набор узлов, состоящий из двух корневых узлов разных документов, затем он фильтруется, чтобы получить узел, идущий первым «в порядке документа». Когда узлы набора узлов взяты из разных документов, порядок следования не определен, поэтому нельзя предугадать, какой из двух корневых узлов будет выбран. Тем не менее, порядок стабилен и не меняется в отдельном процессе трансформации.

В обоих контекстах, где может использоваться предикат, может использоваться и последовательность из нуля или более предикатов. Любопытно, что определение синтаксиса отличается в этих двух случаях: в порождающем правиле для шага пути он определяется итеративно (Шаг ⇒ СпецификаторОси КритерийУзла Предикат*), а в правиле для фильтрующего выражения – рекурсивно (ФильтрующееВыражение ⇒ ФильтрующееВыражение Предикат). Однако результат одинаков: может использоваться последовательность из нуля или более предикатов.

Задание двух отдельных предикатов отличается от объединения двух предикатов в один с помощью оператора «and». Причина этого в том, что контекст второго предиката отличается от контекста первого. В частности, во втором предикате контекстная позиция (значение функции position()) и размер контекста (значение функции last()) зависят только от узлов, прошедших через предыдущий предикат. В следующих примерах показано, что это означает на практике.

Выражение	Описание
книга[автор="Ф. Д. Джеймс"][1]	Первая книга, если ее автор – Ф. Д. Джеймс.
книга[1][автор="Ф. Д. Джеймс"]	Первая книга, в том случае, если ее автор – Ф. Д. Джеймс.
книга[position()=1 and автор="Ф. Д. Джеймс"]	Первая книга, в том случае, если ее автор – Ф. Д. Джеймс. Действие примера совпадает с предыдущим, потому что второй предикат не зависит от контекстной позиции.

Примеры в контексте

Выражение	Описание
`абзац[1]`	Первый дочерний элемент <абзац> контекстного узла.
`абзац[last()]`	Последний дочерний элемент <абзац> контекстного узла
`абзац[position()!=1]`	Все дочерние элементы <абзац> контекстного узла, кроме первого.
`абзац[@заголовок]`	Все дочерние элементы <абзац> контекстного узла, у которых есть атрибут заголовок.
`абзац[string(@заголовок}]`	Все дочерние элементы <абзац> контекстного узла, у которых есть атрибут заголовок, не являющийся пустой строкой.
`абзац[* or text()]`	Все дочерние элементы <абзац> контекстного узла, являющиеся родителями узла элемента или текстового узла.

ПредикативноеВыражение (PredicateExpr)

`ПредикативноеВыражение` — это выражение, находящееся внутри предиката.

Выражение	Синтаксис
`ПредикативноеВыражение`	`Выражение`

Определено в

XPath, раздел 2.4, правило 9

Используется в

`Предикат`

Справка по использованию

Синтаксически `ПредикативноеВыражение` — это обычное `Выражение`: другими словами, в предикате может использоваться любое выражение XPath.

Если результатом вычисления выражения предиката является число, он рассматривается как числовой предикат, истинный, если его значение совпадает с контекстной позицией, и ложный в противном случае.

Во всех остальных случаях значение выражения предиката рассматривается как значение логического типа и при необходимости преобразуется к логическому типу при помощи функции `boolean()`.

Заметьте, что правила, по которым предикат рассматривается как числовой, очень строги. Например, XSLT-переменная, объявленная в нижеследующих примерах, является не числом, а набором узлов (в XSLT 1.0 фрагментом ко-

нечного дерева, подробное объяснение находится в разделе <xsl:variable> главы 4):

```
<xsl:variable name="номер">3</xsl:variable>
```

Если вы хотите использовать ее значение в качестве предиката, или перепишите объявление так, чтобы значением переменной стало число:

```
<xsl:variable name="номер" select="3"/>
```

(но не пишите «select="'3'"», потому что в этом случае значением станет строка); или используйте явное приведение к числовому типу в предикате:

```
<xsl:value-of select="элемент[number($номер)]"/>
```

или напишите логический предикат целиком:

```
<xsl:value-of select="элемент[position()=$номер]"/>
```

Примеры в контексте

Любое выражение может использоваться в качестве предиката (кроме выражения, значением которого является внешний объект, потому что внешний объект нельзя преобразовать в логическое значение). Вот несколько примеров; выражение предиката находится между квадратными скобками:

Выражение	Описание
раздел[заголовок]	«заголовок» — это выражение пути; выражение предиката истинно в том случае, если у элемента <раздел> есть хотя бы один дочерний элемент <заголовок>.
раздел[@заголовок='Введение']	Здесь предикатом является более привычное логическое выражение.
заголовок[substring-before(.,':')]	Выражение предиката возвращает истину, если в строковом значении элемента заголовок есть один или более символов перед первым двоеточием, то есть, если функция substring-before() возвращает непустую строку.
книга[not(автор= preceding-sibling ::автор)]	Здесь значением выражения предиката является истина, если автор книги не тот же самый, что и автор какой-либо предыдущей книги из того же самого родительского элемента. Это выражение выбирает первую книгу каждого автора.

ПробельныеСимволыВыражения (ExprWhitespace)

Правило ПробельныеСимволыВыражения точно определяет, что такое пробельные символы в выражении XPath.

Выражение	Синтаксис
ПробельныеСимволыВыражения	S
S	(#x20 | #x9 | #xD | #xA)+

Определено в

XPath, раздел 3.7, правило 39

Правило для S взято из XML Recommendation.

Используется в

На это порождающее правило нет ссылок из других правил, оно используется только в пояснительном тексте спецификации XPath.

Справка по использованию

Определение пробельных символов взято напрямую из стандарта XML: пробельные символы – это любая последовательность пробелов, символов табуляции, символов возврата каретки и перевода строки.

XPath позволяет использовать пробельные символы до или после любой лексемы ЭлементВыражения. Список конструкций, которые считаются лексемами и поэтому не могут включать в себя пробельные символы, находится выше в разделе ЭлементВыражения. Не всегда интуитивно понятно, какие сочетания символов в XPath являются лексемами, а какие – нет. Например, выражение «self::x» состоит из трех лексем (поэтому можно вставлять пробелы до или после двойного двоеточия, но не внутри ег о), а выражение «xsl:*» состоит из одной лексемы, поэтому нигде в нем нельзя вставить пробельные символы. Это следует из того, что выражение «xsl:*» является КритериемИмени, а он, в свою очередь, является одной из конструкций, перечисленных в порождающем правиле для ЭлементаВыражения.

Лексические правила для XPath гласят, что любая лексема поглощает максимально возможное количество символов. Если символ может рассматриваться как часть предыдущей лексемы, то именно так он и рассматривается, даже если в дальнейшем это вызывает синтаксическую ошибку. Это означает, что в некоторых ситуациях необходимо вставлять пробельные символы, чтобы разделить лексемы; наиболее частым примером этого является необходимость вставлять пробел перед знаком «минус», чтобы отличать арифметическое выражение вида «цена - скидка» от имени с дефисом «цена-скидка». Пробельные символы нужны также для того, чтобы отделять имена от операторов «or», «and», «div» и «mod». В выражении «5 mod 2» пробел перед оператором «mod» необязателен, потому что строка «5mod» не является допустимой лексемой, в то время как пробел после оператора «mod» обязателен, потому что в противном случае строка «mod2» будет воспринята как лексема ПолноеИмя.

Хотя принято писать выражения XPath на одной строке, потому что именно так обычно пишутся атрибуты XML, ничто не мешает использовать вместо пробелов символы табуляции и перевода строки.

Примеры

Выражение	Описание
	В левом столбце показано, как обычно выглядят пробельные символы.
Если вы действительно хотите видеть пробельные символы, то можете записывать их таким образом.

СокращенныйАбсолютныйМаршрутПоиска (AbbreviatedAbsoluteLocationPath)

СокращенныйАбсолютныйМаршрутПоиска — это выражение для выбора всех узлов документа, удовлетворяющих какому-либо условию.

Выражение	Синтаксис
СокращенныйАбсолютныйМаршрутПоиска	«//» ОтносительныйМаршрутПоиска

Определено в

XPath, раздел 2.5, правило 10

Используется в

АбсолютныйМаршрутПоиска

Справка по использованию

Начальные символы «//» указывают на то, что путь выборки начинается с корня документа; затем относительный маршрут поиска (relative location path) указывает продолжение выборки.

Вычисление этой формы выражений может оказаться ресурсоемкой операцией, потому что в общем случае XSLT-процессор должен выполнить поиск по всему документу для того, чтобы найти выбранные узлы. Если вы можете задать более узкий критерий поиска, то лучше использовать именно его. Например, если известно, что все элементы <книга> являются непосредственными потомками элемента документа, то выражение «/*/книга» в общем случае будет более эффективным, чем «//книга». Конечно же, фактические характеристики производительности разных продуктов могут отличаться.

Примеры

Выражение	Описание
//рисунок	Выбирает в документе все элементы <рисунок>.
//книга[@категория='беллетристика']	Выбирает в документе все элементы <книга>, имеющие атрибут категория со значением «беллетристика».
//*/*	Выбирает все узлы элементов, имеющие в качестве родителя элемент, иными словами, все элементы, кроме непосредственных потомков корневого узла. Здесь «*» – это критерий узла КритерийИмени, соответствующий любому элементу.
//книга/название	Выбирает все дочерние элементы <название> элемента <книга>.

Формально «//X» – сокращение для «/descendant-or-self::node()/X». Так что выражение «//рисунок[1]» означает «/descendant-or-self::node()/рисунок[1]», то есть каждый элемент <рисунок> документа, являющийся первым дочерним элементом <рисунок> своего родительского элемента. Если вы хотите выбрать первый элемент <рисунок> в документе, используйте выражение «/descendant::рисунок[1]» или «(//рисунок)[1]».

СокращенныйОтносительныйМаршрутПоиска (AbbreviatedRelativeLocationPath)

СокращенныйОтносительныйМаршрутПоиска – это относительный маршрут поиска, в котором используется оператор «//», являющийся простым способом запроса всех потомков узла, а не только дочерних узлов.

Выражение	Синтаксис
СокращенныйОтносительныйМаршрутПоиска	ОтносительныйМаршрутПоиска «//» Шаг

Определено в

XPath, раздел 2.5, правило 11

Используется в

ОтносительныйМаршрутПоиска

Справка по использованию

Сокращенный относительный маршрут поиска – это сокращенная запись выборки потомков узла.

Так же как и оператор «//», используемый в начале пути, эта конструкция может оказаться ресурсоемкой, потому что XSLT-процессор должен выпол-

нить поиск среди всех потомков узла. Если вы можете задать более узкий критерий поиска, то лучше использовать именно его. Например, если вы знаете, что все искомые узлы являются правнуками начального узла, то лучше написать «$A/*/B» вместо «$A//B». Конечно же, фактические характеристики производительности разных продуктов могут отличаться.

Формально такое выражение, как «$главы//график», является сокращением выражения «$главы/descendant-or-self::node()/график». Это означает: найти для каждого элемента в наборе узлов «$главы» все его узлы-потомки и среди них найти дочерние узлы, являющиеся элементами <график>. С первого взгляда кажется, что происходит то же самое, что и при вычислении выражения «$главы/descendant::график», но есть тонкое различие, касающееся использования позиционных предикатов, как в нижеследующем примере:

Пример: Сравнение оператора «//» с «/descendant::»

Рассмотрим выражения «$главы//график[1]» и «$главы/descendant::график[1]»:

- «$главы//график[1]» означает «$главы/descendant-or-self::node()/график[1]», а именно каждый элемент <график>, который является первым дочерним элементом <график> своего родителя и потомком узла из $главы.
- «$главы/descendant::график[1]» означает первый элемент <график> (в порядке следования элементов в документе), являющийся потомком узла из $главы. Этот же результат можно получить и другим образом: «($главы//график)[1]».

Чтобы понять разницу, взгляните на следующий исходный документ:

```
<глава>
  <раздел>
    <график номер="12"/>
    <график номер="13"/>
  </раздел>
  <график номер="14"/>
  <раздел>
    <график номер="15"/>
    <график номер="16"/>
  </раздел>
</глава>
```

Если набор узлов $главы содержит только наружный элемент <глава> данного документа, то выражение «$главы//график[1]» выберет диаграммы 12, 14 и 15, тогда как выражения «$главы/descendant::график[1]» и «($главы//график)[1]» выберут только диаграмму 12.

Примеры

Выражение	Описание
`глава//сноска`	Выбирает все элементы <сноска>, являющиеся потомками элемента <глава>, который, в свою очередь, является дочерним узлом контекстного узла. Объяснение того, что такое контекстный узел, вы можете найти в описании конструкции Выражение на стр. 394.
`.//сноска`	Выбирает все элементы <сноска>, являющиеся потомками контекстного узла.
`document('lookup.xml')//запись`	Выбирает все элементы <запись> в документе, заданном относительным URL `lookup.xml`. Функция `document()` описана в главе 7.
`$победители//*/@имя`	Выбирает атрибут `имя` всех элементов, являющихся потомками узла, принадлежащего набору узлов заданному переменной `$победители`.
`.//..`	В этом странном, но вполне допустимом выражении, комбинируется оператор «//», отыскивающий потомков узла, и оператор «..», отыскивающий его родителя. В результате выполняется поиск всех узлов, являющихся родителями потомков контекстного узла, а также родителя контекстного узла.

СокращенныйСпецификаторОси (AbbreviatedAxisSpecifier)

СокращенныйСпецификаторОси указывает, что ось для выбора узлов в выражении пути является либо осью дочерних узлов, либо осью атрибутов.

Выражение	Синтаксис
СокращенныйСпецификаторОси	«@» ?

Обратите внимание на знак вопроса: он означает, что «@» необязателен. Другими словами, СокращенныйСпецификаторОси может быть совершенно пустой строкой.

Определено в

XPath, раздел 2.5, правило 13

Используется в

СпецификаторОси

Справка по использованию

Сокращенным спецификатором оси может быть либо символ «@», указывающий на ось атрибутов, либо отсутствие какого-либо символа, указывающее на ось дочерних узлов. Сокращенный спецификатор оси «@» – это сокращение для «attribute::», а пустой сокращенный спецификатор оси «» – сокращение для «child::». Взглянув на синтаксис СпецификатораОси на стр. 455, мы увидим, что он сам может быть пустым, а это означает (если посмотреть на конструкцию Шаг на стр. 465), что Шаг может состоять как из КритерияУзла самого по себе, так и с одним или более последующими Предикатами.

На практике это означает, что в выражении пути «A/@B» B относится к атрибуту A, тогда как в выражении пути «A/B» B относится к дочернему элементу A.

> ***Будьте осторожны при использовании оси*** «self::»***. Вы можете написать*** «self::название»***, чтобы проверить, является ли контекстный узел элементом*** <название>***, но вы не можете написать*** «self::@название»***, чтобы проверить, является ли он атрибутом*** название**.**

Примеры в контексте

Выражение	Описание
@категория	Это ОтносительныйМаршрутПоиска, состоящий из одного Шага, состоящего, в свою очередь, из СокращенногоСпецификатораОси «@», за которым следует один из КритериевУзла – КритерийИмени или, более конкретно, ПолноеИмя. Это выражение выбирает любой узел атрибута контекстного узла с именем «категория». В развернутом виде это выражение записывается так: «./attribute::категория».
название	Это ОтносительныйМаршрутПоиска, состоящий из одного Шага, состоящего, в свою очередь, из пустого СокращенногоСпецификатораОси, за которым следует один из КритериевУзла - КритерийИмени или, более конкретно, ПолноеИмя. В развернутом виде это выражение записывается так: «./child::название».

СокращенныйШаг (AbbreviatedStep)

СокращенныйШаг – это упрощенный способ выбора контекстного узла или родителя контекстного узла.

Выражение	Синтаксис
СокращенныйШаг	«.» | «..»

Определено в

XPath, раздел 2.5, правило 12

Используется в

Шаг

Справка по использованию

Порождающее правило СокращенныйШаг определяет два распространенных символа: «.», ссылающийся на контекстный узел, и «..», ссылающийся на родителя контекстного узла. (Обсуждение понятия контекстный узел см. в описании конструкции Выражение на стр. 394.) Эти символы являются сокращениями для «self::node()» и «parent::node()», соответственно.

Хотя формально «.» – Шаг, и, следовательно, может использоваться справа от оператора пути «/», поступать так не имеет смысла, так как результатом будет пустой, никуда не ведущий шаг. Символ «.» обычно используется в следующих двух случаях:

- С оператором «//» в относительном выражении пути, таком как «.//A», выбирающем (грубо говоря) все элементы <A>, являющиеся потомками контекстного узла. Точка «.» необходима здесь, так как если бы выражение начиналось с оператора «//», то оно выбрало бы всех потомков корневого узла.
- Сам по себе, для обозначения набора узлов, состоящего только из контекстного узла. Необходимость в этом появляется в таких выражениях, как «.=3» или «string-length(.)», в которых мы хотим проверить значение контекстного узла, или в инструкции XSLT <xsl:value-of select=".”/>, которая выдает строковое значение контекстного узла в результирующее дерево.

Некоторые разработчики предпочитают для ясности использовать оператор «.» в начале относительных выражений пути, таких как «./НАЗВАНИЕ», но на самом деле оно совершенно эквивалентно выражению «НАЗВАНИЕ».

Использование «..» для ссылки на родительский узел также часто встречается в начале относительных выражений пути, например, «../@название» выбирает атрибут название родителя контекстного узла. Оператор «..» можно использовать в любом месте выражения пути, однако необходимость в этом возникает редко. Например, выражение «//@ширина/..» выбирает все элементы документа, имеющие атрибут ширина. Тот же результат может быть достигнут использованием более очевидного выражения «//*[@ширина]».

Заметьте, что у каждого узла, кроме корневого, есть родитель (из этого следует, что выражение «/..» всегда возвращает пустой набор узлов, а выражение «not(..)» – простой способ проверки того, является ли контекстный узел корневым). Как было объяснено в главе 2, элемент, содержащий атрибут, считается родителем атрибута, хотя атрибут не является дочерним узлом элемента. Поэтому можно выбрать все элементы, содержащие атрибут ID, с помощью такого выражения как «//@ID/..» (хотя выражение «//*[@ID]» делает то же самое, но, возможно, более эффективно). В отличие от биологических отношений, в XPath отношения между «родителем» и «ребенком» не

обратимы по отношению друг к другу. То же самое относится и к узлам пространств имен.

Обратите внимание, что непосредственно за операторами «.» и «..» не может следовать предикат, например, нельзя написать выражение <xsl:if test=".[@цвет='черный']">. Если вы использовали диалект XSL Microsoft WD-xsl, не поддерживающий логические выражения, то могли привыкнуть к использованию подобных конструкций. В XSLT вы можете просто написать <xsl:if test="@цвет='черный'">.

Примеры в контексте

Инструкция XSLT	Описание
<xsl:value-of select="."/>	Выводит строковое значение контекстного узла.
<xsl:value-of select="../@название"/>	Выводит значение атрибута название родителя контекстного узла.

СпецификаторОси (AxisSpecifier)

СпецификаторОси – это либо название оси, либо сокращенное название оси.

Выражение	Синтаксис
СпецификаторОси	НазваниеОси «::» | СокращенныйСпецификаторОси

Определено в

XPath, раздел 2.1, правило 5

Используется в

Шаг

Справка по использованию

СпецификаторОси – это либо название оси, за которым следует двойное двоеточие, либо СокращенныйСпецификаторОси. Так как СокращенныйСпецификаторОси может быть либо символом «@», либо пустым, соответственно и СпецификаторОси может быть пустым, отражая тот факт, что в данном контексте выражение «АБЗАЦ» – сокращение для выражения «child::АБЗАЦ».

СпецификаторОси определяет направление движения через документ. Для данного начального узла документа СпецификаторОси определяет упорядоченный список узлов, которые можно проходить по очереди. Дополнительная информация о каждой оси находится в разделе НазваниеОси на стр. 401.

Примеры

Выражение	Описание
`ancestor::`	Задает ось ancestor.
`preceding-sibling::`	Задает ось preceding-sibling.
`@`	Задает ось attribute.
	Задает ось child. (Да-да, левая ячейка таблицы пустая!)

СсылкаНаПеременную (VariableReference)

`СсылкаНаПеременную` – это ссылка на переменную или параметр XSLT.

Выражение	Синтаксис
`СсылкаНаПеременную`	«$» `ПолноеИмя`

Определено в

XPath, раздел 3.7 (Lexical Rules), правило 36

Используется в

`ПервичноеВыражение`

Справка по использованию

`ПолноеИмя` должно совпадать с именем переменной или параметра, находящихся в области действия того места в таблице стилей, где находится выражение, в котором встречается имя переменной. Обычно это означает, что имя будет совпадать с атрибутом `name` соответствующего элемента `<xsl:variable>` или `<xsl:param>`. Однако если имя с префиксом, то должны совпадать URI пространств имен, а префиксы – необязательно.

Ссылка на переменную является лексемой, поэтому между знаком $ и полным именем не могут находиться пробельные символы.

Значение ссылки на переменную – это то, что было ей присвоено в соответствующем объявлении `<xsl:variable>` или `<xsl:param>` (в случае с `<xsl:param>`, значение также может быть получено из элемента `<xsl:with-param>` вызывающего шаблона). Значение может быть любого типа: логического, числового, строкового, набором узлов или, в случае XSLT 1.0, фрагментом конечного дерева. Это также может быть внешний объект, возвращенный функцией расширения. При необходимости значение преобразуется в тип данных, требуемый в данном контексте: например, если переменная, значением которой является набор узлов, встречается в контексте, где необходимо значение логического типа (таком как атрибут `test` элемента `<xsl:if>`), ее значение

приводится к логическому типу; то есть, ее значение будет считаться истиной, если в наборе узлов есть один или более узлов, или ложью в противном случае. Однако в некоторых случаях приведение к другому типу невозможно, и в этом случае возникнет ошибка времени выполнения. Например, нельзя использовать значение логического типа там, где требуется набор узлов, например, в атрибуте `select` элементов `<xsl:apply-templates>` или `<xsl:for-each>`.

Ссылка на переменную может использоваться буквально в любом месте выражения XPath, где требуется значение: то есть экземпляр одного из пяти типов данных: логического, числового, строкового, набор узлов или внешний объект. Она не может использоваться для представления других концепций языка, например, нельзя использовать переменную вместо имени, типа узла или оси. Также нельзя использовать переменную для хранения самого выражения.

Часто встречаются ошибки, подобные следующей:

```
<xsl:sort select="$выражение"/>
```

где в переменной `$выражение` ***находится выражение XPath, представленное в виде строки. Это не будет работать, потому что ключом сортировки будет само выражение, а не его значение, и, конечно же, его значение будет одним и тем же для каждого узла, так что на самом деле сортировка не будет происходить вообще.***

При условии, что выражение – это не более, чем имя элемента, вы можете достичь желаемого эффекта, написав:

```
<xsl:sort select="*[name()=$имя-элемента]"/>
```

Если нужны более сложные ключи сортировки, можно добиться желаемого эффекта с помощью функции расширения `evaluate()`***, предоставляемой несколькими процессорами (включая Saxon и Xalan). Эта функция позволяет создать выражение XPath из строки во время выполнения. В качестве альтернативного варианта можно изменять таблицу стилей перед ее выполнением, возможно, применяя к ней самой XSLT-трансформацию или напрямую используя интерфейсы DOM. В приложении A находится пример, демонстрирующий, как это делается в процессоре MSXML3.***

Примеры

```
$x
$наименьшее-общее-кратное
$ALPHA
$my-ns-prefix:param1
$p
```

ТипУзла (NodeType)

Конструкция ТипУзла представляет ограничение на тип узла.

Выражение	Синтаксис
ТипУзла	«comment» | «text» | «processing-instruction» | «node»

Конструкция ТипУзла является лексемой, поэтому может содержать пробельные символы до и после имени, но не внутри него.

Заметьте, что четыре имени типов узлов не могут использоваться в качестве имен функций, но несмотря на это они не являются зарезервированными словами. Ничто не мешает называть элементы или атрибуты в XML-документе именем «text» или «node», поэтому в XPath допустимы обычные имена «text» или «node». Вот почему имена помечаются в критерии узла скобками, например «text()».

Определено в

XPath, раздел 3.7 (Lexical Rules), правило 38

Используется в

КритерийУзла

Справка по использованию

ТипУзла может использоваться в критерии узла (который, в свою очередь, используется в Шаге), чтобы заставить шаг возвращать узлы нужного типа. Значение ключевых слов «comment», «text» и «processing-instruction» очевидно: они ограничивают выбор узлов данным типом. Ключевое слово «node» выбирает узлы любого типа. Это полезная возможность, так как Шаг должен включать в себя какой-либо критерий узла, так что если вам нужны все узлы на оси, можно просто задать тип «node()». Например, если нужны все дочерние узлы, задайте «child::node()».

Заметьте, что не существует способа обращения к четырем другим типам узлов, а именно: корневому узлу, узлу элемента, атрибута и пространства имен. Если нужен только лишь корневой узел, то нет необходимости его поиска с применением оси, достаточно использовать специальное выражение «/». Отсутствие типов узлов для атрибутов и пространств имен объясняется тем, что для данных типов существуют собственные оси attribute и namespace: эти узлы можно найти только с помощью соответствующей оси, и все узлы на ней будут узлами требуемого типа. Что касается узлов элементов, то у всех осей, на которых они могут находиться, основным типом узла является

узел элемента, а узлы основного типа можно выбирать специальным критерием имени «*».

Примеры в контексте

Выражение	Описание
`parent::node()`	Выбирает родителя контекстного узла, будь это узел элемента или корневой узел. Данное выражение отличается от выражения «parent::*», выбирающего родительский узел лишь в том случае, если он элемент. Выражение «parent::node()» обычно сокращается до «..».
`//comment()`	Выбирает все узлы комментариев в документе.
`child::text()`	Выбирает дочерние текстовые узлы контекстного узла. Обычно сокращается до «text()».
`@comment()`	Странный, но допустимый способ получения пустого набора узлов: ищутся все узлы комментариев на оси attribute, но, конечно, не находится ни один.

УнарноеВыражение (UnaryExpr)

Нетривиальный вариант УнарногоВыражения используется для изменения знака числа.

Выражение	Синтаксис
УнарноеВыражение	ВыражениеОбъединения | «–» УнарноеВыражение

Определено в

XPath, раздел 3.5, правило 27

Используется в

МультипликативноеВыражение

Справка по использованию

Унарное выражение состоит из ВыраженияОбъединения с необязательным предшествующим знаком «минус». На самом деле, выражению объединения может предшествовать любое число (ноль или более) знаков «минус», каждый из которых меняет знак числа.

Выражение «-0» обозначает отрицательный ноль – концепцию, определенную в модели арифметики с плавающей точкой IEEE 754. Почти во всех отношениях он ведет себя как положительный ноль (например, он равен положительному нулю и выводится как «0»). Одно из отличий состоит в том, что

если вы поделите на него +1, то получите минус бесконечность, а не плюс бесконечность: но на самом деле сложно найти практическое применение этого факта в таблице стилей.

Может показаться странной возможность применения арифметического оператора (унарный минус) к объединению, являющемуся набором узлов. Однако это всего лишь побочный эффект способа определения приоритета операторов. В ВыраженииОбъединения не обязательно должен присутствовать оператор объединения, существует лишь возможность его присутствия, равно как и в унарном выражении не требуется наличие знака «минус».

На самом деле можно, если не очень полезно, написать такое выражение, как «-$a|$b». Это выражение создает набор узлов, являющийся объединением переменных $a и $b, преобразует его в число, а затем меняет его знак на противоположный. Набор узлов преобразуется в число путем преобразования в число строкового значения первого узла набора (в порядке документа).

Примеры

Выражение	Описание
–2	Числовое значение «минус два». Знак «минус» не является частью числа – это самостоятельная лексема (и, следовательно, может отделяться от числа пробельными символами).
– @кредит	Значение атрибута кредит контекстного узла элемента со знаком, измененным на противоположный. Если у контекстного узла нет атрибута кредит или если его значение не является числом, результатом выражения будет не-число.
1 - - - 1	Не очень полезный, но вполне допустимый способ написания значения 0. Первый знак «минус» является оператором вычитания, остальные два – унарными знаками «минус».

ФильтрующееВыражение (FilterExpr)

ФильтрующееВыражение используется для применения одного или более Предикатов к набору узлов, выбирая подмножество узлов, удовлетворяющих какому-либо условию.

Выражение	Синтаксис
ФильтрующееВыражение	ПервичноеВыражение | ФильтрующееВыражение Предикат

Определено в

XPath, раздел 3.3, правило 20

Используется в

ВыражениеПути

Справка по использованию

Нетривиальный вариант ФильтрующегоВыражения состоит из ПервичногоВыражения, значением которого является набор узлов, за которым следует один или более Предикат, выбирающий подмножество набора узлов. Каждый предикат состоит из выражения, заключенного в квадратные скобки, например «[@название='Лондон']» или «[position()=1]».

Из-за применяемого в XPath способа определения синтаксиса каждое ПервичноеВыражение также является тривиальным вариантом ФильтрующегоВыражения, включая такие простые конструкции, как «23», «'Вашингтон'» и «true()».

Если используется хотя бы один предикат, значением ПервичногоВыражения должен быть набор узлов. Каждый предикат по очереди применяется к набору узлов; только те узлы, для которых предикат вернул истину, остаются к следующему шагу. Конечный результат состоит из тех узлов исходного набора узлов, которые удовлетворяют условию каждого предиката.

Предикат может быть числовым (например, «[1]» или «[last()-1]») или логическим (например, «[count(*) > 5]» или «[@имя and @адрес]»). Если значением выражения является число, оно рассматривается как числовой предикат, в противном случае оно преобразуется в логический тип (при помощи функции boolean()) и рассматривается как логический предикат. Числовой предикат со значением N эквивалентен логическому предикату «[position()=N]». Так, например, числовой предикат «[1]» означает «[position()=1]», а числовой предикат «[last()]» означает «[position()=last()]».

Важно помнить, что неявное сравнение с возвращаемым значением функции position() происходит, только если результатом вычисления выражения предиката является число. Например, выражение «$абзацы[1 or last()]» отличается от выражения «$абзацы[position()=1 or position()=last()]», потому что результатом вычисления выражения «1 or last()» будет логическое значение, а не число (в данном случае это всегда будет истина). Аналогично, выражение «книга[../@номер-книги]» отличается от выражения «книга[position()=../@номер-книги]», потому что «../@номер-книги» возвращает набор узлов, а не число.

Почти во всех случаях числовой предикат выбирает либо один узел из набора узлов, либо вообще ни одного узла, но это совершенно не обязательно: например, выражение «$x[count(*)]» выбирает все узлы, позиция которых совпадает с числом их дочерних элементов.

Как уже обсуждалось во второй главе, каждое выражение XPath вычисляется в каком-либо контексте. Для выражения, используемого в качестве предиката, контекст отличается от контекста содержащего его выражения. При вычислении каждого предиката контекст устанавливается следующим образом:

- **Контекстным узлом** (узлом, на который ссылается «.») становится проверяемый узел.
- **Контекстной позицией** (значение функции position()) становится позиция узла в множестве узлов, оставшихся с предыдущего шага, расположенных в порядке документа.
- **Размером контекста** (значение функции last()) становится число узлов, оставшихся с предыдущего шага.

Чтобы понять, как это работает, рассмотрим ФильтрующееВыражение «$заголовки [self::h1] [last()]». Вычисление начинается с набора узлов, находящегося в переменной «$заголовки» (если ее значение другого типа, то возникает ошибка). Первый предикат «[self::h1]» по очереди применяется к каждому узлу переменной «$заголовки». Во время его применения контекстным узлом является узел, к которому он в данный момент применяется. Выражение «self::h1» является путем типа ОтносительныйМаршрутПоиска, состоящим из единственного Шага: оно выбирает набор узлов. Если контекстным узлом является элемент <h1>, в этом наборе узлов будет находиться один узел – контекстный. В противном случае набор узлов будет пуст. Когда набор узлов преобразуется в логический тип, он будет истинен, если в нем есть узел, и ложен, если он пуст. Следовательно, первый предикат отфильтровывает узлы переменной «$заголовки», не являющиеся элементами <h1>.

Затем второй предикат применяется по очереди к каждому узлу из множества отобранных элементов <h1>. Каждый раз предикат «[last()]» возвращает одно и то же значение: число элементов <h1> в полученном множестве. Так как это числовой предикат, узел проходит проверку, если выполняется условие «[position()=last()]», то есть когда позиция узла в наборе (упорядоченном в порядке документа) совпадает с числом узлов в наборе. Следовательно, выражение «$заголовки [self::h1] [last()]» означает: «последний элемент <h1> в наборе узлов $заголовки, расположенных в порядке документа».

Заметьте, что это не то же самое, что выражение «$заголовки [last()] [self::h1]», которое означает: «набор узлов, состоящий из последнего узла переменной $заголовки, если он является элементом <h1>».

Действие предиката в ФильтрующемВыражении очень похоже на применение предиката к Шагу в МаршрутеПоиска, и, хотя они не связаны напрямую в грамматических правилах XPath, часто можно использовать предикаты, не задумываясь над тем, где они применяются: в ФильтрующемВыражении или в МаршрутеПоиска. Например, выражение «$абзац[1]» является ФильтрующимВыражением, в то время как выражение «абзац[1]» является ОтносительнымМаршрутомПоиска, состоящим из одного Шага. Основное отличие, на которое нужно обратить внимание в первую очередь, – это то, что в МаршрутеПоиска предикаты применяются только к самому последнему Шагу (например, в выражении «книга/автор[1]» предикат «[1]» выбирает первого автора в каждой книге) и что в ФильтрующемВыражении узлы всегда рассматриваются в порядке документа (в то время как в Шаге они могут находиться в прямом или обратном порядке документа, в зависимости от направления используемой оси).

Примеры

Выражение	Описание
`$абзацы`	Это ФильтрующееВыражение состоит из единственной СсылкиНаПеременную. Оно не обязательно является набором узлов.
`$абзацы[23]`	Данное ФильтрующееВыражение состоит из СсылкиНаПеременную, отфильтрованной предикатом. Оно выбирает 23-й узел в наборе узлов, являющемся значением переменной $абзацы, взятых в порядке документа.
`document('lookup.xml')`	Это ФильтрующееВыражение состоит из единственного ВызоваФункции. Оно выбирает корневой узел документа, заданного URI lookup.xml.
`key('имясотр', 'Иван Петров')[@loc='Москва']`	Данное ФильтрующееВыражение состоит из ВызоваФункции, отфильтрованного предикатом. В предположении, что ключ «имясотр» был определен очевидным образом, оно выбирает всех сотрудников с именем Иван Петров, находящихся в Москве.
`(//раздел|//подраздел)[заголовок='Введение']`	Данное ФильтрующееВыражение состоит из заключенного в скобки ВыраженияОбъединения, отфильтрованного предикатом. Оно выбирает все элементы <раздел> и <подраздел>, у которых есть дочерний элемент <заголовок>, содержимым которого является строка «Введение».
`document(//@href)[прейскурант][1]`	Данное ФильтрующееВыражение сначала выбирает все документы, на которые ссылаются ссылки URL, находящиеся в атрибутах href из любого места документа, затем выбирает из этого набора документов документы с элементом самого верхнего уровня <прейскурант>, после чего выбирает первый документ из оставшихся. Порядок узлов, находящихся в различных документах, не определен, поэтому если есть ссылки сразу на несколько прейскурантов, то невозможно заранее определить, который из них будет выбран.

Цифры (Digits)

Цифры — это последовательность десятичных цифр в диапазоне от 0 до 9. Последовательность может использоваться как часть всего числа, часть перед десятичной точкой или часть после десятичной точки.

Выражение	Синтаксис
Цифры	`[0-9]+`

Это порождающее правило записано в виде **регулярного выражения** и означает, что конструкция Цифры является последовательностью одного или более символов, каждое в диапазоне от 0 до 9. Квадратные скобки не обозначают необязательную конструкцию, как в некоторых других системах обозначе-

ния для описания синтаксиса; в данном случае они обозначают диапазон символов.

Определено в

XPath, раздел 3.7 (Lexical Structure), правило 31

Используется в

Число

Справка по использованию

Экземпляром конструкции Цифры является обычная последовательность десятичных цифр от 0 до 9.

Так как конструкция Число – лексема, внутри ее не могут находиться внутренние пробельные символы, поэтому конструкция Цифры также не может содержать пробельные символы.

Примеры в контексте

Выражение	Описание
89	Это Число состоит из единственной последовательности цифр, то есть из одного экземпляра Цифры.
3.14159	Это Число состоит из двух последовательностей цифр, то есть двух экземпляров Цифры. Первый экземпляр конструкции Цифры – «3», второй экземпляр – «14159».

Число (Number)

Число представляет постоянное числовое значение (с плавающей точкой).

Выражение	Синтаксис
Число	Цифры («.» Цифры?)? | «.» Цифры

Определено в

XPath, раздел 3.7 (Lexical Rules), правило 30

Используется в

ПервичноеВыражение

Справка по использованию и примеры

Число является постоянным числовым значением, записываемым в десятичной нотации. Его порождающее правило довольно сложным способом показывает четыре варианта записи числа:

Формат	Пример
Цифры	839
Цифры «.»	10.
Цифры «.» Цифры	3.14159
«.» Цифры	.001

Помните, что в XSLT все числовые значения обрабатываются как числа с плавающей точкой двойной точности. Более точное определение диапазона допустимых значений дано в главе 2 в разделе «Числовые значения». Для преобразования этих значений в целые числа можно использовать функции `round()`, `ceiling()` и `floor()`, описанные в главе 7.

Число как таковое не может иметь перед собой знак «минус», однако в любом контексте, где может использоваться число, также возможно использование УнарногоВыражения, которое состоит из знака «минус», за которым следует число.

В отличие от многих других языков программирования, в XPath **не** допускается запись числа в экспоненциальном представлении; миллион надо записывать как «1000000», а не как «1.0e6».

Шаг (Step)

Шаг пути выбирает в документе набор узлов, относящихся определенным образом к заданной точке отсчета; например, для данного набора узлов A шаг может найти их дочерние узлы или их предков и так далее.

Выражение	Синтаксис
Шаг	СпецификаторОси КритерийУзла Предикат* | СокращенныйШаг

Определено в

XPath, раздел 21, правило 4

Используется в

ОтносительныйМаршрутПоиска

Справка по использованию

Существует два способа определения шага: длинная форма и короткая форма. Короткая форма, СокращенныйШаг, может использоваться только для поиска контекстного узла или родителя контекстного узла. Полная форма может использоваться для следования по любой оси и для поиска любого типа узла.

Логически, шаг всегда находится справа от оператора пути «/». Левая сторона оператора пути возвращает набор узлов; шаг определяет для каждого из этих узлов набор связанных узлов, найденных путем перемещения от данного узла в заданном направлении, а конечным результатом выражения является результат применения шага к каждому узлу из левого набора узлов. Левая сторона оператора пути не всегда присутствует явно; например, шаг «ЗАГОЛОВОК» – это сокращение пути «self::node()/child::ЗАГОЛОВОК», а выражение пути «/descendant::РИСУНОК» может рассматриваться как сокращение для воображаемого выражения «root()/descendant::РИСУНОК», где «root()» обозначает корневой узел.

Хотя спецификация XPath определяет операцию шага относительно неформальным языком, более математическое определение может оказаться полезным для некоторых читателей. Шаг S может быть определен как функция $S(X) \Rightarrow N$, которая для данного узла X возвращает набор узлов N в том же документе. Оператор пути «/» может быть определен как функция $map(A, F) \Rightarrow U$, аргументами которой являются набор узлов A и функция шага F и возвращающая набор узлов U, являющийся объединением результатов применения функции шага F к каждому из узлов во входном наборе узлов A.

Например, шаг «ancestor::node()» для любого узла находит его предков. Когда этот шаг используется в выражении пути, таком как «$n/ancestor::node()», он возвращает набор узлов, в котором находятся все предки всех узлов из набора $n.

Сам по себе шаг определяется в терминах более простой концепции – оси. Каждая ось возвращает набор узлов, связанных с определенным узлом начала отсчета: например, его предыдущих одноуровневых узлов или предков. Шаг возвращает подмножество узлов на этой оси, выбранных по типу узла, его имени и по выражениям предикатов. Критерий узла может накладывать любые ограничения на тип и имя выбранных узлов, выражения предиката предоставляют любые логические условия, которым должны удовлетворять узлы, а позиционные фильтры ограничивают их относительное местонахождение.

Заметьте, что функция шага определяется в терминах неупорядоченных множеств и не существует концепции упорядоченности результата. Для понимания позиционных предикатов в шаге полезно представлять себе, что шаг извлекает узлы в определенном порядке, хотя формальное определение этого не требует. Вместо этого, предикаты определяются в терминах близости узла к началу отсчета оси. У оси есть направление; каждая ось, которая может быть использована в узле, является прямо или обратно направленной

осью, и действие позиционных предикатов (таких как «книги/книга[3]») определяется рассмотрением узлов из набора в порядке документа или обратном порядке документа. Для прямонаправленной оси позиционный предикат «[3]» вернет третий узел в порядке документа; для обратно направленной оси этот же предикат вернет третий узел в порядке, обратном порядку документа. Так что вычисление функции шага для данного контекстного узла происходит следующим образом:

1. Отыскиваются все узлы на оси, начиная с контекстного узла.
2. Из них выбираются удовлетворяющие критерию узла (то есть с требуемым типом узла и именем).
3. Остающиеся узлы нумеруются в порядке документа, если ось прямонаправленная, или в обратном порядке, если ось обратно направленная.
4. Первый (самый левый) предикат по очереди применяется к каждому узлу. При вычислении предиката контекстным узлом (то есть результат выражения «.») является рассматриваемый узел, контекстной позицией (возвращаемое значение функции position()) является число, присвоенное узлу на третьем шаге, а размером контекста (возвращаемым значением функции last()) – наибольший номер узла, полученный на третьем шаге. Числовые предикаты, такие как «[2]» или «[last()-1]», интерпретируются как сокращенные предикаты «[position()=2]» и «[position()=last()-1]» соответственно.
5. Шаги 3 и 4 повторяются для всех оставшихся предикатов.

Примеры

Выражение	Описание
child::заголовок	Выбирает дочерние элементы <заголовок> контекстного узла.
заголовок	Сокращенная форма выражения «child::заголовок».
attribute::заголовок	Выбирает атрибуты заголовок контекстного узла.
@заголовок	Сокращенная форма выражения «attribute::заголовок».
ancestor::xyz:*	Выбирает элементы-предки контекстного узла, с именами в пространстве имен с префиксом «xyz».
*[@ширина]	Выбирает все дочерние элементы контекстного узла с атрибутом ширина.
text() [starts-with(.,'The')]	Выбирает все дочерние текстовые узлы контекстного узла, текстовое содержимое которых начинается с символов «The».
*[@код][position() < 10]	Выбирает девять первых дочерних элементов контекстного узла с атрибутом код.

Выражение	Описание
`*[position() < 10][@код]`	Выбирает из девяти первых дочерних элементов контекстного узла те, у которых есть атрибут код.
`self::*[not(@код = preceding-sibling::*/@код)]`	Выбирает текущий узел элемента, при условии, что значение его атрибута код не совпадает со значением атрибута код любого предшествующего одноуровневого элемента.
`comment()`	Выбирает все узлы комментариев, являющиеся дочерними узлами контекстного узла.
`@comment()`	Сокращенная форма выражения «attribute::comment()», выбирающая все узлы комментариев на оси attribute. На оси attribute могут находиться только лишь узлы атрибутов, поэтому данное выражение всегда возвращает пустой набор узлов, тем не менее, это допустимый шаг пути.

ЭлементВыражения (ExprToken)

ЭлементВыражения является лексической единицей языка выражений XPath. Конструкция ЭлементВыражения определяет правила разделения лексем пробельными символами; сама по себе она не является лексической единицей, используемой в любом другом порождающем правиле.

Выражение	Синтаксис
ЭлементВыражения	«(» | «)» | «[» | «]» | «.» | «..» | «@» | «,» | «::» | КритерийИмени | ТипУзла | Оператор | ИмяФункции | НазваниеОси | Литерал | Число | ССылкаНаПеременную

Определено в

XPath, раздел 3.7 (Lexical Structure), правило 28

Используется в

Данное порождающее правило является порождающим правилом самого высокого уровня, на него не ссылается ни одна другая конструкция.

Справка по использованию

Данное порождающее правило определяет конструкции языка, считающиеся лексемами. Если конструкция является лексемой, она неделима: ее компоненты должны быть непрерывны и не могут разделяться пробельными символами. Конструкции более высокого уровня состоят из последовательности лексем и могут включать в себя пробельные символы для разделения лексем. Например, в выражении «child::xsl:*» три лексемы: «child», «::» и «xsl:*». Пробелы и переводы строки можно вставлять между лексемами, но не внутри их. Это означает, что в выражении могут находиться необязательные пробельные символы в местах, отмеченных ромбиком: «♦child♦::♦xsl:*♦».

> *В спецификации не говорится об этом явно, но по смыслу ясно, что нельзя свободно вставлять пробельные символы в середину экземпляра ЭлементВыражения, кроме, конечно, литерала. Это означает, например, что между двумя двоеточиями в выражении «child::node()» не может находиться пробел, потому что «::» является лексемой.*

Это не означает, что вы всегда должны вставлять пробельные символы между лексемами. Это необходимо только в том случае, если лексема может быть ошибочно воспринята как часть предыдущей лексемы, например в таком выражении, как «$x div 2» или «цена - скидка».

Пробельные символы определяются в правиле ПробельныеСимволыВыражения. Вполне допустимо разбивать выражение XPath на несколько строчек – перевод строки может использоваться в качестве разделителя везде, где можно вставлять пробел.

Примеры

Вот несколько примеров экземпляров ЭлементВыражения. Не допускается использование пробельных символов внутри лексем, кроме лексемы Литерал.

Лексема	Описание
[	Символ пунктуации, используемый в Предикатах.
item	ИмяБезДвоеточия.
my-namespace:*	КритерийИмени. Не допускается использование пробелов до или после двоеточия.
$сумма-недельных-продаж	ССылкаНаПеременную. Не допускается использование пробелов после знака «$».
93.7	Число.
"щепотка соли"	Литерал. Любые пробельные символы, находящиеся внутри Литерала, имеют значение – они часть его строкового значения.

Резюме

Выражения XPath используются в XSLT для выборки данных из исходного документа и для его обработки с целью получения данных, помещаемых в конечный документ. Выражения XPath играют в XML ту же роль, что и SQL-оператор `SELECT` в реляционных базах данных, – они позволяют выбирать определенные части документа для трансформации, так что можно получить требуемый вывод. Однако их применение не ограничивается таблицами стилей XSLT – их также можно применять с указателями XPointer для определения гиперссылок между документами, а многие реализации DOM позволяют использовать выражения XPath для поиска узлов внутри DOM.

Эта глава служит полным справочником по написанию таких выражений.

Подмножество языка выражений XPath можно использовать для создания образцов в таблицах стилей, соответствующих конкретным узлам. Понимание образцов – это следующий шаг к пониманию того, как работает XSLT, и в главе 6 они будут подробно рассмотрены.

6

Образцы

В этой главе определяется синтаксис и назначение **образцов XSLT**.

Образцы (patterns) используются в четырех местах таблицы стилей XSLT:

- В атрибуте `match` элемента `<xsl:template>` для определения узлов исходного документа, к которым применяется шаблон.
- В атрибуте `match` элемента `<xsl:key>` для определения узлов исходного документа, к которым применяется определение ключа.
- В атрибутах `count` и `from` элемента `<xsl:number>` для определения подсчитываемых узлов при генерировании чисел.

В каждом случае образец определяет условия, которым должен удовлетворять узел для того, чтобы быть выбранным. Наиболее часто образцы используются в атрибуте `match` элемента `<xsl:template>`, где они служат для обозначения узлов, к которым применяется шаблон. Например, элемент `<xsl:template match="abstract">` создает шаблонное правило, соответствующее каждому элементу `<abstract>`.

Большинство образцов таблиц стилей просты и интуитивно понятны, например:

Образец	Значение
`заголовок`	Соответствует любому элементу `<заголовок>`.
`глава/заголовок`	Соответствует любому элементу `<заголовок>`, родителем которого является элемент `<глава>`.
`speech[speaker="Гамлет"]`	Соответствует любому элементу `<speech>`, с дочерним элементом `<speaker>`, строковое значение которого равно строке «Гамлет».
`раздел/абзац[1]`	Соответствует любому элементу `<абзац>`, являющемуся первым дочерним элементом `<абзац>` элемента `<раздел>`.

Однако точные правила для более сложных образцов довольно специфичны, поэтому я опасаюсь, что некоторые места данной главы не будут легкими для понимания.

Образцы определяются в терминах имени, типа и строкового значения узла, а также его местоположения относительно других узлов дерева. Чтобы понять, как работают образцы, необходимо понимать древовидную модель, описанную во второй главе, и различные типы узлов.

Образцы очень похожи на выражения XPath, описанные в предыдущей главе, и оказывается, что они тесно связаны; однако образцы и выражения – это далеко не одно и то же. В терминах синтаксиса каждый образец является допустимым выражением XPath, но не каждое выражение XPath является допустимым образцом. Например, не имеет смысла использовать выражение «2+2» в качестве образца – каким узлам он будет соответствовать?

Правила для выражений были описаны в предыдущей главе. Выражения определяются в рекомендации XPath 1.0, позволяющей использовать их в контекстах, отличных от таблиц стилей XSLT. Например, выражения XPath применяются в спецификации XPointer для определения гиперссылок между документами, а также в некоторых реализациях DOM как предоставляемый приложениям способ перемещения по структуре данных DOM. Образцы, в свою очередь, тесно связаны со спецификацией XSLT Recommendation (правила определяются в разделе 5.2 спецификации) и применяются только в таблицах стилей.

Можно было бы определить в XSLT синтаксис и значение образцов независимо от правил XPath для выражений, но тогда могла бы возникнуть ненужная непоследовательность. Вместо этого проектировщики языка XSLT решили определить синтаксис образцов таким образом, чтобы каждый образец всегда был допустимым выражением, а затем определить формальное значение образца в терминах значения выражения.

Взгляните на простейший образец из вышеприведенных примеров – «заголовок». Если строка «заголовок» используется в качестве выражения, то из предыдущей главы мы знаем, что она является сокращенной записью выражения «./child::заголовок» и означает: «выбрать все дочерние элементы <заголовок> текущего узла». Каким образом мы получаем из этого определение образца «заголовок» как чего-то, соответствующего всем элементам <заголовок>?

В следующем разделе будет пояснено, как действует формальное определение образцов в терминах выражений. Тем не менее, на практике проще считать, что большинство образцов следует собственным правилам, подобно приведенным выше интуитивно понятным примерам, и обращаться к формальному определению в терминах выражений только для разрешения сложных случаев. Поэтому за формальным объяснением последует неформальное.

Формальное определение

Формальное определение процесса вычисления образцов выражается в терминах выражения XPath, эквивалентного образцу. Мы уже знаем, что каждый образец является допустимым выражением XPath. На самом деле правила составлены таким образом, что выражениями, которые могут использоваться в качестве образцов, могут быть лишь выражения, возвращающие набор узлов. Это сделано для того, чтобы существовала возможность определить, соответствует ли узел образцу, проверив, находится ли он в наборе узлов, возвращенном соответствующим образцом. Это ставит вопрос о контексте. Результатом выражения XPath «заголовок» являются все дочерние элементы <заголовок> контекстного узла. Входит ли сюда тот конкретный элемент <заголовок>, который мы пытаемся найти, или нет? Очевидно, что это зависит от контекста. Так как мы хотим, чтобы образец «заголовок» соответствовал каждому элементу <заголовок>, мы можем определить правило следующим образом: проверяемый нами узел (назовем его *N*) соответствует образцу «заголовок», если мы можем найти где-либо в документе узел (скажем, *A*), обладающий таким свойством, что если мы сделаем *A* контекстным узлом и вычислим выражение «заголовок», возвращающее набор узлов, то узел *N* будет являться частью результата. В этом примере не надо далеко искать узел *A*. На самом деле он находится не далее родительского узла *N*.

Так что причина, по которой элемент <заголовок> соответствует образцу «заголовок», заключается в том, что у него есть родительский узел, при использовании которого в качестве контекстного узла для выражения «./child::заголовок» оно возвращает набор узлов, в который входит элемент <заголовок>. Образец может быть интуитивно понятен, но, как видите, формальное объяснение начинает становиться довольно сложным.

В предварительной версии спецификации XSLT правила позволяли использовать в качестве образца практически любое выражение, возвращающее набор узлов. Например, можно было определить образец «ancestor::*[3]», совпадающий с любым узлом-прадедом какого-либо узла документа. Так вышло, что такой уровень универсальности не только не был необходим, но и трудно реализуем, поэтому было наложено дополнительное ограничение: в образцах можно использовать только оси child и attribute (различные оси описываются в разделе «НазваниеОси (AxisName)» главы 5). Вследствие этого, XSLT-процессор может искать узел *A* (использующийся в качестве контекстного для вычисления выражения) только среди предков искомого узла (*N*), включая и сам *N*.

Это подводит нас к формальному определению значения образца (прочтите это медленно):

> ***Узел N соответствует образцу P тогда и только тогда, когда существует узел A, являющийся предком узла N или самим узлом N, такой что вычисление P как выражения с узлом A в качестве контекстного, возвращает набор узлов, содержащий N.***

Правило говорит, что выражение вычисляется с узлом *A* в качестве контекстного. Оно не говорит, какими должны быть контекстная позиция и размер контекста, потому что это не имеет значения. Образцы могут использовать функции `position()` и `last()` только в предикате, поэтому этот вопрос не возникает.

Это означает, что существует следующий теоретический алгоритм проверки соответствия заданного узла *N* образцу *P*: для каждого узла, начиная с *N* и перебирая далее его предков вплоть до корневого узла, вычислять *P* как выражение XPath, с данным узлом в качестве контекстного. Если в результате получается набор узлов, содержащий *N*, значит, совпадение с образцом найдено; в противном случае надо продолжать попытки вплоть до корневого узла. Например:

- Элемент `<заголовок>` соответствует образцу «`заголовок`», потому что когда контекстным узлом является родитель элемента `<заголовок>`, выражение «`заголовок`» (являющееся сокращением для выражения «`./child::заголовок`») возвращает набор узлов, в который входит элемент `<заголовок>`.
- Узел со значением ID-атрибута `'n123'` соответствует образцу «`id('n123')`», потому что он входит в результат выражения «`id('n123')`», независимо от используемого контекстного узла.
- Образец «`глава//рисунок`» соответствует каждому элементу `<рисунок>`, являющемуся потомком элемента `<глава>`, потому что при вычислении выражения «`глава//рисунок`» с родительским узлом элемента `<глава>` в качестве контекстного будет возвращен каждый потомок `<рисунок>` элемента `<глава>`.
- Корневой узел соответствует образцу «`/`», потому что при вычислении выражения «`/`» с корневым узлом в качестве контекстного в полученном наборе узлов находится корневой узел.
- Атрибут `ширина` со значением `100` соответствует образцу «`@ширина[.=100]`», потому что, если сделать его родителя контекстным узлом, то данный атрибут будет входить в результат выражения «`@ширина[.=100]`».

На практике XSLT-процессоры вряд ли будут использовать этот алгоритм, он существует лишь для определения формальных правил. Обычно процессор может найти более быстрый способ выполнения проверки, что тоже хорошо, потому что в противном случае сопоставление с образцом было бы слишком дорогой операцией.

Хотя обычно формальные правила дают интуитивно ожидаемый результат, иногда могут случаться сюрпризы. Например, можно ожидать, что образец «`node()`» соответствует любому узлу, однако это не так. Выражение «`node()`» является сокращенным выражением «`./child::node()`», а оно может вернуть только узлы, являющиеся дочерними по отношению к какому-либо узлу. Так как корневые узлы, узлы атрибутов и пространств имен никогда не являются дочерними по отношению к какому-либо узлу (см. описание древовидной модели в разделе «Древовидная модель» главы 2), образец «`node()`» никогда с ними не совпадет.

Образцы с предикатами

Формальная эквивалентность образцов и выражений становится особенно важной при рассмотрении значения предикатов (условий в квадратных скобках), главным образом, для предикатов, явно или неявно использующих функции `position()` и `last()`.

Например, образец «`абзац[1]`» соответствует выражению «`./абзац[position()=1]`». Это выражение берет все дочерние элементы `<абзац>` контекстного узла, а затем фильтрует полученный набор, удаляя все узлы, кроме первого (в порядке документа). Поэтому образец «`абзац[1]`» соответствует любому элементу `<абзац>`, являющемуся первым дочерним элементом `<абзац>` своего родителя. Аналогично образец «`*[1][self::абзац]`» соответствует любому элементу, являющемуся первым дочерним узлом своего родителя и элементом `<абзац>`, тогда как образец «`абзац[last()!=1]`» соответствует любому элементу `<абзац>`, являющемуся дочерним узлом элемента с двумя или более дочерними элементами `<абзац>`.

Неформальное определение

Формальные правила для таких образцов, как «`книга//абзац`», содействуют рассмотрению образца как вычисляемого слева направо, потому что они записываются в терминах выражений, что означает поиск элемента `<книга>`, а затем всех его потомков `<абзац>` для того, чтобы определить, является ли один из них искомым.

Альтернативным пониманием значения выражения и способом, который, вероятно, используется в большинстве XSLT-процессоров для реализации алгоритма сопоставления с образцом, является чтение справа налево. Настоящий алгоритм сопоставления узла с образцом «`книга//абзац`», вероятно, похож на следующий:

- Проверить, является ли узел элементом `<абзац>`. Если нет, то он не совпадает с образцом.
- Проверить, существует ли у узла предок, являющийся элементом `<книга>`. Если нет, то он не совпадает с образцом.
- В противном случае узел совпадает с образцом.

Если в образце используются предикаты, то их можно проверять *по порядку*, например, для сопоставления с образцом «`speech[speaker='Гамлет']`» алгоритм, вероятно, будет следующим:

- Проверить, является ли узел элементом `<speech>`. Если нет, то он не совпадает с образцом.
- Проверить, есть ли у данного узла дочерний элемент `<speaker>` со строковым значением «`Гамлет`». Если нет, то он не совпадает с образцом.
- В противном случае узел совпадает с образцом.

Таким образом, большинство образцов может быть проверено просмотром только самого узла и, возможно, его предков, атрибутов и дочерних узлов. Образцы, проверка которых требует наибольшего времени, – это, вероятно, образцы, требующие просмотра и других узлов.

Рассмотрим, например, образец «абзац[last() - 1]», соответствующий элементу <абзац>, являющемуся предпоследним дочерним элементом <абзац> своего родителя. Большинство XSLT-процессоров, если в них не встроен исключительно хороший оптимизатор, будут проверять соответствие образцу конкретного элемента <абзац>, подсчитывая число дочерних узлов у его родительского элемента, сколько предшествующих одноуровневых узлов <абзац> есть у рассматриваемого узла, и сравнивая эти два числа. Выполнение этих действий для каждого обрабатываемого элемента <абзац> может потребовать много вычислительных ресурсов, особенно если у одного родителя сотни таких дочерних элементов. При использовании образцов «абзац[1]» или «абзац[last()]» есть шанс, что процессор найдет более быстрый способ выполнения проверки, но я бы на это не надеялся.

Если вы создаете таблицу стилей с большим количеством шаблонных правил, то время, требующееся на то, чтобы найти конкретное правило для применения к конкретному узлу, может иметь большое значение. Способ, который использует процессор для сопоставления, может меняться от одной реализации к другой, но можно быть уверенным в том, что образцы со сложными предикатами будут увеличивать время поиска совпадения.

Разрешение конфликтов

При использовании образца в определении шаблонного правила может возникнуть ситуация, когда несколько образцов соответствует одному узлу. Для таких ситуаций существуют правила разрешения конфликта. Один из факторов, принимаемых этими правилами во внимание, – это назначаемый по умолчанию приоритет образца, определяемый по способу его написания. Правила разрешения конфликтов и способ определения приоритета образца описаны в разделе <xsl:template> главы 4.

Как читать эту главу

Полная структура правил показана в приведенной ниже иерархии. Конструкции, помеченные звездочкой, определены в главе 5 «Выражения».

```
Образец
   ОбразецМаршрутаПоиска
      ОбразецОтносительногоПути
         ОбразецШага
            СпецификаторОсиChildИлиAttribute
               СокращенныйСпецификаторОси *
            КритерийУзла *
            Предикат *
```

```
        ОбразецКлючаИлиID
           Литерал *
```

Так как эта структура относительно простая и последовательная, я решил поступить иначе, чем в пятой главе, и расположил правила в порядке сверху вниз, начиная с конструкции Образец. Вы можете найти правила на следующих страницах:

Конструкция	Номер страницы
Образец	477
ОбразецМаршрутаПоиска	478
ОбразецОтносительногоПути	480
ОбразецШага	482
СпецификаторОсиChildИлиAttribute	487
ОбразецКлючаИлиID	488

Порождающие правила записываются так же, как и в главе 5.

Образец (Pattern)

Это конструкция самого высокого уровня в синтаксисе образцов XPath. Образец определяет условие, которое истинно или ложно для любого данного узла в исходном документе. Синтаксис образца является подмножеством синтаксиса ВыраженияОбъединения (и, соответственно, Выражения).

Синтаксис

Выражение	Синтаксис
Образец	ОбразецМаршрутаПоиска | Образец «|» ОбразецМаршрутаПоиска

Образец является либо ОбразцомМаршрутаПоиска, либо последовательностью образцов маршрута поиска, разделенных оператором «|».

Синтаксис образца маршрута поиска приведен ниже.

Используется в

атрибуте match элемента <xsl:template> (см. главу 4)

атрибуте match элемента <xsl:key> (см. главу 4)

атрибутах from и count элемента <xsl:number> (см. главу 4)

Справка по использованию

Хотя, с технической точки зрения, символ «|» является оператором объединения, проще считать его оператором «ИЛИ» – узел соответствует образцу «A | B», если он соответствует A или B или им обоим.

Примеры

ЗАГОЛОВОК	Образец «ЗАГОЛОВОК» является ОбразцомМаршрутаПоиска и поэтому Образцом.
предисловие | глава | приложение	Узел соответствует данному образцу, если он является элементом <предисловие>, <глава> или <приложение>.
/ | *	Узел соответствует данному образцу, если он является корневым узлом или узлом элемента.

ОбразецМаршрутаПоиска (LocationPathPattern)

ОбразецМаршрутаПоиска задает условия, основанные на имени узла, его типе, местоположении относительно других узлов и/или значений его атрибута ID и ключа, которым должен удовлетворять узел.

Эта конструкция является подмножеством конструкции ВыражениеПути в языке выражений (а не МаршрутаПоиска, как вы могли ожидать).

Синтаксис

Выражение	Синтаксис
ОбразецМаршрутаПоиска	«/» ОбразецОтносительногоПути ? | ОбразецКлючаИлиID ((«/»|«//») ОбразецОтносительногоПути) ? | «//» ? ОбразецОтносительногоПути

Приведенное выше порождающее правило – это способ определения синтаксиса в спецификации XSLT. Однако следующее порождающее правило легче понять и оно соответствует описанию в разделе «Справка по использованию».

Выражение	Синтаксис
ОбразецМаршрутаПоиска	«/» | ОбразецОтносительногоПути | «/» ОбразецОтносительногоПути | «//» ОбразецОтносительногоПути | ОбразецКлючаИлиID | ОбразецКлючаИлиID «/» ОбразецОтносительногоПути | ОбразецКлючаИлиID «//» ОбразецОтносительногоПути

Синтаксис ОбразцаОтносительногоПути **описывается на стр. 480, а синтаксис** ОбразцаКлючаИлиID **на стр. 488.**

Используется в

Образец **(см. стр. 477)**

Справка по использованию

Синтаксическое правило, скопированное выше из спецификации XSLT, может быть лучше понято перечислением семи различных видов образцов маршрута поиска:

«/»	Соответствует корневому узлу.
ОбразецОтносительногоПути	Соответствует образцу, который может находиться в любом месте документа.
«/» ОбразецОтносительногоПути	Соответствует образцу, определенному относительно непосредственных дочерних узлов корневого узла.
«//» ОбразецОтносительногоПути	Соответствует образцу, который может находиться в любом месте документа. Включение начального оператора «//» не оказывает влияния на значение образца, хотя влияет на его приоритет по умолчанию. Приоритет образца по умолчанию начинает действовать, когда два шаблонных правила соответствуют одному и тому же узлу: подробности находятся в описании элемента <xsl:template> в главе 4.
ОбразецКлючаИлиID	Соответствует узлу с заданным значением атрибута ID или ключа.
ОбразецКлючаИлиID «/» ОбразецОтносительногоПути	Соответствует образцу, определенному относительно дочерних узлов узла с заданным значением ID-атрибута или ключа.
ОбразецКлючаИлиID «//» ОбразецОтносительногоПути	Соответствует образцу, определенному относительно потомков узла с заданным значением ID-атрибута или ключа.

Образец «/» соответствует корневому узлу любого дерева. На самом деле это единственный образец, соответствующий корневом узлу. Это означает, что если у вас есть несколько деревьев (такая ситуация возникает при использовании в таблице стилей функции document(), описанной в главе 7), то образец «/» будет соответствовать корневому узлу каждого. Поэтому нельзя написать различные шаблонные правила для сопоставления с корневыми узлами разных деревьев. Обычным обходным способом решения этой проблемы является использование различных режимов для обработки каждого дерева (см. описание элемента <xsl:apply-templates> в главе 4) или начало обработки вторичных документов с узлов, находящихся ниже корневого. Такой обра-

зец, как «/item», будет соответствовать элементу <item>, являющемуся непосредственным дочерним элементом корневого узла. Такого рода образцы полезны, если в таблице стилей используются несколько документов, потому что позволяют различать их по имени элемента документа.

Образец «/@ширина» допустим, но не имеет смысла – он соответствует атрибуту ширина корневого узла, но такого узла не существует, так как у корневого узла не может быть атрибутов.

Другие варианты образцов маршрута поиска находятся в разделах ОбразецОтносительногоПути на стр. 480 и ОбразецКлючаИлиID на стр. 488.

Примеры

/	Соответствует корневому узлу.
/*	Соответствует самому внешнему узлу элемента (элементу документа). В случае, если дерево не является правильно построенным (см. раздел «Древовидная модель» главы 2), данный образец соответствует любому элементу, родителем которого является корневой узел.
/книги	Соответствует элементу <книги>, родителем которого является корневой узел.
//книга	Соответствует элементу <книга>, среди предков которого есть корневой узел; другими словами, любому элементу <книга>.
книга	Соответствует любому элементу <книга>.
id('рисунок-1')	Соответствует элементу со значением ID-атрибута 'рисунок-1'.
id('рисунок-1')//*	Соответствует любому элементу, являющемуся потомком элемента со значением ID-атрибута 'рисунок-1'.
key('empnr', '624381')/@dob	Соответствует атрибуту dob элемента со значением ключа empnr, равным '624381'.

ОбразецОтносительногоПути (RelativePathPattern)

ОбразецОтносительногоПути состоит из ОбразцаШага, определяющего условия, которым должен удовлетворять узел. Перед ним может находиться образец относительного пути, которому должны соответствовать родительский узел или другой узел-предок. Синтаксис образца относительного пути является подмножеством синтаксиса ОтносительногоМаршрутаПоиска в языке выражений XPath.

Синтаксис

Выражение	Синтаксис
ОбразецОтносительногоПути	ОбразецШага | ОбразецОтносительногоПути «/» ОбразецШага | ОбразецОтносительногоПути «//» ОбразецШага

Образец относительного пути является последовательностью, состоящей из одного или более ОбразцовШага, разделенных операторами «/» или «//».

Синтаксис ОбразцаШага описывается на стр. 482.

Используется в

ОбразецМаршрутаПоиска

Справка по использованию

В первом варианте узел соответствует образцу, если он удовлетворяет условиям (имя узла, тип узла и предикаты), определенным в ОбразцеШага. Простейшей и наиболее часто используемой разновидностью ОбразцаШага является просто имя элемента, например «заголовок».

Во втором варианте узел соответствует образцу, если он удовлетворяет условиям (имя узла, тип узла и предикаты), определенным в ОбразцеШага, а его **родительский узел** соответствует ОбразцуОтносительногоПути. Данный образец относительного пути может, в свою очередь, включать в себя условия, которым должен удовлетворять родитель узла или его предки.

В третьем варианте узел соответствует образцу, если он удовлетворяет условиям (имя узла, тип узла и предикаты), определенным в ОбразцеШага, и если у него есть предок, соответствующий ОбразцуОтносительногоПути. Данный образец относительного пути может, в свою очередь, включать в себя условия, которым должен удовлетворять родитель предка узла или его предки.

Заметьте, что хотя образец относительного пути в языке образцов и относительный маршрут поиска в языке выражений равнозначны, значение образца относительного пути проще всего описывается чтением ОбразцовШага справа налево, начиная с проверяемого узла и, при необходимости, перебором далее его предков. Значение относительного маршрута поиска, напротив, описывается рассмотрением Шагов слева направо, начиная с контекстного узла. Вероятно, большинство реализаций примет стратегию, похожую на описанный мной алгоритм.

Теоретически все, что можно сделать в образце относительного пути, также можно сделать и в одном ОбразцеШага, так как образец «A/B» означает то же самое, что и образец «B[parent::A]», а образец «A//B» означает то же самое, что и образец «B[ancestor::A]». Однако когда используется несколько шагов, запись с использованием операторов «/» и «//» читается значительно легче.

Примеры

заголовок	Это ОбразецШага, и, следовательно, простейшая разновидность образца относительного пути. Он соответствует любому элементу <заголовок>.
раздел/заголовок	Это образец относительного пути, состоящий из двух ОбразцовШага, объединенных оператором «/». Он соответствует элементу <заголовок>, родителем которого является элемент <раздел>.
глава//сноска	Это образец относительного пути, состоящий из двух ОбразцовШага, объединенных оператором «//». Он соответствует элементу <сноска>, являющемуся потомком элемента <раздел>.
глава/раздел//сноска	Более сложный образец относительного пути, соответствующий любому элементу <сноска>, являющемуся потомком элемента <раздел>, являющегося дочерним элементом элемента <глава>.

ОбразецШага (StepPattern)

ОбразецШага определяет условия, которым должен удовлетворять отдельный узел. Обычно это имя узла, тип узла и необязательный набор логических предикатов. Синтаксис ОбразцаШага является подмножеством синтаксиса Шага в языке выражений XPath.

Синтаксис

Выражение	Синтаксис
ОбразецШага	СпецификаторОсиChildИлиAttribute КритерийУзла Предикат *

Синтаксис конструкции СпецификаторОсиChildИлиAttribute приводится на стр. 487.
Конструкции КритерийУзла и Предикат описываются в главе 5.

Конструкция СпецификаторОсиChildИлиAttribute обязательно должна присутствовать, однако если вы посмотрите на стр. 487, то увидите, что она может быть пустой (и на практике обычно является именно такой). Так что в том случае, если вы не выбрали длинный вариант спецификатора оси, ОбразецШага состоит из необязательного знака «@» для указания на ось attribute, затем критерия узла, который обычно является именем узла или типом узла, таким как «text()» или «comment()», за которым следует ноль или более предикатов, которые являются логическими выражениями, заключенными в квадратные скобки.

Используется в

ОбразецОтносительногоПути

Справка по использованию

Мы по очереди рассмотрим каждый элемент конструкции `ОбразецШага`.

СпецификаторОси

`СпецификаторОсиChildИлиAttribute` может принимать вид «`attribute::`» (сокращенно «`@`») или «`child::`» (сокращаемый до пустой строки «»).

В формальных правилах вычисления образца шаги в образце относительного пути вычисляются слева направо, и выбор оси определяет, какие узлы просматривает шаг среди найденных на предыдущем шаге: дочерние узлы или узлы атрибутов.

Неформально говоря, проще всего представлять себе спецификатор оси как способ задания требуемого типа узлов.

- Если используется ось child, а критерий узла является критерием имени (например, «`заголовок`», «`*`» или «`svg:*`»), то мы ищем узел элемента.
- Если используется ось child, а критерий узла является типом узла (например, «`comment()`» или «`text()`»), то мы ищем узел данного типа. Если критерий узла – «`node()`», то мы ищем любые узлы на оси child: элементы, текстовые узлы, комментарии или инструкции обработки. Заметьте, что образец «`node()`», являющийся сокращенным вариантом образца «`child::node()`», не будет соответствовать корневым узлам, узлам атрибутов и пространств имен, потому что данные узлы никогда не бывают дочерними узлами другого узла.
- Если используется ось attribute, а критерий узла является критерием имени (например, «`@заголовок`», «`@*`» или «`@svg:*`»), то мы ищем узел атрибута.
- Если используется ось attribute, а критерий узла является типом узла (например, «`@comment()`» или «`@text()`»), то мы ищем узел данного типа, но только на оси attribute. Образцы «`@comment()`» и «`@text()`» допустимы, но не имеют смысла, потому что на оси attribute не могут находиться узлы комментариев или текстовые узлы. Тем не менее, критерий узла «`@node()`» ищет любой узел на оси attribute, поэтому он эквивалентен критерию «`@*`».

КритерийУзла

Данная разновидность критерия узла определена в языке выражений XPath, в частности она может являться:

- Критерием имени, например «`заголовок`», «`*`» или «`prefix:*`». (Последний вариант соответствует любому элементу или атрибуту, имя которого находится в заданном пространстве имен.)
- Типом узла, то есть одним из следующих выражений: «`comment()`», «`text()`», «`processing-instruction()`» или «`node()`».
- Именованной инструкцией обработки, например: «`processing-instruction('pi-target')`».

Предикат

Предикат определяется в языке выражений XPath (см. главу 5) как любое выражение, заключенное в квадратные скобки. Например, «[speaker='Гамлет']», «[@ширина > 100]», «[*]» или «[1]».

Числовой предикат интерпретируется как проверка местоположения узла относительно одноуровневых узлов. Во всех остальных случаях результат вычисления выражения преобразуется в логический тип, и если полученное значение – ложь, то образец не соответствует узлу.

Существует три ограничения на предикаты, используемые в образцах:

- Когда образец используется в атрибуте match элемента <xsl:template> или <xsl:key>, в предикате не должно быть ссылок на переменные. Это предназначено для предотвращения зацикленных ссылок – глобальные переменные могут вызывать ключи и шаблоны, поэтому если бы ключи и шаблоны можно было определять при помощи глобальных переменных, это могло бы приводить к бесконечной рекурсии. Это ограничение не относится к образцам, используемым в элементе <xsl:number>.
- По этим же причинам, когда образец используется в атрибуте match элемента <xsl:template> или <xsl:key>, предикат не должен использовать функцию key().
- Образцы не должны использовать в предикате функцию current() (описанную в главе 7). Это необходимо для того, чтобы принятие решения о соответствии узла образцу было предсказуемым и не зависело от текущего состояния обработки. Например, для атрибута match элемента <xsl:key> это означает, что узел либо включен в ключ, либо нет; он не может включаться в одних случаях и исключаться в других.

Предикаты можно разделить на две категории: зависящие от положения узла относительно одноуровневых узлов и не зависящие от нее. Позиционный предикат – это предикат, значением которого является число или использующий функции position() или last(); все остальные предикаты непозиционные. Например, предикаты «[1]», «[position()!=1]» и «[last()-1]» являются позиционными, тогда как «[@название='Токио']» и «[*]» – непозиционными.

Непозиционный предикат означает, что ОбразецШага не соответствует узлу, если не соответствует узлу предикат. Например, предикат «[@секретность='секрет']» истинен, когда у узла есть атрибут секретность со значением 'секрет', поэтому любой ОбразецШага, использующий этот предикат, не совпадет с узлом, у которого отсутствует атрибут секретность или его значение отлично от 'секрет'.

Значение позиционного предиката может быть установлено по формальным правилам, приведенным в начале этой главы. Однако их значение проще понять при помощи неформальных правил. Числовой предикат, такой как «[1]» или «[last()-1]», эквивалентен логическому предикату «[position()=1]» или «[position()=last()-1]». Поэтому для вычисления позиционного предиката мы должны знать, чему равны функции position() и last().

Не имеет смысла использовать позиционные предикаты на оси attribute, потому что порядок атрибутов не определен. Поэтому в следующем описании предполагается, что используется ось child.

Если в ОбразцеШага используется только один предикат или если предикат является первым, то:

- функция last() возвращает число одноуровневых узлов проверяемого узла, удовлетворяющих критерию узла (включая сам узел). Например, если мы проверяем соответствие элемента <абзац> образцу «абзац[last()=1]», то функция last() вернет число элементов <абзац>, являющихся дочерними узлами родителя проверяемого узла <абзац>. Данный образец будет соответствовать любому элементу <абзац>, являющемуся единственным дочерним элементом <абзац> своего родителя.
- функция position() возвращает местоположение узла среди этих одноуровневых узлов, рассматривая их в порядке следования в документе и начиная отсчет с 1. Поэтому образец «абзац[1]», означающий «абзац[position()=1]», будет соответствовать любому элементу <абзац>, являющемуся первым дочерним элементом <абзац> своего родителя в порядке следования элементов в документе.

Заметьте, что имеет значение положение узла относительно одноуровневых узлов, а не положение в последовательности обрабатываемых узлов. Представьте, например, что вам нужно обработать в документе все элементы <элемент-словаря> в алфавитном порядке. Вы можете написать:

```
<xsl:apply-templates select="//элемент-словаря">
     <xsl:sort/>
<xsl:apply-templates>
```

Затем представьте, что у вас есть следующие два шаблонные правила:

```
<xsl:template match="элемент-словаря[1]">
. . .
</xsl:template>

<xsl:template match="элемент-словаря">
. . .
</xsl:template>
```

Первое шаблонное правило будет применено к любому элементу <элемент-словаря>, являющемуся первым дочерним элементом <элемент-словаря> своего родителя. Но не к первому элементу <элемент-словаря> в алфавитном порядке и даже не к первому элементу <элемент-словаря> в документе, как можно было бы ожидать. Если вы хотите по-другому обработать элемент <элемент-словаря>, являющийся первым в алфавитном порядке, необходимо делать это следующим образом:

```
<xsl:template match="элемент-словаря">
<xsl:choose>
<xsl:when test="position()=1">
   . . .
```

```
</xsl:when>
<xsl:otherwise>
   . . .
</xsl:otherwise>
</xsl:template>
```

Это необходимо потому, что контекстной позицией в теле шаблонного правила является позиция узла в списке обрабатываемых узлов, в то время как результат принятия решения о соответствии узла образцу всегда один и тот же, независимо от контекста обработки.

Если и в ОбразцеШага используется несколько предикатов, то результат функций position() и last() в предикатах, следующих за первым, зависит от узлов, оставшихся после предыдущих предикатов. Таким образом, образец «speech[speaker='Гамлет'][1]» соответствует первому элементу <speech> среди одноуровневых элементов, у которых содержимым одного из дочерних элементов <speaker> является строка 'Гамлет'.

Функции position() и last() относятся к дочерним узлам одного и того же родителя, даже если используется оператор «//». Например, образец «глава//сноска[1]» соответствует любому элементу <сноска>, являющемуся потомком элемента <глава> и первым дочерним элементом <сноска> своего родителя. Не существует простого способа написать образец, соответствующий первому элементу <сноска> в элементе <глава>, потому что соответствующее выражение «(глава//сноска)[1]» не является правильным образцом. (Почему нет? Не существует какой-либо особой причины, просто это не разрешается спецификацией.)

При применении позиционных предикатов в образцах необходимо обращать внимание на производительность. Например, шаблон с образцом соответствия «абзац[last()-1]» кажется вполне разумным способом обработки предпоследнего абзаца в разделе. Однако примитивный XSLT-процессор раскроет этот предикат в «абзац[position()=last()-1]» и вычислит его, определив сначала положение текущего абзаца в разделе, а затем подсчитав число всех абзацев в разделе и сравнивая эти два числа. Если число абзацев в разделе велико, это может стать очень дорогостоящей операцией. Оптимизированный XSLT-процессор найдет лучшую стратегию, но если производительность имеет значение, то сначала стоит проделать предварительные измерения производительности.

Примеры

child::заголовок	Соответствует элементам с именем <заголовок>.
заголовок	Сокращенная форма выражения «child::заголовок».
attribute::заголовок	Соответствует атрибутам с именем <заголовок>.
@заголовок	Сокращенная форма выражения «attribute::заголовок».

`*[@ширина]`	Соответствует узлу элемента, у которого есть атрибут с именем `ширина`.
`text()[starts-with(.,'The')]`	Соответствует текстовым узлам, текстовое содержимое которых начинается с символов «The».
`p[@code][position() < 10]`	Соответствует элементу `<p>` среди первых девяти элементов `<p>` своего родителя, у которых есть атрибут `code`.
`p[position() < 10][@code]`	Соответствует элементу `<p>` среди первых девяти элементов `<p>` своего родителя, у которого есть атрибут `code`.
`*[not(@code = preceding-sibling::*/@code)]`	Выбирает узел элемента при условии, что у него нет атрибута `code`, значение которого совпадает со значением атрибута `code` любого предшествующего одноуровневого элемента.
`comment()`	Соответствует любому узлу комментария.
`@comment()`	Данный образец соответствует узлам комментариев, найденных на оси attribute их родительского узла. Так как на оси attribute могут находиться только узлы атрибутов, это условие никогда не может быть выполнено, тем не менее, оно является вполне допустимым ОбразцомШага.

СпецификаторОсиChildИлиAttribute (ChildOrAttributeAxisSpecifier)

Эта конструкция является подмножеством СпецификатораОси в синтаксисе выражений XPath. В образцах напрямую используются только две оси: ось child и ось attribute. Каждая может быть записана в полном или сокращенном виде.

Синтаксис

Выражение	Синтаксис
`СпецификаторОсиChildИлиAttribute`	`СокращенныйСпецификаторОси | ( «child» | «attribute» ) «::»`

Конструкция СокращенныйСпецификаторОси является частью синтаксиса выражения, определенного в главе 5, и является либо символом «@», обозначающим ось attribute, либо пустой строкой, обозначающей ось child.

Так как СокращенныйСпецификаторОси может быть пустым, то и СпецификаторОсиChildИлиAttribute также может быть пустым. Практически это означает, что критерий узла без явно указанного спецификатора оси (например, имя элемента или тип узла, такой как «text()») неявно использует ось child.

Удивительным последствием этого является то, что образец «node()» будет соответствовать только узлу, являющемуся чьим-нибудь дочерним узлом, то есть он никогда не соответствует корневому узлу, узлу атрибута или пространства имен.

Используется в

ОбразецШага

Справка по использованию

См. раздел ОбразецШага на стр. 482.

В отличие от выражений, в образцах явно доступны только две оси: ось child и ось attribute. Однако проверка наличия родственных узлов на другой оси может быть выполнена в предикате ОбразцаШага. В предикате может быть использовано любое выражение, поэтому доступны любые оси. Например, образец

```
подпись[preceding-sibling::*[1][self::рисунок]]
```

соответствует элементу <подпись>, предшествующим одноуровневым элементом которого является элемент <рисунок>.

Примеры

child::	Указывает на ось child.
	(Левая ячейка таблицы намеренно оставлена пустой.) Указывает на ось child, так как «child::» является спецификатором оси по умолчанию.
attribute::	Указывает на ось attribute.
@	Эквивалентно спецификатору «attribute::».

Примеры, показывающие использование спецификатора СпецификатораОси-ChildИлиAttribute, находятся в разделе ОбразецШага на стр. 482.

ОбразецКлючаИлиID (IdKeyPattern)

Образцы данного типа совпадают с проверяемым узлом (или одним из его предков) в том случае, если у него есть атрибут ID или ключ с заданным значением.

Данная конструкция является подмножеством конструкции ВызовФункции в синтаксисе выражений XPath, описанной в главе 5. Единственные функции, которые можно вызывать в самих образцах, а не в предикатах, – это функции id() и key(), и вызывать их можно, передавая в качестве аргументов только литералы.

Функции id() и key() описаны в главе 7.

Синтаксис

Выражение	Синтаксис
ОбразецКлючаИлиID	«id» «(» Литерал «)» | «key» «(» Литерал «,» Литерал «)»

Используется в

ОбразецМаршрутаПоиска

Справка по использованию

Это средство предоставляет возможность, эквивалентную возможности определять в таблицах стилей CSS стиль для конкретного узла в исходном документе. Существует несколько способов использования этого средства:

- Если вы хотите использовать для конкретного исходного документа таблицу стилей общего назначения, но переопределить ее поведение для определенных узлов, вы можете написать таблицу стилей, импортирующую таблицу стилей общего назначения, а затем написать переопределяющие правила в виде шаблонов, соответствующих заданным узлам в исходном документе.
- В некоторых случаях исходный документ динамически генерируется из базы данных. Возможно, что в документе есть что-то, что вы хотели бы выделить, скажем, строка, использовавшаяся для поиска данной записи. Вы можете пометить эту строку при генерации исходного документа, присвоив ей особое значение ID-атрибута, известное таблице стилей.

На практике эта конструкция не так полезна, как кажется с первого взгляда, потому что фактическое значение идентификатора или ключа должно быть строковой константой, как мы видели ранее, в образцах соответствия нельзя использовать переменные.

Функции id() и key() могут быть выражены в терминах предикатов с выражением, использующим атрибут ID или значение ключа во всех случаях, поэтому это средство может рассматриваться как простое сокращение. Например, если элементы <книга> идентифицированы по свойству ISBN, являющемуся их дочерним элементом, следующие объявления эквивалентны:

используя обычное сопоставление с образцом:

```
<xsl:template match="книга[isbn='1-861002-68-8']">
```

используя определение ключа:

```
<xsl:key name="isbn-key" match="книга" use="isbn"/>
<xsl:template match="key('isbn-key', '1-861002-68-8')>
```

Конечно, производительность этих вариантов будет отличаться, но это зависит от реализации XSLT-процессора.

Примеры

`id('рисунок1')`	Соответствует узлу с ID-атрибутом, равным строке `'рисунок1'`. Атрибут является ID-атрибутом, если его тип в DTD объявлен как ID (имя атрибута не имеет значения).
`key('empnr', '517541')`	Соответствует узлу со значением `'517541'` ключа с названием «`empnr`».

В следующем примере показано, как можно использовать это средство в таблице стилей.

Пример: Использование образца key() для форматирования конкретного узла

В этом примере я покажу, как использовать образец `key()`, чтобы отформатировать один выбранный узел иначе, чем остальные.

Исходный документ

Исходный документ, `маршрут.xml`, является маршрутом путешествия:

```
<маршрут>
<день номер="1">Прибытие в Каир</день>
<день номер="2">Посещение пирамид в Гизе</день>
<день номер="3">Археологический музей в Каире</день>
<день номер="4">Перелет в Луксор; автобус в Асуан</день>
<день номер="5">Посещение храма на острове Филы и Асуанской плотины</день>
<день номер="6">Круиз в Идфу</день>
<день номер="7">Круиз в Луксор; посещение храма в Карнаке</день>
<день номер="8">Долина Фараонов</день>
<день номер="9">Перелет назад из Луксора</день>
</маршрут>
```

Таблица стилей

Мы начнем с простой таблицы стилей, `маршрут.xsl`, для отображения данного маршрута:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="">

<xsl:template match="/">
     <html>
       <head>
          <title>Маршрут путешествия</title>
     </head>
     <body><center>
          <xsl:apply-templates select="//день"/>
          </center></body>
```

```
        </html>
</xsl:template>

<xsl:template match="день">
    <h3>День <xsl:value-of select="@номер"/></h3>
    <p><xsl:apply-templates/></p>
</xsl:template>
</xsl:stylesheet>
```

Теперь мы уточним таблицу, импортировав ее в другую таблицу стилей, `сегодня.xsl`**, которая отображает красным цветом планы на пятый день:**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
     version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import href="маршрут.xsl"/>
<xsl:key name="номер-дня" match="день" use="@номер"/>

<xsl:template match="key('номер-дня','5')//text()">
    <font color="red"><xsl:value-of select="."/></font>
</xsl:template>

</xsl:stylesheet>
```

Конечный документ

После применения таблицы стилей получаем следующий конечный документ:

```
<html>
   <head>
      <META http-equiv="Content-Type"
          content="text/html; charset=utf-8">
      <title>Маршрут путешествия</title>
   </head>
   <body>
      <center>
         <h3>День 1</h3>
         <p>Прибытие в Каир</p>
         <h3>День 2</h3>
         <p>Посещение пирамид в Гизе</p>
         <h3>День 3</h3>
         <p>Археологический музей в Каире</p>
         <h3>День 4</h3>
         <p>Перелет в Луксор; автобус в Асуан</p>
         <h3>День 5</h3>
         <p><font
       color="red">Посещение храма на острове Филы и Асуанской плотины</font></p>
<h3>День 6</h3>
         <p>Круиз в Идфу</p>
         <h3>День 7</h3>
         <p>Круиз в Луксор; посещение храма в Карнаке</p>
```

```
            <h3>День 8</h3>
            <p>Долина Фараонов</p>
            <h3>День 9</h3>
            <p>Перелет назад из Луксора</p>
         </center>
      </body>
   </html>
```

Этот пример показывает один вариант использования данной возможности, и я должен признать, что он не очень убедителен. Вы можете сделать то же самое, переписав образец в виде «день[@номер=5]» вообще без использования ключа. На практике я бы предпочел выделять красным конкретный день в зависимости от параметра, переданного таблице стилей. К сожалению, не существует способа создать образец, результат которого будет зависеть от значения параметра, поэтому логика должна быть закодирована с применением элемента <xsl:choose>.

Резюме

В этой главе описаны синтаксис и значение образцов, которые применяются в основном в таблицах стилей XSLT для определения соответствия между шаблонными правилами и узлами исходного документа, а также в элементах <xsl:key> и <xsl:number>.

Образцы, хотя их синтаксис является подмножеством синтаксиса выражений XPath, который был описан в главе 5, вычисляются иначе, чем выражения; правда, мы видели, что формальные правила выражают значение образца в терминах соответствующего выражения.

В следующей главе описывается библиотека стандартных функций, которые можно использовать в выражениях XPath таблицы стилей.

7

Функции

В этой главе описаны все стандартные функции из спецификаций XPath и XSLT, которые могут использоваться в выражениях.

Для каждой функции приводится имя, краткое описание назначения, ссылка на определение функции в спецификации XSLT или XPath, перечень ожидаемых аргументов и возвращаемое значение, формальные правила, определяющие алгоритм работы функции, и, наконец, советы по применению и примеры.

Некоторые из функций определены в спецификации XPath, другие – в спецификации XSLT. Если функция используется в пределах таблицы стилей XSLT, не имеет значения, где она определена; в случае использования XPath вне контекста XSLT следует помнить, что гарантированно доступны только функции XPath (известные как *основные функции*). Эти функции в приводимой ниже таблице отмечены символом ✓.

Синтаксис вызова функции описан в качестве составной части синтаксиса выражений XPath в главе 5. Он определяет, в какой части выражения может быть сделан вызов, а в какой это делать запрещено: единственным значимым ограничением является то, что вызов функции не может присутствовать в правой части оператора пути «/». В пределах вызова функции передаваемые в качестве аргументов значения могут быть представлены произвольными выражениями XPath, для которых выполняются только правила, связанные с типизацией данных (например, некоторые функции в качестве одного из аргументов ожидают получение набора узлов). Так что вызов функции вроде «`count(..)`» хоть и выглядит странно, на деле является абсолютно допустимым. «`..`» – это корректное выражение XPath, возвращающее родителя контекстного узла.

Функции упорядочены в алфавитном порядке для ускорения поиска в случаях, когда известно, какая именно из функций нужна. Но в случае когда

известна лишь требуемая функциональность, вполне может пригодиться приведенная ниже классификация. В таблице основные функции XPath отмечены символом ✓, а дополнительные функции, определяемые спецификацией XSLT, символом ×. Эта разница имеет значение только для тех случаев, когда выражение XPath используется вне контекста таблиц стилей XSLT.

Категория	Функция
Функции, преобразующие значения одного типа в значения другого типа	✓ `boolean()` × `format-number()` ✓ `number()` ✓ `string()`
Арифметические функции	✓ `ceiling()` ✓ `floor()` ✓ `round()`
Работа со строками	✓ `concat()` ✓ `contains()` ✓ `normalize-space()` ✓ `starts-with()` ✓ `string-length()` ✓ `substring()` ✓ `substring-before()` ✓ `substring-after()` ✓ `translate()`
Суммирование	✓ `count()` ✓ `sum()`
Получение имен узлов и идентификаторов	× `generate-id()` × `lang()` ✓ `local-name()` ✓ `name()` ✓ `namespace-uri()` × `unparsed-entity-uri()`
Логические функции	✓ `false()` ✓ `true()` ✓ `not()`
Функции, возвращающие информацию о контексте	× `current()` ✓ `last()` ✓ `position()`
Функции поиска узлов	× `document()` × `key()` ✓ `id()`

Категория	Функция
Функции, предоставляющие информацию о рабочем окружении	× element-available() × function-available() × system-property()

Все эти функции принадлежат стандартному пространству имен, их имена записываются без префиксов. По сути дела этими функциями стандартное пространство имен и ограничивается – функции расширения, предоставляемые разработчиками пакетов, пользователями или сторонними разработчиками, всегда принадлежат другому пространству имен, так что их имена при вызове всегда предваряются соответствующими префиксами.

boolean

Функция boolean() преобразует аргумент в значение логического типа.

К примеру, выражение «boolean(1)» возвращает значение истина.

Определена в

XPath, раздел 4.3

Формат

boolean(значение) ⇒ логическое значение

Аргументы

Аргумент	Тип данных	Смысл
значение	любой	Значение, которое должно быть приведено к логическому типу.

Результат

Значение логического типа; результат преобразования аргумента.

Правила

Любое значение может быть преобразовано в значение логического типа. Правила преобразования следующие:

Тип данных аргумента	Правила преобразования
Число	Ноль преобразуется в ложь; все прочие числа – в значение истина.
Строка	Строка нулевой длины преобразуется в ложь; любая другая строка – в значение истина.

Тип данных аргумента	Правила преобразования
Логический	Значение не изменяется.
Набор узлов	Пустой набор узлов преобразуется в ложь; любой другой набор – в значение истина. Набор узлов, представляющий какое-либо дерево, всегда преобразуется в значение истина, поскольку у дерева всегда есть корневой узел.

Применение

В большинстве случаев приведение к логическому типу происходит автоматически, когда такое преобразование предписывается контекстом, и явно вызывать функцию boolean() имеет смысл только для принудительного преобразования.

Примеры

В следующем примере сообщение отображается в том случае, если исходный документ содержит элемент <header>, но не содержит элемента <footer>, либо содержит элемент <footer> и не содержит элемента <header>.

```
<xsl:if test="boolean(//header) != boolean(//footer)">
   <xsl:message>Документ должен содержать заголовки и нижние колонтитулы
                        либо ни тех, ни других </xsl:message>
</xsl:if>
```

Преобразование двух наборов узлов – «//header» (истинен, если в документе присутствует хотя бы один элемент <header>) и «//footer» (истинен, если в присутствует хотя бы один элемент <footer>) – в данном случае производится явным вызовом, поскольку необходимо сравнивать логические значения, а не наборы узлов.

В следующем примере переменной присваивается значение истина или ложь в зависимости от того, присутствуют ли в документе сноски. В данном случае можно обойтись без явного преобразования, поскольку его можно будет выполнить позже, при использовании переменной, но с другой стороны – более эффективно хранить в переменной единственное значение логического типа вместо целого набора узлов. Сообразительный XSLT-процессор поймет, что выражение «//footnote» используется в контексте, где требуется получить значение логического типа, и просканирует документ лишь до первой встречной сноски, вместо того чтобы собирать сноски со всего документа.

```
<xsl:variable name="используются-нижние-колонтитулы" select="boolean(//footnote)"/>
```

См. также

true() на стр. 613

false() на стр. 528

ceiling

Функция `ceiling()` возвращает наименьшее целое, которое больше либо равно численному значению аргумента.

К примеру, результатом вызова «`ceiling(33.9)`» является число `34`.

Определена в

XPath, раздел 4.4

Формат

`ceiling(значение) ⇒ число`

Аргумент

Аргумент	Тип данных	Смысл
значение	число	Исходное значение. Если аргумент имеет тип, отличный от числового, сначала происходит приведение к этому типу в соответствии с правилами, определенными для функции `number()`.

Результат

Целочисленное значение: результат преобразования первого аргумента в число и последующего округления этого числа до ближайшего большего целого.

Правила

Если аргумент не является числом, то прежде всего он преобразуется в число. Правила преобразования приведены в описании функции `number()` на стр. 580. Если значение аргумента представлено набором узлов, эти правила применяются к первому из узлов набора, в порядке следования в документе.

Если аргумент является целым числом, возвращается это число. В противном случае происходит округление до ближайшего большего целого.

Числовой тип данных в XPath включает специальные значения, такие как бесконечность, отрицательный нуль и NaN (не-число), которые описаны в разделе «Типы данных» главы 2.

Если аргумент представлен не-числом – это происходит в случае, когда передаваемая строка не может быть преобразована в число, – результатом вызова также будет не-число. Аналогичным образом функция работает для значений положительной и отрицательной бесконечности.

Если значение аргумента больше -1.0, но меньше нуля, оно округляется до отрицательного нуля.

Применение и примеры

Проиллюстрируем работу функции следующими примерами:

```
ceiling(1.0) = 1.0
ceiling(1.6) = 2.0
ceiling(17 div 3) = 6.0
ceiling(-3.0) = -3.0
ceiling(-8.2) = -8.0
ceiling('xxx') = NaN
ceiling(-0.5) = -0
```

Эта функция может быть полезна при вычислении размера таблицы. Если дан набор узлов `$ns` и необходимо расположить значения в три колонки, число строк можно вычислить с помощью следующего выражения: «`ceiling(count($ns) div 3)`».

См. также

`floor()` на стр. 529

`round()` на стр. 586

concat

Функция `concat()` принимает два или более аргументов. Каждый из аргументов преобразуется в строку, после чего происходит их сцепление.

Так, результатом вызова «`concat('Джейн', ' ', 'Браун')`» будет строка «`Джейн Браун`».

Определена в

XPath, раздел 4.2

Формат

`concat(значение1, значение2, значение3, …)` ⇒ `строка`

Аргументы

Эта функция уникальна в том смысле, что может принимать произвольное число аргументов (два или более).

Аргумент	Тип данных	Смысл
значение 1 … значение n	строка	Исходное значение. Если значение имеет тип, отличный от строкового, сначала происходит преобразование аргумента в строку в соответствии с правилами, определенными в описании функции `string()`.

Результат

Строковое значение; результат преобразования каждого из аргументов в строку и конкатенации получившихся строк.

Правила

Каждый из аргументов преобразуется в строку в соответствии с правилами, определенными для функции string(), после чего из аргументов формируется строка результата – в порядке их следования.

Применение и примеры

Функция concat() зачастую является удобной альтернативой использованию многочисленных элементов <xsl:value-of> для создания отображаемой строки. Следующие конструкции эквивалентны:

```
<xsl:value-of select="concat(имя, ' ', фамилия)"/>
```

и:

```
<xsl:value-of select="имя"/>
<xsl:text> </xsl:text>
<xsl:value-of select="фамилия"/>
```

Другой случай, когда использование concat() удобно, – это при определении ключа (см. <xsl:key> в главе 4, а также описание функции key() на стр. 548). Ключи XSLT не могут состоять из нескольких значений, но это ограничение можно обойти, соединяя части ключа подходящим разделителем. К примеру:

```
<xsl:key name="полное-имя" match="персона"
                           use="concat(имя, ' ', фамилия)"/>
```

Этот ключ может впоследствии использоваться для поиска человека (или людей) с определенным именем, с помощью такого выражения:

```
<xsl:for-each select="key('полное-имя', 'Питер Джонс')"/>
```

У функции concat() есть и более интересное применение: создание списка имен, элементы которого разделены пробелами. Поскольку в XSLT не существует сложных типов данных, позволяющих хранить промежуточные результаты, такие списки довольно часто являются оптимальным вариантом для хранения, заменяя существующие в других языках массивы. Разумеется, можно использовать любые разделители, но пробелы, как правило, наиболее удобны, поскольку компоненты списка в случае использования пробелов легко разделяются с помощью функций normalize-space(), substring-before() и substring-after().

Пример: Создание списка, элементы которого разделяются запятыми

В следующем примере содержится шаблон, который для набора узлов из элементов <город> (включающих атрибут страна) создает список городов в формате *город, страна*.

Исходный документ

Исходный документ, города.xml:

```
<города>
   <город название="Париж" страна="Франция"/>
   <город название="Рим" страна="Италия"/>
   <город название="Ницца" страна="Франция"/>
   <город название="Мадрид" страна="Испания"/>
   <город название="Милан" страна="Италия"/>
   <город название="Флоренце" страна="Италия"/>
   <город название="Неаполь" страна="Италия"/>
   <город название="Лион" страна="Франция"/>
   <город название="Барселона" страна="Испания"/>
</города>
```

Таблица стилей

Таблица стилей вывести-города.xsl перебирает все элементы <город> и для каждого вызывает функцию concat() с целью отображения элемента в формате *город, страна*.

```
<xsl:transform
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
>
<xsl:output indent="yes"/>
<xsl:template match="/">
  <вывод>
    <xsl:for-each select="//город">
      <город><xsl:value-of select="concat(@название, ', ', @страна)"/></город>
    </xsl:for-each>
  </вывод>
</xsl:template>
</xsl:transform>
```

Вывод

```
<?xml version="1.0" encoding="utf-8" ?>
<вывод>
   <город>Париж, Франция</город>
   <город>Рим, Италия</город>
   <город>Ницца, Франция</город>
   <город>Мадрид, Испания</город>
   <город>Милан, Италия</город>
```

```
    <город>Фиренце, Италия</город>
    <город>Неаполь, Италия</город>
    <город>Лион, Франция</город>
    <город>Барселона, Испания</город>
  </вывод>
```

См. также

contains() в следующем разделе

substring() на стр. 595

contains

Функция contains() определяет факт вхождения подстроки в строку.

К примеру, выражение «contains('Santorini', 'ant')» имеет значение истина.

Определена в

XPath, раздел 4.2

Формат

contains(значение, подстрока) ⇒ логическое значение

Аргументы

Аргумент	Тип данных	Смысл
значение	строка	Строка, в которой производится поиск. Если значение имеет тип, отличный от строкового, сначала происходит преобразование аргумента в строку в соответствии с правилами, определенными в описании функции string().
подстрока	строка	Подстрока. Если значение имеет тип, отличный от строкового, сначала происходит преобразование аргумента в строку в соответствии с правилами, определенными в описании функции string().

Результат

Значение логического типа: истина, если подстрока содержится в строке, или ложь в противном случае.

Правила

Результатом вычисления функции является значение истина, если первая из строк содержит последовательность символов, в которой каждый из симво-

лов представлен тем же Unicode-значением, что и соответствующий символ подстроки.

Для пустой подстроки результат всегда истина.

Если первая строка (в которой производится поиск) пуста, результат всегда ложь, за исключением тех случаев, когда и вторая строка также пуста.

Применение и примеры

Функция contains() часто полезна в случаях, когда строка обладает какой-либо внутренней структурой. К примеру, проверка:

```
<xsl:if test="contains($name, 'Michael') and contains($name, 'Kay')"/>
```

заканчивается положительно, если переменная «$name» содержит строковое значение «Michael Kay» или «Kay, Michael», или «Michael H. Kay», или «Michaelmas Kayaks, Inc.».

Следует проявлять осторожность при использовании букв с акцентами или других сложных символов, поскольку в Unicode они могут представляться различными способами, а если представление символа в различных строках разное, сравнение теряет смысл.

См. также

substring() на стр. 595

substring-after() на стр. 599

substring-before() на стр. 601

count

Аргументом функции count() служит набор узлов; функция возвращает число узлов в этом наборе.

Так, выражение «count(.)» всегда возвращает значение 1.

Определена в

XPath, раздел 4.1

Формат

```
count(узлы)  ⇒  число
```

Аргументы

Аргумент	Тип данных	Смысл
узлы	набор узлов	Исходный набор узлов. Если аргумент не является набором узлов, возникает ошибка.

Результат

Число, выражающее количество отдельных узлов в наборе.

Правила

Функция `count()` принимает набор узлов в качестве аргумента и возвращает число узлов в наборе.

Считаются только узлы, которые сами по себе входят в набор узлов. Узлы, являющиеся потомками этих узлов, исключаются из рассмотрения.

Применение

Набор узлов – это математическое множество, и, следовательно, все входящие в него узлы различны (то есть являются различными узлами, хотя могут иметь одинаковые строковые значения). Если сформировать набор узлов при помощи оператора объединения «`|`», каждый узел, присутствующий в обоих операндах, будет включен в набор лишь в одном экземпляре. Из этого, к примеру, следует, что результатом вычисления выражения «`count( . | / )`» будет `1` только в том случае, когда контекстный узел является корневым.

В XPath не существует иного способа определить, что узел входит одновременно в два набора узлов, так что описанный прием может быть весьма полезен. К примеру, чтобы понять, входит ли контекстный узел в набор узлов, представленный переменной «`$особый`», можно выполнить:

```
<xsl:if test="count($особый | .) = count($особый)">
    . . .
</xsl:if>
```

Следует избегать применения функции `count()` в целях определения, пуст ли набор узлов, таким образом:

```
<xsl:if test="count(книга[автор='Хемингуэй']) != 0">
    . . .
</xsl:if>
```

Эту задачу лучше решать иначе:

```
<xsl:if test="книга[автор='Хемингуэй']"> . . . </xsl:if>
```

Оба примера проверяют, есть ли у текущего узла дочерние элементы `<книга>` с дочерним элементом `<автор>`, значение которого «Хемингуэй». Второй пример не только более лаконичен, он более прост в оптимизации для XSLT-процес-

сора. Во многих реализациях сканирование элементов книг будет прекращено при нахождении хотя бы одного элемента, соответствующего шаблону.

Функция count() зачастую является эффективной альтернативой использованию <xsl:number>. К примеру, если текущим узлом является элемент <маркер>, результатом вычисления «count(preceding-sibling::маркер)+1» будет то же значение, которое получается при выполнении <xsl:number/> без атрибутов. Преимущество использования функции count() состоит в том, что она позволяет более гибко определять предмет вычислений и может использоваться в выражениях. При этом <xsl:number> является простым способом получения порядкового номера, его форматирования и вставки в конечное дерево – в одно действие; и в некоторых случаях легче поддается оптимизации процессором.

Не следует использовать count(), если требуемый результат можно получить с помощью функции last(). Подобные ситуации возникают при обработке набора узлов с помощью <xsl:apply-templates> или <xsl:for-each>; в этом случае число узлов в наборе может быть получено вызовом функции last(). Скорее всего, малоэффективной будет такая конструкция:

```
<xsl:for-each select="книга[автор='Хемингуэй']">
  <h2>Книга <xsl:value-of select="position()"/> из
    <xsl:value-of
         select="count(../книга[автор='Хемингуэй'])"/>
  </h2>
    . . .
</xsl:for-each>
```

поскольку – если используемый XSLT-процессор не очень умен – выражение «../книга[автор='Хемингуэй']» будет вычисляться на каждой итерации цикла.

Вместо этого следует использовать:

```
<xsl:for-each select="книга[автор='Хемингуэй']">
   <h2>Книга <xsl:value-of select="position()"/> из
            <xsl:value-of select="last()"/>
   </h2>
    . . .
</xsl:for-each>
```

Как вариант можно сохранить набор узлов в переменной, для которой вычисления не будут производиться повторно.

Примеры

В следующем примере мы получаем число элементов <footnote> в исходном документе:

```
<xsl:value-of select="count(//footnote)"/>
```

В следующем примере переменной присваивается значение числа атрибутов текущего узла:

```
<xsl:variable name="num-atts" select="count(@*)"/>
```

Пример: Подсчет уникальных значений

В данном примере мы определяем число уникальных значений атрибута страна в списке элементов <город>.

Исходный документ

Исходный документ города.xml:

```
<города>
   <город название="Париж" страна="Франция"/>
   <город название="Рим" страна="Италия"/>
   <город название="Ницца" страна="Франция"/>
   <город название="Мадрид" страна="Испания"/>
   <город название="Милан" страна="Италия"/>
   <город название="Фиренце" страна="Италия"/>
   <город название="Неаполь" страна="Италия"/>
   <город название="Лион" страна="Франция"/>
   <город название="Барселона" страна="Испания"/>
</города>
```

Таблица стилей

Таблица стилей подсчет-стран.xsl. Это полная таблица стилей, созданная с помощью упрощенного синтаксиса таблиц стилей, описанного в главе 3. Происходит создание набора узлов, содержащего только те элементы <город>, значение атрибута страна которых отличается от всех предыдущих. Затем происходит подсчет узлов в наборе.

```
<итог
    xsl:version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:value-of
    select="count(//город[not(@страна=preceding::город/@страна)])"/>
</итог>
```

Вывод

```
<?xml version="1.0" encoding="utf-8"?>
<итог>3</итог>
```

current

Функция current() возвращает набор узлов, содержащий единственный узел, текущий.

Определена в

XSLT, раздел 12.4

Формат

`current() ⇒ набор узлов`

Аргументы

Отсутствуют.

Результат

Набор узлов, содержащий единственный узел, текущий.

Правила

Важно понимать разницу между текущим узлом и контекстным узлом.

Текущий узел определяется следующим образом:

- При вычислении глобальной переменной текущим узлом является корневой узел исходного документа.
- При использовании `<xsl:apply-templates>` для обработки определенного набора узлов каждый из узлов по очереди становится текущим. Таким образом, когда происходит применение шаблонного правила для определенного узла, это узел становится текущим. После того как элемент `<xsl:apply-templates>` отработан, происходит возвращение к предыдущему значению для текущего узла.
- Соответственно, когда происходит явный вызов шаблона для обработки корневого узла исходного документа, текущим узлом является корневой узел.
- Когда конструкция `<xsl:for-each>` применяется для обработки набора узлов, каждый из узлов по очереди становится текущим. После завершения цикла `<xsl:for-each>` восстанавливается предыдущее значения текущего узла.
- Выполнение `<xsl:call-template>` и `<xsl:apply-imports>` не изменяет значение текущего узла.
- При вычислении предиката в выражении пути текущий узел (в отличие от контекстного) не изменяется.

Контекстный узел – это узел, возвращаемый выражением XPath «.». При самостоятельном использовании XPath-выражения «`current()`» и «.» возвращают одинаковые результаты. Но при использовании этих выражений в предикате результаты, как правило, различаются.

Функция `current()` не может использоваться в образце. Решение о том, соответствует ли узел образцу, должно быть контекстно-независимым, то есть не зависеть от обстоятельств, в которых происходит вычисление образца.

Применение и примеры

Смысл существования функции current() в том, чтобы можно было определить текущий узел в случаях, когда он отличается от контекстного узла – в частности, в пределах предиката. Контекстный узел всегда может быть определен с помощью выражения пути «.» (или его полной формы, «self::node()»).

Эффективное применение функции current() чаще всего приходится на ситуации, в которых существует необходимость совершить переход по перекрестной ссылке от текущего узла к другому узлу. К примеру, предположим, что в книжном каталоге у элемента <книга> существует атрибут категория, принимающий в качестве значения код, вроде "КЛ" или "НФ", и что существует справочная таблица, в которой содержится расшифровка этих кодов. Скажем, кодом КЛ может обозначаться классическая литература, а кодом НФ – научная фантастика.

Пример: current()

В данном примере происходит перечисление книг из каталога; в описании каждой из книг содержится перечень всех остальных книг из той же категории.

Исходный документ

Исходный документ, книги.xml:

```
<книги>
<книга категория="Н">
     <название>Числа, язык науки</название>
     <автор>Данциг</автор>
</книга>
<книга категория="ДД">
     <название>Юные гости</название>
     <автор>Дейзи Эшфорд</автор>
</книга>
<книга категория="ДД">
     <название>Когда мы были очень молодыми</название>
     <автор>А. Милн</автор>
</книга>
<книга категория="И">
     <название>Паттерны проектирования</название>
     <автор>Эрих Гамма</автор>
     <автор>Ричард Хелм</автор>
     <автор>Ральф Джонсон</автор>
     <автор>Джон Влиссидес</автор>
</книга>
</книги>
```

Таблица стилей

Таблица стилей вывести-книги.xsl. Происходит обработка всех книг в цикле <xsl:for-each>, для каждой книги отображается название и имя первого автора. Для каждой книги происходит определение всех книг, входящих в ту же категорию. Для этого используется предикат «[./@категория = current()/@категория]», который истинен, если значение атрибута категория контекстного элемента совпадает со значением атрибута категория текущего элемента. Контекстный элемент – тот, для которого производится проверка, а текущий – тот, для которого происходит отображение. Помимо этого, результаты вызова generate-id() для этих двух элементов должны быть различными, что позволяет исключить дублирование. Второй способ проверить, что элементы различны, – с помощью записи «count(.|current())=2», поскольку объединение контекстного узла и текущего узла будет содержать единственный элемент, если они совпадают, и два, если они различаются. В этом случае можно сократить запись до «.!=current()», то есть проверить, различны ли строковые значения этих двух узлов, но этот тест может быть более требовательным к ресурсам, а сравнение производится на основании несколько иных свойств.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>
<xsl:template match="/">
  <html><body>
    <xsl:variable name="все-книги" select="//книга"/>
    <xsl:for-each select="$все-книги">
      <h1><xsl:value-of select="название"/></h1>
      <p><i>Автор </i><xsl:value-of select="автор"/>
         <xsl:if test="count(автор)!=1"> и другие</xsl:if>
      </p>
      <xsl:variable name="другие"
        select="$все-книги[./@категория=current()/@категория and
                           generate-id(.)!=generate-id(current())]"/>
        <xsl:if test="$другие">
          <p>Другие книги в этой категории:</p><ul>
          <xsl:for-each select="$другие">
             <li><xsl:value-of select="название"/></li>
          </xsl:for-each>
          </ul>
        </xsl:if>
    </xsl:for-each>
  </body></html>
</xsl:template>

</xsl:transform>
```

Вывод

```
<html>
   <body>
```

```
        <h1>Числа, язык науки</h1>
        <p><i>Автор </i>Данциг
        </p>
        <h1>Юные гости</h1>
    <p><i>Автор </i>Дейзи Эшфорд
        </p>
        <p>Другие книги в этой категории:</p>
        <ul>
    <li>Когда мы были очень молодыми</li>
        </ul>
        <h1>Когда мы были очень молодыми</h1>
        <p><i>Автор </i>А. Милн
        </p>
        <p>Другие книги в этой категории:</p>
        <ul>
           <li>Юные гости</li>
        </ul>
        <h1>Паттерны проектирования</h1>
        <p><i>Автор </i>Эрих Гамма и другие
        </p>
      </body>
    </html>
```

См. также

Раздел «СокращенныйШаг» главы 5

document

Функция `document()` производит поиск внешнего XML-документа путем разрешения URI-ссылки, производит разбор XML-данных в древовидную структуру, а затем возвращает корневой узел этого дерева. Функция может также использоваться для поиска набора внешних документов или для поиска узла, отличающегося от корневого, путем использования идентификатора фрагмента в URI.

К примеру, в выражении «`document('data.xml')`» происходит поиск файла `data.xml` в каталоге, где расположена таблица стилей, и анализ этого файла, после чего возвращается корневой узел конечного дерева.

Определена в

XSLT, раздел 12.1

Формат

`document(uri)` ⇒ набор узлов

`document(uri, базовый-uri)` ⇒ набор узлов

Аргументы

Аргумент	Тип данных	Смысл
uri	любой	Либо: (a) набор узлов исходного документа, строковые значения которых являются URI, либо (b) значение, которое будучи преобразованным в строку, может считаться URI.
базовый uri (необязательный)	набор узлов	Если данный аргумент присутствует, он должен являться набором узлов. Базовый URI первого узла из набора используется для разрешения всех относительных URI, присутствующих в первом аргументе функции.

Результат

Результатом вычисления всегда является набор узлов.

Действие

Ниже приводятся правила, советы по применению, а также примеры – по отдельному набору на каждую комбинацию аргументов.

Если вкратце, функция `document()` производит поиск XML-документа, используя URI. Полученный XML-документ подвергается анализу, создается дерево данных. Удаляются узлы, содержащие только пробельные символы, в соответствии с правилами, определенными для исходного документа, на основе действующих объявлений `<xsl:strip-space>` и `<xsl:preserve-space>`. Эта процедура применяется даже к документам, которые являются таблицами стилей. Результатом вызова функции `document()` является набор узлов, содержащий корневой узел нового документа.

Но прежде всего обратимся к URI и URL, терминам, которые достаточно свободно используются в данном разделе.

Задание базового URI

В спецификации XSLT постоянно используется термин URI: Uniform Resource Identifier (унифицированный идентификатор ресурса). Концепция URI-идентификаторов является обобщением механизма URL-идентификаторов (Uniform Resource Locators, унифицированные указатели ресурсов), которые в настоящее время получили широкое применение в веб-пространстве и присутствуют даже на упаковках с кукурузными хлопьями. Идея URI заключается в расширении механизма URL, который основан на установившейся системе доменных имен (имена в которой, скажем, *www.ibm.com* или *www.cam.ac.uk*, имеют иерархическую структуру). URI позволяют использовать другие системы именования или перечисления, включая и существующие, вроде ISBN-номеров книг и международных телефонных номеров. Но несмотря на всю привлекательность этой идеи, в настоящее время работают

лишь уже известные нам URL-идентификаторы. Поэтому кажется, что термины URI и URL взаимозаменяемы в контексте этого раздела, да и всей книги, впрочем, тоже. Но, читая внимательно, вы поймете, что я стараюсь корректно использовать оба термина.

Спецификация XSLT оставляет решения, связанные с поддержкой конкретных URI-схем, на усмотрение разработчиков конкретных реализаций. Краткосрочная перспектива такова, что в большинстве реализаций XSLT-процессоров будут поддерживаться только традиционные URL. Тем не менее, некоторые из них (включая любые с поддержкой TrAX API, который описан в приложении F) могут предоставлять возможность использовать свой собственный дополнительный код для реализации работы с любыми видами URI. В этом случае связь URI, использованного при вызове, и документа, который возвращается дополнительным кодом, целиком определяется только этим дополнительным кодом.

URI, используемый при вызове функции `document()`, должен указывать на XML-документ. В случае некорректного URI либо идентификатора, ссылающегося на несуществующий XML-документ, реализация XSLT-процессора может поступать по своему усмотрению (как рекомендуется спецификацией): сообщить об ошибке либо вернуть пустой набор узлов.

В спецификации не сказано, должен ли быть действительным собственно сам XML-документ, но в большинстве реализаций существует какой-то способ контролировать использование проверяющего анализатора.

URI может быть не только абсолютным, но и относительным. Типичный пример относительного URI – это имя `data.xml`. Такой идентификатор разрешается (то есть преобразуется в абсолютный, уникальный в глобальном контексте URI) путем интерпретации его относительно определенного базового URI. По умолчанию относительный URI в тексте XML-документа интерпретируется относительно URI-идентификатора документа (или внешней XML-сущности), его содержащего; в случае с функцией `document()` это обычно исходный документ либо таблица стилей. Так, если в исходном документе присутствует относительный URI `data.xml`, XSLT-процессор попытается найти этот файл в том же каталоге, который содержит исходный документ, а если идентификатор присутствует в таблице стилей – тогда в каталоге, содержащем эту таблицу стилей. Однако функция `document()` имеет второй аргумент, который позволяет, при необходимости, явным образом задавать базовый URI.

Расширение относительных URI использует тот факт, что в модели дерева XSLT, которая была описана в разделе «Древовидная модель» главы 2, у каждого из узлов есть базовый URI. (Не путайте его с URI пространства имен, это отдельный разговор.) По умолчанию базовым URI узла исходного документа или таблицы стилей является URI XML-документа или сущности, из которой был создан узел. В некоторых случаях, скажем, когда источником данных служит DOM-документ, процессор может не определить базовый URI, поскольку такое понятие отсутствует в стандарте DOM. События в подобной ситуации зависят целиком от реализации процессора. Компания

Microsoft, процессор которой (MSXML3) основан на собственной реализации DOM, расширила этот стандарт с целью сохранения URI, откуда был загружен документ.

В XSLT 1.1 существует возможность обойти стандартные правила определения базового URI узла с помощью атрибута `xml:base`. Этот атрибут определен в Рекомендации W3C, которая носит название XML Base; он несет ту же функциональную нагрузку, что и элемент `<base>` в HTML. Если элемент содержит атрибут `xml:base`, значение атрибута должно быть URI, и этот идентификатор определяет базовый URI для самого элемента и для всех потомков узла элемента, которые не имеют собственного атрибута `xml:base`.

URI, определяемый в `xml:base`, вполне может быть относительным URI, и в этом случае происходит его разрешение относительно базового URI родительского узла (то есть URI, который считался бы для элемента базовым, если бы этот элемент не имел атрибута `xml:base`).

В XSLT 1.1 узел, использованный для задания базового URI для функции `document()`, может быть узлом временного дерева, созданного в качестве значения переменной. Как правило, базовым URI для такого узла будет базовый URI элемента `<xsl:variable>` (либо `<xsl:param>`, либо `<xsl:with-param>`), который определяет временное дерево. И опять же, если элемент имеет атрибут `xml:base`, этот атрибут задает базовый URI – так же, как и для любого исходного документа.

Если в нескольких вызовах функции `document()` используется один и тот же URI (после разрешения относительного URI в абсолютный), каждый раз возвращается один и тот же корневой узел. Можно определить, что возвращается один и тот же корневой узел, с помощью функции `generate-id()`, которая будет возвращать одинаковый результат, и функции `count()`, которая определит дублирующиеся узлы. К примеру, результатом вычисления «`count(document("a.xml") | document("a.xml"))`» должна быть единица (`1`).

Идентификатор фрагмента указывает на фрагмент ресурса. Так, в URL *http://www.wrox.com/booklist#april2001* идентификатором фрагмента является подстрока «`april2001`». В принципе, идентификатор фрагмента позволяет с помощью URI ссылаться на произвольный узел или набор узлов целевого документа; то есть может быть, например, XPointer-выражением, содержащим сложную конструкцию для отбора узлов целевого документа. Но на практике мы сталкиваемся с особенностями реализаций. Интепретация частичного идентификатора зависит от типа среды (часто называемого MIME-типом) возвращаемого документа. Реализация не обязана поддерживать любой существующий тип среды (и даже не обязана поддерживать идентификаторы фрагментов вовсе). В документации процессора, соответствующего стандарту, должно быть указано, поддержка каких идентификаторов фрагментов реализована, но этим требованием довольно часто пренебрегают; мне практически не встречалась информация о поддержке идентификаторов фрагментов. (Однако в процессоре Saxon существует поддержка простых идентификаторов фрагментов – имен, являющихся значениями ID-атрибутов элементов целевого документа).

В следующих разделах описано поведение функции document() для каждой из существующих комбинаций аргументов.

document(набор-узлов)

Данный вариант включает и распространенный случай, когда аргумент является ссылкой на атрибут, к примеру «document(@href)».

Правила

В простом случае (скажем, «document(@href)») результатом вызова является набор узлов, состоящий из единственного узла, а именно – корневого узла документа, на который указывает атрибут href.

Говоря более формально, результат является объединением результатов вызовов функции document() для каждого узла *N* из поступившего на вход набора узлов. В этих вызовах функция document() получает два аргумента: строковое значение для узла *N* и набор узлов, в который входит только узел *N*. Это определение рекурсивно, чтобы понять его точный смысл, следует прочитать раздел «*document* (*строка, набор-узлов-2*)» на стр. 521.

Действительный же смысл таков: каждый из узлов исходного набора должен содержать URI в качестве своего строкового значения. Если URI относительный, он будет разрешен относительно базового URI данного узла. Базовый URI узла определяется в соответствии с правилами, которые приводятся на стр. 510. По сути, каждый узел исходного набора потенциально может иметь уникальный в пределах набора базовый URI.

Все это может показаться очень сложным, но на самом деле речь идет лишь о том, что если исходный документ содержит ссылку «data.xml», система будет искать файл data.xml в том же каталоге, где хранится исходный документ.

В наиболее распространенной ситуации, если речь идет о вызове вроде «document(@href)», передаваемый функции набор узлов содержит всего один узел, и результат является набором узлов из единственного узла, а именно – корневого узла документа, URI которого определяется значением атрибута href контекстного узла. При этом относительный URI разрешается на основе базового URI элемента, которому принадлежит атрибут.

Приведенные правила также применимы к случаю, когда аргумент является ссылкой на переменную, содержащую временное дерево; к примеру:

```
<xsl:variable name="индекс">индекс.xml</xsl:variable>
<xsl:for-each select="document($индекс)">
. . .
</xsl:for-each>
```

В данном случае относительный URI «индекс.xml» разрешается относительно базового URI элемента таблицы стилей <xsl:variable>.

Применение

Чаще всего функция document() используется для доступа к документу, ссылка на который присутствует в исходном документе, как правило, в ат-

рибуте вроде href. К примеру, книжный каталог может содержать ссылки на рецензии к каждой из книг в следующем формате:

```
<книга>
   <рецензия дата="28-12-1999" издание="New York Times"
             текст="рецензии/NYT/28121999/рец3.xml"/>
   <рецензия дата="06-01-2000" издание="Washington Post"
             текст="рецензии/WPost/06012000/рец12.xml"/>
</книга>
```

Если существует необходимость включить текст резенций в конечный документ, это можно сделать с помощью функции document(). К примеру, так:

```
<xsl:template match="книга">
   <xsl:for-each select="рецензия">
      <h2>Рецензия <xsl:value-of select="@издание"/></h2>
      <xsl:apply-templates select="document(@текст)"/>
   </xsl:for-each>
</xsl:template>
```

Поскольку аргумент @текст является набором узлов, результатом будет корневой узел документа, URI которого представлен значением атрибута текст и интерпретируется относительно базового URI элемента <рецензия>, который совпадает с URI исходного документа. Обратное может быть верно, если элемент <рецензия> принадлежит внешней XML-сущности либо находится в пределах влияния атрибута xml:base одного из элементов-предков.

Обратите внимание, что при обработке документа с рецензией будут использованы в точности те же правила, что и для исходного документа. Не существует возможности связать определенные шаблонные правила с конкретными типами документов. Если в документе с рецензией используются те же теги элементов, что и в книжном каталоге, но в ином смысле, могут возникнуть осложнения. Существует два способа решить проблему:

- Пространства имен: воспользоваться разными пространствами имен для книжного каталога и документов с рецензиями.
- Режимы: воспользоваться отдельным режимом для обработки узлов документа с рецензией. Приведенная выше инструкция <xsl:apply-templates> в этом случае будет выглядеть так:
  ```
  <xsl:apply-templates select="document(@текст)" mode="рецензия"/>
  ```

Может статься, что даже для случая непересекающихся наборов элементов применение режимов окажется полезным для улучшения читаемости таблицы стилей. Более подробно о режимах можно прочитать в разделах <xsl:apply-templates> и <xsl:template> в главе 4.

Другой полезный прием, позволяющий реализовать модульную структуру таблиц стилей, заключается во включении внешних таблиц стилей для обработки документов с рецензиями – с помощью элемента <xsl:include>.

Пример: Применение функции document() для анализа таблицы стилей

Таблица стилей является XML-документом, и может быть использована в качестве исходных данных для другой таблицы стилей. Поэтому очень легко создавать небольшие программы для манипуляции таблицами стилей. Такой инструмент мы создадим в этом примере: программу для создания отчета по иерархической структуре модулей таблицы стилей.

Функция `document()` используется для просмотра таблицы стилей и поиска модулей, включаемых с помощью `<xsl:include>` или `<xsl:import>`. Модули загружаются и обрабатываются рекурсивно.

Исходный документ

Подойдет любая таблица стилей, в которой используются элементы `<xsl:include>` или `<xsl:import>`. В качестве таковой можно использовать файл `dummy.xsl`, который входит в состав загружаемых исходных файлов для этой книги.

Таблица стилей

В таблице стилей `вывести-включаемые-файлы.xsl` функция `document()` используется для доступа к документам, ссылки на которые содержатся в атрибуте `href` элементов `<xsl:include>` и `<xsl:import>`. Эта же процедура применяется для включаемых модулей, рекурсивно. Заметим, корневой шаблон применяется только к начальному исходному документу – в целях создания скелета HTML-страницы.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:template match="/">
  <html><body>
    <h1>Структура модулей таблицы стилей</h1>
    <ul>
    <xsl:apply-templates select="*/xsl:include | */xsl:import"/>
    </ul>
  </body></html>
</xsl:template>

<xsl:template match="xsl:include | xsl:import">
     <li><xsl:choose>
       <xsl:when test="local-name()='import'">
         <xsl:value-of select="concat('импортирует ', @href)"/>
       </xsl:when>
       <xsl:otherwise>
         <xsl:value-of select="concat('включает ', @href)"/>
       </xsl:otherwise>
     </xsl:choose>
```

```
    <xsl:variable name="модуль" select="document(@href)"/>
    <ul>
    <xsl:apply-templates
        select="$модуль/*/xsl:include | $модуль/*/xsl:import"/>
    </ul>
    </li>
</xsl:template>

</xsl:transform>
```

Вывод

В качестве обрабатываемого файла использовался файл `dummy.xsl`:

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>
<xsl:include href="dummya.xsl"/>
</xsl:transform>
```

Включаемый файл `dummya.xsl` не намного содержательнее:

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:import href="dummyb.xsl"/>
<xsl:import href="dummyc.xsl"/>
</xsl:transform>
```

Импортируемые файлы `dummyb.xsl` и `dummyc.xsl` — это просто пустые файлы-заглушки:

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

</xsl:transform>
```

Вывод для таблицы стилей `dummy.xsl` имеет следующий вид (рис. 7.1):

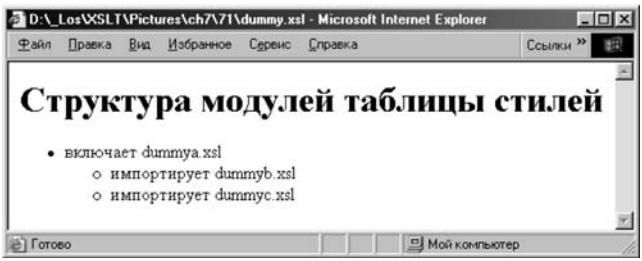


***Рис. 7.1.** Модульная структура dummy.xsl*

document(набор-узлов-1, набор-узлов-2)

Этот случай идентичен предыдущему, с той лишь разницей, что вместо использования самого узла для разрешения относительного URI используется базовый URI первого узла из набор-узлов-2. Другими словами, если узел набор-узлов-1 содержит относительный URL (вроде «data.xml»), система будет искать файл data.xml в каталоге, содержащем XML-документ, которому принадлежат узлы набора набор-узлов-2.

Правила

Чаще всего набор-узлов-2 состоит из единственного узла. К примеру, в вызове «document(@href, /)» корневой узел исходного документа будет использован в качестве базового URI, даже если элемент, которому принадлежит атрибут href, был извлечен из внешней сущности с иным URI-идентификатором.

Говоря более формально: результат является объединением результатов вызовов функции document() для каждого узла *N* из поступившего на вход набора узлов набор-узлов-1. В этих вызовах функция document() получает два аргумента: строковое значение для узла *N* и набор узлов набор-узлов-2. Конечно, это определение рекурсивно, чтобы понять его точный смысл, следует прочитать раздел «*document* (*строка, набор-узлов-2*)» на стр. 521.

Пустой набор набор-узлов-2 является ошибкой. XSLT-процессор сообщит о ней как о критической ошибке либо вернет в качестве результата пустой набор узлов.

Применение

Этот вариант не из тех, что часто встречаются на практике, но, тем не менее, он существует.

Чтобы интерпретировать URI относительно таблицы стилей, можно выполнить, к примеру, такой вызов:

```
document(@href, document(''))
```

Это работает, потому что второй аргумент возвращает корневой узел таблицы стилей, который затем используется для определения базового URI и разрешения относительного идентификатора, содержащегося в атрибуте href.

document(строка)

А этот вариант используется для единственного аргумента, который не является набором узлов. Наиболее распространенный случай – URL, жестко прописанный в таблице стилей, к примеру «document('ставки-налогов.xml')».

Наиболее распространенный особый случай – вызов «document('')», ссылающийся на сам документ таблицы стилей.

Правила

Если аргумент не является строкой, выполняется преобразование в соответствии с правилами для функции string(), – но поскольку преобразование

значения логического типа или числа не приведет к получению осмысленного URL, практического значения это преобразование не имеет.

Строка интерпретируется как URI. Если она содержит относительный URI, происходит разрешение идентификатора на основе базового URI того элемента таблицы стилей, который содержит выражение с вызовом функции. Обычно это URI основного документа таблицы стилей, но это может быть и не так, если использовался элемент <xsl:include> или <xsl:import> либо если части таблицы стилей содержатся во внешних XML-сущностях, либо (в случае XSLT 1.1) если базовый URI соответствующего элемента таблицы стилей был явным образом задан с помощью атрибута xml:base.

Опять же, речь идет лишь о том, что относительные URL обрабатываются так же, как относительные URL в HTML: если написать «document('ставки-налогов.xml')» в каком-либо модуле таблицы стилей, система будет искать файл ставки-налогов.xml в каталоге, где расположен этот модуль.

Если в качестве параметра передается пустая строка, используется документ, на который ссылается базовый URI. Спецификация XSLT говорит, что вызов «document('')» приводит к получению корневого узла таблицы стилей. Но, строго говоря, это верно только в том случае, когда базовый URI элемента, содержащего вызов функции document(), совпадает с системным идентификатором модуля таблицы стилей. Если базовый URI отличается, скажем, по той причине, что таблица стилей собирается из набора внешних сущностей, либо потому (в XSLT 1.1), что был использован атрибут xml:base, то объект, загружаемый вызовом «document('')», совершенно не обязательно будет являться текущим модулем таблицы стилей. По сути дела, этот объект может вообще не быть корректным, что приведет к возникновению ошибки.

Если вызов содержится в таблице стилей, включенной с помощью элемента <xsl:include> или <xsl:import>, возвращается корневой узел включенной или импортированной таблицы стилей, а не основного документа таблицы стилей.

Применение

В XSLT 1.0 этот вариант вызова функции document() весьма полезен для работы со справочными данными, которые могут использоваться таблицами стилей: скажем, для реализации таблиц соответствий для расшифровки сокращений, файлов сообщений на различных языках либо выбора текста приветствия, отображаемого на экране при входе в систему. Такие данные могут содержаться в самой таблице стилей (и извлекаться по ссылке «document('')») либо в самостоятельном файле, хранимом в том же каталоге, что и таблица стилей (по ссылке «document('сообщения.xml')»), или же в каталоге, который может быть определен относительным путем (например «document('../data/сообщения.xml')»).

Что касается XSLT 1.1 в существующей редакции, для этих целей нет необходимости использовать дополнительные документы, поскольку данные могут храниться в древовидных структурах в переменных таблицы стилей, и к ним существует прямой доступ. Но в некоторых случаях, тем не менее, может быть удобнее хранить данные в самостоятельном файле (скажем, если

данные периодически генерируются на основе базы данных) либо сделать таблицу стилей совместимой с XSLT-процессорами версии 1.0, в особенности, если предполагается, что таблицы стилей будут использоваться на стороне клиента. Поэтому сначала рассмотрим приемы для XSLT 1.0, а затем решим ту же задачу с помощью возможностей XSLT 1.1.

В XSLT внутри любого элемента верхнего уровня могут присутствовать данные, вроде таблиц соответствий, принадлежащие нестандартному пространству имен.

Пример: Таблица соответствий в таблице стилей

Исходный документ

Файл `книги.xml` мы уже использовали ранее:

```
<книги>
<книга категория="Н">
     <название>Числа, язык науки</название>
     <автор>Данциг</автор>
</книга>
<книга категория="ДД">
     <название>Юные гости</название>
     <автор>Дейзи Эшфорд</автор>
</книга>
<книга категория="ДД">
     <название>Когда мы были очень молодыми</название>
     <автор>А. Милн</автор>
</книга>
<книга категория="И">
     <название>Паттерны проектирования</название>
     <автор>Эрих Гамма</автор>
     <автор>Ричард Хелм</автор>
     <автор>Ральф Джонсон</автор>
     <автор>Джон Влиссидес</автор>
</книга>
</книги>
```

Таблица стилей

Файл таблицы стилей называется `вывести-категории.xsl`. Для каждого из элементов `<книга>` исходного документа в таблице стилей производится поиск элемента `<книга:категория>`, атрибут `код` которого совпадает с атрибутом `категория` данного элемента `<книга>`. Обратите внимание на использование функции `current()` для ссылки на текущую книгу. Ошибочно будет использовать конструкцию «.», поскольку в данном случае «.» указывает на контекстный узел, то есть на очередной элемент `<книга:категория>`.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
 xmlns:книга="книги.uri"
```

```
 exclude-result-prefixes="книга"
>

<xsl:template match="/">
  <html><body>
    <xsl:for-each select="//книга">
<h1><xsl:value-of select="название"/></h1>
       <p>Категория: <xsl:value-of
              select="document('')/*/книга:категория
              [@код=current()/@категория]/@описание"/>
       </p>
    </xsl:for-each>
  </body></html>
</xsl:template>

<книга:категория код="Н" описание="наука"/>
<книга:категория код="И" описание="информатика"/>
<книга:категория код="ДД" описание="литература для детей"/>

</xsl:transform>
```

Вывод

```
<html>
   <body>
      <h1>Числа, язык науки</h1>
      <p>Категория: наука</p>
      <h1>Юные гости</h1>
      <p>Категория: литература для детей</p>
      <h1>Когда мы были очень молодыми</h1>
      <p>Категория: литература для детей</p>
      <h1>Паттерны проектирования</h1>
      <p>Категория: информатика</p>
   </body>
</html>
```

Таблица стилей XSLT 1.1

Теперь модифицируем таблицу стилей и воспользуемся преимуществами XSLT 1.1. Переименуем файл в `вывести-категории1_1.xsl`**. Содержательные различия будут невелики – измененные строки отмечены затененным фоном.**

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.1"
 xmlns:book="книги.uri"
 exclude-result-prefixes="книга"
>

<xsl:template match="/">
  <html><body>
    <xsl:for-each select="//книга">
       <h1><xsl:value-of select="название"/></h1>
       <p>Категория: <xsl:value-of
```

```
            select="$категории/категория
            [@код=current()/@категория]/@описание"/>
          </p>
        </xsl:for-each>
      </body></html>
    </xsl:template>

    <xsl:variable name="категории">
      <категория код="Н" описание="наука"/>
      <категория код="И" описание="информатика"/>
      <категория код="ДД" описание="литература для детей"/>
    </xsl:variable>

    </xsl:transform>
```

document(строка, набор-узлов-2)

Данный формат вызова используется для двух аргументов, первый из которых **не является** набором узлов. (В данном случае даже в XSLT 1.0 временное дерево считается набором узлов.)

Правила

Если первый аргумент не является строкой, выполняется преобразование в соответствии с правилами для функции `string()`. Но поскольку это происходит только в случае значения логического типа или числа, вряд ли такой вариант может встретиться на практике.

Второй аргумент, как правило, является набором, состоящим из единственного узла: если узлов больше, принимается во внимание только первый узел (по порядку следования в документе). Поведение XSLT-процессора в случае пустого набора узлов не определено.

Строка интерпретируется в качестве URI. Если идентификатор является относительным, разрешение происходит на основе базового URI первого узла из набора `набор-узлов-2`.

Применение

Маловероятно, что этот вариант будет часто встречаться на практике, но имеет смысл знать и о нем. В XSLT 1.1 столь же удобно просто указывать базовый URI с помощью атрибута `xml:base`.

Примечание: использование ключей и идентификаторов в дополнительных документах

Функции `key()` и `id()` всегда возвращают узлы из того документа, которому принадлежит контекстный узел. Невозможно извлечь узлы из другого документа с помощью такого выражения:

```
<!--НЕВЕРНО-->
<xsl:value-of select="document('a.xml')/key('k1', 'val1')"/>
<!--НЕВЕРНО-->
```

Вместо этого в качестве наиболее удобного способа достижения желаемого можно предложить смену контекстного узла с помощью элемента <xsl:for-each>:

```
<xsl:for-each select="document('a.xml')">
   <xsl:value-of select="key('k1', 'val1')"/>
</xsl:for-each>
```

См. также

id() на стр. 545
key() на стр. 548

element-available

Эта функция позволяет определить, доступна ли для использования определенная инструкция XSLT или элемент расширения.

К примеру, результатом вычисления выражения «element-available('xsl:text')» является значение истина.

Определена в

XSLT, раздел 15

Формат

element-available(имя) ⇒ логическое значение

Аргументы

Аргумент	Тип данных	Смысл
имя	строка	Имя элемента, для которого производится проверка. Если аргумент не является строкой, его значение преобразуется в строку в соответствии с правилами, определенными для функции string(). Конечная строка должна являться полным именем.

Результат

Значение логического типа: истина, если указанный элемент доступен для использования в качестве инструкции в шаблоне, и ложь в противном случае.

Правила

Первый аргумент должен являться полным именем, то есть именем XML с необязательным префиксом пространства имен, который соответствует объявлению пространства имен, находящемуся в области действия на момент вызова функции element-available().

Если объявление пространства имен связано с пространством имен XSLT *http://www.w3.org/1999/XSL/Transform*, функция возвращает истину, если имя является определенной в XSLT *инструкцией*, и ложь в противном случае. В спецификации XSLT версии 1.0 определены следующие инструкции:

```
<xsl:apply-imports>
<xsl:apply-templates>
<xsl:attribute>
<xsl:call-template>
<xsl:choose>
<xsl:comment>
<xsl:copy>
<xsl:copy-of>
<xsl:element>
<xsl:fallback>
<xsl:for-each>
<xsl:if>
<xsl:message>
<xsl:number>
<xsl:processing-instruction>
<xsl:text>
<xsl:value-of>
<xsl:variable>
```

В Рабочем проекте стандарта XSLT 1.1 к списку добавлена одна новая инструкция, а именно <xsl:document>.

Инструкции являются элементами XSLT, которые могут непосредственно присутствовать в теле шаблона. Элементы XSLT верхнего уровня, такие как <xsl:template> и <xsl:key>, не являются инструкциями, поэтому для них должно возвращаться значение ложь (тем не менее, следует проявлять осторожность: по крайней мере один из распространенных процессоров, Microsoft MSXML3, похоже, возвращает значение истина для всех элементов XSLT). То же справедливо и для элементов <xsl:param>, <xsl:with-param>, <xsl:sort>, <xsl:when> и <xsl:otherwise>, которые могут присутствовать только в определенных контекстах, а не в произвольной точке шаблона.

Наличие возможности создавать несколько конечных документов, которая появилась в XSLT 1.1, можно проверить с помощью вызова element-available('xsl:document'). Если элемент не существует, можно воспользоваться альтернативным подходом: к примеру, применить фирменный синтаксис, реализованный для этих целей в используемом процессоре XSLT 1.0 производителем.

Проиллюстрируем сказанное выше следующим примером:

Пример: Создание нескольких конечных документов

В качестве исходного документа используется стихотворение, каждая из строф которого записывается в отдельный конечный файл. Более актуальным примером было бы разделение книги на главы, но я старался минимизировать размеры используемых файлов. Этот пример будет работать с процессором Saxon, а также с любым процессором, который реализует Рабочий проект спецификации XSLT 1.1.

Исходный документ

Исходный документ стих.xml. Вот его начало:

```
<стихотворение>
```

```
<автор>Руперт Брук</автор>
<дата>1912</дата>
<название>Song</название>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
<строфа>
<строка>My heart all Winter lay so numb,</строка>
<строка>The earth so dead and frore,</строка>
. . .
```

Таблица стилей

Таблица стилей разделить.xsl. Обратите внимание на инструкцию `<xsl:choose>`, в каждой ветви которой происходит вызов `element-available()` для определения доступности конкретной инструкции перед ее использованием.

Заметим, что строка «saxon» определена как префикс элемента расширения, так что элемент `<saxon:output>` является инструкцией. Элемент `<xsl:stylesheet>` содержит атрибут «version="1.1"», поскольку в противном случае процессор XSLT 1.0 посчитает конструкцию `<xsl:document>` ошибочной.

```
<?xml version="1.0"?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.1">

<xsl:template match="стихотворение">
    <стихотворение>
        <xsl:copy-of select="название"/>
        <xsl:copy-of select="автор"/>
        <xsl:copy-of select="дата"/>
        <xsl:apply-templates select="строфа"/>
    </стихотворение>
</xsl:template>
<xsl:template match="строфа">
    <xsl:variable name="файл"
select="concat('строфа', position(), '.xml')"/>
    <строфа номер="{position()}" href="{$файл}"/>
    <xsl:choose>
    <xsl:when test="element-available('xsl:document')">
        <xsl:document href="{$файл}">
            <xsl:copy-of select="."/>
        </xsl:document>
    </xsl:when>
    <xsl:when test="element-available('saxon:output')"
            xmlns:saxon="http://icl.com/saxon">
```

```
        <saxon:output file="{$file}"
            xsl:extension-element-prefixes="saxon">
            <xsl:copy-of select="."/>
        </saxon:output>
    </xsl:when>
    <xsl:otherwise>
        <xsl:message terminate="yes">
Возможность записи в несколько файлов недоступна.
        </xsl:message>
    </xsl:otherwise>
    </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Вывод

Если используется XSLT-процессор, реализующий требуемую функциональность, основной файл результата будет содержать основу стихотворения (для удобочитаемости произведено разбиение на строки):

```
<?xml version="1.0" encoding="utf-8" ?>
<стихотворение>
<название>Song</название>
<автор>Rupert Brooke</автор>
<дата>1912</дата>
<строфа номер="1" href="строфа1.xml"/>
<строфа номер="2" href="строфа2.xml"/>
<строфа номер="3" href="строфа3.xml"/>
</стихотворение>
```

Три последующих конечных документа, `строфа1.xml`, `строфа2.xml` и `строфа3.xml`, создаются в текущем каталоге. Содержимое файла `строфа1.xml`:

```
<?xml version="1.0" encoding="utf-8" ?>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
```

Функцию `element-available()` невозможно использовать для проверки доступности элемента XSLT 1.1 `<xsl:script>`, поскольку это элемент верхнего уровня, и честный процессор XSLT 1.1 должен возвращать значение `ложь`, в точности, как и любой процессор XSLT 1.0. Если использовать этот элемент в таблице стилей XSLT 1.0, в случае указания атрибута «`version="1.0"`» для элемента `<xsl:stylesheet>` возникнет ошибка, но если указать атрибут «`version="1.1"`», то элемент `<xsl:script>` будет проигнорирован, по правилам обработки в режиме совместимости с последующими версиями. Если при этом использовать альтернативный, определенный разработчиком системы спо-

соб выбора функций расширения, таблица стилей, вероятно, будет корректно работать для процессоров с поддержкой как XSLT 1.0, так и XSLT 1.1.

Если префикс полного имени указывает на произвольное пространство имен, кроме пространства имен XSLT, функция возвращает значение истина только тогда, когда процессор реализует указанную инструкцию: то есть если элемент может быть использован в качестве инструкции шаблона и не будет интерпретирован как конечный литеральный элемент.

Заметим, что результат вызова функции element-available() не зависит от того, было ли пространство имен объявлено как пространство имен для элементов расширения с помощью атрибута [xsl:]extension-element-prefixes. Если XSLT-процессор предоставляет реализацию указанной инструкции, функция обязана вернуть значение истина вне зависимости от того, в каком пространстве имен находится данная инструкция.

Если указано полное имя без префикса, используется пространство имен по умолчанию. Данный случай – один из немногих, когда такое происходит. Дело в том, что это имя всегда является именем элемента: в пространство имен по умолчанию входят только элементы.

Но если указанное имя принадлежит пространству имен с пустым URI, результатом выполнения функции будет ложь. Причина в следующем: инструкции и элементы расширения XSLT должны принадлежать пространствам имен с непустыми URI.

Применение

Функцию можно использовать для определения доступности элементов, появившихся в более поздних версиях XSLT, а также для определения доступности расширений, реализованных поставщиком или сторонним разработчиком.

Проверка доступности возможностей из более поздних версий XSLT

Возможность проверить, доступна ли определенная инструкция XSLT, появилась в XSLT 1.0, но стала полезной только теперь, с изданием версии 1.1 стандарта. Предполагалось, что она будет востребована, когда в новых версиях спецификации появятся новые инструкции. Как мы видим, в варианте XSLT 1.1 новых инструкций немного, одна лишь <xsl:document>. (Вспомним, что <xsl:script> инструкцией не является, поскольку не может использоваться в теле шаблона.) Если есть необходимость использовать инструкцию вроде <xsl:document>, которая появилась в конкретной версии XSLT, следует проверить ее доступность в выбранной реализации XSLT-процессора. Если инструкция недоступна, можно либо воспользоваться <xsl:if> и избежать выполнения этой инструкции, либо применить механизм <xsl:fallback>, чтобы справиться с ее отсутствием.

Но почему функция element-available() вообще появилась в составе XSLT версии 1.0? Если вдуматься, то ответ очевиден. Предположим, что необходимо создать таблицу стилей, в которой используются возможности версии 2.0, и мы используем element-available(), чтобы обеспечить корректное завершение в случае работы с процессорами, которые поддерживают только возможности версий 1.0 и 1.1. Но мы сможем это сделать только в том случае, если процессоры версий 1.0 и 1.1 поддерживают функцию element-available(). Именно поэтому такая функция была создана с самого начала. Неожиданно прозорливое решение со стороны разработчиков XSLT, которые старались избежать проблем совместимости, преследующих HTML. Разумеется, это по-прежнему приведет к созданию и отладке ветвящегося кода, но, по крайней мере, существует возможность создавать таблицы стилей, работающие с процессорами разного уровня.

В принципе можно проверять и доступность инструкции из версии 1.0, так как могут существовать процессоры, реализующие лишь подмножество стандарта XSLT 1.0; но это будет работать только в том случае, если реализация подмножества включает функцию element-available(). Но гарантий нет, поскольку она входит в разряд вещей, которые разработчики обычно оставляют на потом.

Предположим, что мы создадим таблицу стилей, в которой используются возможности XSLT версии 2.0. Чтобы выполнить код таблицы в процессоре XSLT 1.0, следует указать атрибут «version="2.0"» для элемента <xsl:stylesheet> либо «xsl:version="2.0"» для одного из конечных литеральных элементов даже в том случае, если <xsl:if> записывается с применением element-available(), чтобы избежать выполнения соответствующего кода. Если указать атрибут «version="1.0"», любое использование новых элементов XSLT 2.0 приведет к ошибке, даже если код не выполняется.

Расширения от разработчиков

Второе применение функции связано с проверкой доступности расширений от поставщика системы или сторонних разработчиков. Если известно, что определенный элемент существует в некоторых реализациях, можно использовать функцию element-available(), чтобы определить его наличие во время выполнения кода таблицы стилей и, опять же, обработать ситуацию его отсутствия с помощью элемента <xsl:if> или <xsl:fallback>.

К примеру, в процессоре Saxon существует элемент расширения, позволяющий вывести ссылку на сущность. Если используется другой процессор, эту задачу можно решить с помощью атрибута disable-output-escaping. Таким образом, чтобы напечатать « », используем:

```
<xsl:choose xmlns:saxon="http://icl.com/saxon">
<xsl:when test="element-available('saxon:entity-ref')">
    <saxon:entity-ref name="nbsp"
          xsl:extension-element-prefixes="saxon"/>
</xsl:when>
<xsl:otherwise>
```

```
      <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
   </xsl:otherwise>
   </xsl:choose>
```

Альтернативой использованию функции element-available() является использование механизма <xsl:fallback>, который описан в главе 4: элемент <xsl:fallback> позволяет определить действия, выполняемые в случае, когда инструкция, содержащая его, недоступна. Эти два механизма, по сути дела, равноценны, хотя у <xsl:fallback> есть потенциальное ограничение – он может использоваться только с элементами, у которых могут существовать потомки. Так, в существующем стандарте откат не может использоваться в элементе <xsl:copy-of>.

Примеры

В приведенном ниже фрагменте кода определен шаблон, который может вызываться с целью установки отладочной точки останова. Его функциональность зависит от гипотетической инструкции <xsl:breakpoint>, которая появилась в XSLT версии 37.1. Если инструкция не поддерживается процессором, она будет проигнорирована.

```
<xsl:stylesheet
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          version="38.0">
<xsl:template name="установить-точку-останова">
   <xsl:if test="element-available('xsl:breakpoint')">
      <xsl:breakpoint/>
   </xsl:if>

</xsl:template>
</xsl:stylesheet>
```

Заметим, что если установить атрибут version элемента <xsl:stylesheet> в значение "1.0", таблица стилей будет отвергнута. Если version принимает любое другое значение, включается режим совместимости с последующими версиями, и реализация процессора уже не считает <xsl:breakpoint> ошибочной инструкцией, если действительного выполнения инструкции не происходит. (В XSLT 1.1 режим совместимости с последующими версиями включается, если значение версии отличается от «1.0» и «1.1».) Режим совместимости с последующими версиями описан в соответствующем разделе главы 3.

См. также

function-available() на стр. 535

Раздел <xsl:fallback> главы 4

false

Функция возвращает логическое значение ложь.

Определена в

XPath, раздел 4.3

Формат

`false() ⇒ логическое значение`

Аргументы

Отсутствуют

Результат

Значение логического типа `ложь`.

Применение

В выражениях XPath не могут использоваться логические константы, но в качестве замены доступны функции `true()` и `false()`.

На практике логические константы нужны не очень часто. В основном они применяются при передаче параметров шаблону.

Запись «`<xsl:if test="false()">`» может быть полезна в качестве способа временно закомментировать фрагмент кода в таблице стилей. Комментарии XML не очень подходят в этом случае, потому что не могут быть вложенными.

Пример

В следующем фрагменте кода происходит вызов именованного шаблона, параметр «`подробный-отчет`» устанавливается в значение `ложь`:

```
<xsl:call-template name="выполнить-работу">
   <xsl:with-param name="подробный-отчет" select="false()"/>
</xsl:call-template>
```

См. также

`true()` на стр. 613

floor

Функция `floor()` возвращает ближайшее целое значение, которое меньше либо равно численному значению аргумента.

К примеру, выражение «`floor(11.3)`» возвращает значение 11.

Определена в

XPath, раздел 4.4

Формат

`floor(значение) ⇒ число`

Аргументы

Аргумент	Тип данных	Смысл
значение	число	Исходное значение. Если аргумент имеет тип, отличный от числового, он преобразуется в соответствии с правилами, определенными для функции `number()`.

Результат

Целочисленное значение: результат преобразования первого аргумента в число с последующим округлением до ближайшего меньшего целого.

Правила

Если аргумент не является числом, он, прежде всего, преобразуется в число. Правила преобразования подробно рассмотрены в описании функции `number()` на стр. 580. Если аргумент является набором узлов, правила применяются к значению первого, в порядке следования в документе, узла из набора.

Если аргумент является целым числом, он возвращается без изменений. В противном случае происходит округление до ближайшего меньшего целого.

Если значением аргумента является не-число – а это может произойти, если исходная строка не поддается преобразованию в число, – результатом будет также не-число. То же верно для отрицательной и положительной бесконечности – аргумент возвращается без изменений. Подробности, связанные с этими специальными числами, приведены в разделе «Типы данных» главы 2.

Применение и примеры

Проиллюстрируем работу функции следующими примерами:

```
floor(1.0) = 1.0
floor(1.6) = 1.0
floor(17 div 3) = 5.0
floor(-3.0) = -3.0
floor(-8.2) = -9.0
```

См. также

`ceiling()` на стр. 497

`round()` на стр. 586

format-number

Функция `format-number()` применяется для преобразования чисел в строки с целью отображения. Она также полезна для преобразования в более старые форматы, в которых число должно иметь фиксированную длину. Формат результата можно изменять с помощью элемента `<xsl:decimal-format>`.

К примеру, выражение «`format-number(12.5, '$#.00')`» возвращает строку «`$12.50`».

Определена в

XSLT, раздел 12.3

Функциональность этой функции определена с помощью ссылки на спецификацию Java JDK 1.1; в этой книге автор извлек из нее всю необходимую информацию.

Формат

```
format-number(значение, формат)  ⇒  строка
format-number(значение, формат, имя)  ⇒  строка
```

Аргументы

Аргумент	Тип данных	Смысл
значение	число	Исходное значение. Если аргумент имеет тип, отличный от численного, он преобразуется в соответствии с правилами, определенными для функции `number()`.
формат	строка	Шаблон формата. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции `string()`.
имя (необязательный)	строка	Имя (полное) десятичного формата записи, определенного с помощью элемента `<xsl:decimal-format>`. По умолчанию используется стандартный десятичный формат.

Результат

Строковое значение. Результат форматирования первого аргумента по шаблонному образцу формата (из второго аргумента) и либо применения правил, определенных для десятичного формата, указанного необязательным третьим аргументом, либо правил для стандартного десятичного формата.

Правила

Имя десятичного формата

Третий аргумент, если он указан, должен иметь вид полного имени; то есть имени XML с необязательным префиксом пространства имен, соответствую-

щим объявлению пространства имен, находящемуся в области действия на момент вызова функции format-number(). В таблице стилей должен существовать элемент <xsl:decimal-format> с таким же полным именем, причем при сравнении используются URI, а не префиксы.

Если третий аргумент отсутствует, то используется стандартный десятичный формат. Определить стандартный формат для таблицы стилей можно с помощью элемента <xsl:decimal-format> без имени. Если такой элемент в таблице стилей отсутствует, система использует встроенный стандартный формат.

Шаблон формата

Шаблон формата в стандарте XSLT описан при помощи ссылки на спецификацию Java JDK 1.1.

Ниже описана структура шаблона с использованием тех же синтаксических обозначений, что и в главе 5 «Выражения»:

pattern	subpattern (pattern-separator subpattern)?
subpattern	prefix? integer (decimal-point fraction)? suffix?
prefix	[#x0 - #xFFFD] - specialCharacters
suffix	[#x0 - #xFFFD] - specialCharacters
integer	digit* zero-digit* zero-digit (grouping-separator также может присутствовать)
fraction	zero-digit* digit*
pattern-separator	«;» (по умолчанию)
decimal-point	«.» (по умолчанию)
grouping-separator	«,» (по умолчанию)
digit	«#» (по умолчанию)
zero-digit	«0» (по умолчанию)
specialCharacters	см. таблицу ниже

В этих правилах символы «;», «.», «,», «#» и «0» являются умолчаниями для разделителей pattern-separator, decimal-point, grouping-separator, digit и zero-digit. Элемент <xsl:decimal-format> может использоваться для определения других символов в качестве разделителей.

Первый частичный шаблон (subpattern) определяет положительные числа. Второй (необязательный) частичный шаблон определяет отрицательные числа.

Используются следующие специальные символы:

Специальный символ	Смысл
zero-digit (по умолчанию «0»)	В этой позиции строки результата всегда присутствует цифра.

Специальный символ	Смысл
`digit` (по умолчанию «#»)	В этой позиции строки результата всегда присутствует цифра, если только она не является отбрасываемым начальным или завершающим нулем.
`decimal-point` (по умолчанию «.»)	Разделитель целой и дробной частей числа.
`grouping-separator` (по умолчанию «,»)	Разделитель групп цифр.
`pattern-separator` (по умолчанию «;»)	Разделитель частичных шаблонов для положительных и отрицательных форматов.
`minus-sign` (по умолчанию «-»)	Знак «минус».
`percent-sign` (по умолчанию «%»)	Умножает число на 100 и отображает его в процентах.
`per-mille` (по умолчанию «‰»)	Умножает на 1000 и отображает в тысячных долях.
`apostrophe` («'»)	Экранирует специальные символы, чтобы вернуть им привычные значения. Чтобы отобразить цифру «9» в формате «#9», следует использовать шаблон «'#'0».

В случае отсутствия отрицательного частичного шаблона символ «-» предваряет положительную форму числа. То есть самостоятельная строка «0.00» эквивалента «0.00;-0.00». Если присутствует явно заданный отрицательный частичный шаблон, он указывает только отрицательный префикс и суффикс; число цифр, минимумы и прочие характеристики совпадают с характеристиками, определенными для положительного частичного шаблона. То есть смысл шаблона «#,##0.0#;(#)» такой же, как у «#,##0.0#;(#,##0.0#)».

Стандарт XSLT определяет шаблоны посредством ссылки на спецификацию JDK 1.1. Некоторые дополнительные специальные символы были определены уже после оригинальной публикации спецификации JDK 1.1: в частности «¤» (международное обозначение символа валюты #xA4) и «E», символ экспоненты, который используется при выводе числа в экспоненциальном представлении. В спецификации четко сказано, что XSLT-процессор не обязан поддерживать эти специальные символы, но не сказано, что процессор должен считать эти символы ошибочными. Если используется процессор, написанный на языке Java, весьма вероятно, что интерпретация этих символов будет меняться в зависимости от используемой виртуальной машины Java.

Разделители груп используются, как правило, для образования групп из трех цифр, но в некоторых странах используется альтернативная группировка – по четыре цифры. Число цифр в группе в конечной строке равно числу цифр шаблона, заключенных между последним разделителем групп и окончанием целого числа. Все прочие разделители групп в шаблоне игнорируются. Шаблон «#,##,###,####» подразумевает отображение разделителя групп после каждых четырех цифр.

Поведение системы в случае поступления некорректного шаблона формата не определено. Это означает, что реализация может либо сообщить об ошибке, либо отобразить число в заранее определенном стандартном виде.

Применение

Заметим, что этот инструмент форматирования чисел реализуется совершенно отдельно от механизмов элемента <xsl:number>. Функциональность частично перекрывается, но синтаксис шаблонов формата отличается. Функция format-number() производит форматирование одного числа, которое не должно быть целым. Основное предназначение <xsl:number> – форматирование списков целых чисел. Для форматирования одного числа можно использовать любой из механизмов.

Примеры

В следующей таблице сведены примеры использования функции format-number() со стандартным десятичным форматом. Примеры для нестандартных десятичных форматов приведены в описании элемента <xsl:decimal-format> в главе 4.

Число	Шаблон формата	Результат
1234.5	#,##0.00	1,234.50
123.456	#,##0.00	123.46
1000000	#,##0.00	1,000,000.00
-59	#,##0.00	-59.00
1 div 0	#,##0.00	Infinity
1234	###0.0###	1234.0
1234.5	###0.0###	1234.5
.00035	###0.0###	0.0004
0.25	#00%	25%
0.736	#00%	74%
1	#00%	100%
-42	#00%	-4200%
-3.12	#.00;(#.00)	(3.12)
-3.12	#.00;#.00CR	3.12CR

См. также

Раздел <xsl:decimal-format> главы 4

function-available

Эта функция используется для определения доступности других функций. Она применяется для определения доступности как стандартных, так и функций расширения системы.

К примеру, выражение «function-available('concat')» принимает значение истина.

Определена в

XSLT, раздел 15

Формат

function-available(имя) ⇒ логическое значение

Аргументы

Аргумент	Тип данных	Смысл
имя	строка	Имя функции, наличие которой проверяется. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции string(). Строка аргумента должна иметь вид полного имени.

Результат

Значение логического типа: истина, если указанная функция доступна, в противном случае – ложь.

Правила

Аргумент должен иметь вид полного имени, то есть имени XML с необязательным префиксом пространства имен, соответствующим объявлению пространства имен, которое присутствовало в области действия в момент вызова функции function-available().

Если префикс отсутствует либо URI пространства имен пуст, вызов проверяет доступность системной функции с указанным именем. Системные функции – это функции, определенные в стандартах XPath и XSLT (и описанные в этой главе); разработчики систем не должны добавлять функции в стандартное пространство имен, но обязаны реализовать все функции этого пространства, описанные в стандарте. Итак, XSLT-процессор, подчиняющийся требованиям стандарта XSLT версии 1.0, возвращает истину для нижеперечисленных имен (ложь для имен, которых здесь нет):

```
boolean                 generate-id             round
ceiling                 id                      starts-with
```

concat	key	string
contains	lang	string-length
count	last	substring
current	local-name	substring-after
document	name	substring-before
element-available	namespace-uri	sum
false	normalize-space	system-property
floor	not	translate
format-number	number	true
function-available	position	unparsed-entity-uri

Текущий вариант спецификации XSLT 1.1 не добавляет функций к этому списку. Но предположим, что в XSLT 2.0 появилась функция distinct(). Проверить ее доступность в реализации XSLT-процессора можно следующим образом:

```
<xsl:if test="function-available('distinct')">
```

Если полное имя содержит префикс непустого пространства имен, процессор возвращает значение истина, если существует функция расширения с указанным именем. В общем случае, если function-available() возвращает ложь, достаточно безопасно считать, что вызов этой функции приведет к ошибке, а если получено значение истина, существует какой-то способ успешно вызвать функцию. При этом не существует способа в процессе выполнения таблицы стилей узнать, сколько аргументов ожидает получить функция или какой тип данных имеет тот или иной аргумент: система ожидает, что автор таблицы стилей вызовет функцию с корректными параметрами.

Применение

Функцию function-available() можно использовать для достижения обратной совместимости при использовании стандартных функций, определенных в версиях стандарта, более поздних, чем 1.0, а также для проверки наличия расширений, реализованных разработчиками процессора либо сторонними разработчиками.

Проверка существования системных функций

В случае версии 1.0 стандарта возможность проверять доступность определенной функции не особенно полезна. Предполагалось, что она начнет использоваться после выхода новых версий стандарта. Если нужно использовать функцию, которая появилась только в версии 1.1 спецификации XSLT, можно проверить ее доступность в используемом XSLT-процессоре. Если функция недоступна, можно воспользоваться элементом <xsl:if> и избежать ее выполнения. Если режим совместимости с последующими версиями включен путем установки значения атрибута version элемента <xsl:stylesheet> в значение, отличное от "1.0" (или "1.1" в случае использования процессора XSLT 1.1), XSLT-процессор не обращает внимания на присутствие в таблице

стилей вызовов неизвестных функций – разумеется, до тех пор, пока выполнение не затрагивает эти вызовы.

Предположим, что в версии 2.0 спецификации XSLT появилась функция schema-data-type(), которая позволяет определять тип данных узла, определенный в схеме исходного документа. Проверить, реализована ли функция, можно с помощью вызова:

```
<xsl:if test="function-available('schema-data-type')>
```

Более полный пример приведен в конце данного раздела.

> *Не всегда можно заранее определить, будет ли вычислено то или иное выражение, – скажем, если оно используется в качестве предиката, в образце, либо в атрибуте use определения ключа. Однако можно позаботиться о том, чтобы выражение не обрабатывалось, заключив его в оператор* <xsl:if> *с вызовом* function-available().

Обратите внимание: важна версия XSLT, а не версия XPath. XSLT версий 1.0 и 1.1 основаны на XPath 1.0. Разумно предполагать, что при публикации новой версии XPath стандарт XSLT будет соответствующим образом изменен, но это вовсе не значит, что номера версий всегда будут синхронизированы.

В принципе функцию function-available() можно использовать для определения доступности функций версии 1.0, исходя из предположения, что некоторые из процессоров реализуют подмножества XSLT. К сожалению, это работает только в том случае, когда function-available() входит в набор реализованных разработчиком функций.

Проверка существования функций, созданных поставщиком или сторонними разработчиками

Второе применение function-available() – проверка доступности функций расширения, реализованных поставщиком системы или сторонними разработчиками. Если известно, что определенная функция существует в некоторых реализациях, можно использовать function-available() для проверки этого факта, а с помощью оператора <xsl:if> обрабатывать случаи, когда эта функция недоступна.

Примеры

Рабочая версия стандарта XSLT 1.1 разрешает использовать временное дерево (значение, создаваемое не пустым элементом <xsl:variable>) в любом контексте, где может использоваться набор узлов. Такой возможности нет в XSLT 1.0, где временные деревья представляют собой отдельный тип данных, известный как фрагмент конечного дерева. Многие разработчики исправили этот недостаток, разрешив преобразование временного дерева в набор узлов с помощью дополнительных функций (к примеру, xt:node-set() или msxml:node-set()). Чтобы создать таблицу стилей, обладающую максимальной переносимостью, в нее следует включить код, определяющий, какие из механизмов доступны.

Приводимая ниже таблица стилей содержит именованный шаблон, который получает временное дерево в качестве параметра, а затем вызывает <xsl:apply-templates> для обработки корневого узла данного дерева в определенном режиме. При использовании процессора XSLT версии 1.1 дерево просто передается напрямую в <xsl:apply-templates>; в прочих случаях шаблон пытается определить, какие из фирменных функций расширения node-set() доступны, и использует их.

```
<xsl:stylesheet
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          version="1.1">
<xsl:template name="обработать-фрагмент-дерева"
          xmlns:msxml="urn:schemas-microsoft-com:xslt"
          xmlns:xt="http://www.jclark.com/xt"
          xmlns:saxon=" http://icl.com/saxon">
   <xsl:param name="фрагмент"/>
   <xsl:choose>
      <xsl:when test="system-property('xsl:version') &gt; 1.0">
         <xsl:apply-templates mode="обработать-фрагмент"
                        select="$фрагмент"/>
   </xsl:when>

   <xsl:when test="function-available('msxml:node-set')">
       <xsl:apply-templates mode="обработать-фрагмент"
                        select="msxml:node-set($фрагмент)"/>
   </xsl:when>
   <xsl:when test="function-available('xt:node-set')">
          <xsl:apply-templates mode="обработать-фрагмент"
                        select="xt:node-set($фрагмент)"/>
   </xsl:when>
   <xsl:when test="function-available('saxon:node-set')">
          <xsl:apply-templates mode="обработать-фрагмент"
                        select="saxon:node-set($фрагмент)"/>
   </xsl:when>
   <xsl:otherwise>
   <xsl:message terminate="yes">
      Фрагмент конечного дерева не может быть преобразован в набор узлов
   </xsl:message>
   </xsl:otherwise>
   </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

Этот именованный шаблон может быть вызван следующим образом в целях обработки всех узлов фрагмента конечного дерева:

```
<xsl:variable name="набор-узлов">
   <xsl:call-template name="обработать-фрагмент-дерева">
      <xsl:with-param name="фрагмент" select="$переданный-фрагмент"/>
   </xsl:call-template>
</xsl:variable>
```

См. также

element-available() на стр. 522

generate-id

Функция generate-id() создает строку вида XML-имени, которая уникальным образом определяет узел. Гарантируется только то, что результат является уникальным для каждого узла.

К примеру, выражение «generate-id(..)» в одном процессоре может вернуть идентификатор «N015732», а в другом «b23a1c79».

Определена в

XSLT, раздел 12.4

Формат

```
generate-id()  ⇒  строка
generate-id(узел)  ⇒  строка
```

Аргументы

Аргумент	Тип данных	Смысл
узел (необязательный)	набор-узлов	Исходный набор узлов (рассматривается только первый узел набора). При вызове без аргумента рассматривается набор узлов, содержащий только контекстный узел.

Результат

Строковое значение, уникальным образом определяющее узел. Состоит только из цифр и букв ASCII, первый символ должен быть буквой. Это делает идентификатор пригодным к использованию во многих контекстах, в том числе и в качестве имени XML.

Правила

Если аргумент является пустым набором узлов, функция возвращает пустую строку.

Если исходный набор узлов содержит более одного узла, рассматривается только первый из узлов (в порядке следования в документе).

Если аргумент отсутствует, в качестве такового используется контекстный узел.

Функция возвращает произвольную строку. Единственные ограничения – для каждого конкретного узла строка должна быть каждый раз одинаковая, а строки для разных узлов различны. В том числе и в случае, когда узлы принадлежат разным документам.

Создаваемые идентификаторы уникальны в пределах сеанса работы с таблицей стилей. Если одна таблица стилей используется несколько раз с одним и тем же или с разными исходными документами, то в различных сеансах могут (но не обязательно будут) создаваться совпадающие идентификаторы.

Применение и примеры

Мы рассмотрим два полезных способа применения функции `generate-id()`. Создание ссылок в конечном документе и сравнение идентичности узлов.

Использование generate-id() для создания ссылок

Важная роль функции `generate-id()` заключается в создании ссылок в конечном документе. Так, ее можно использовать для создания ID- и IDREF-атрибутов, либо пар вида `<a name="X">` и `<a href="#X">` в конечном HTML-документе.

Пример: Использование generate-id() для создания ссылок

В этом примере обрабатывается файл `города.xml`, содержащий информацию о местах отдыха. Для каждой точки присутствует перечень гостиниц.

Исходный документ

```
<города>
   <город>
      <название>Амстердам</название>
      <описание>Многословное описание Амстердама</описание>
      <гостиница>
         <название>Гранд Отель</название>
         <звезды>5</звезды>
         <адрес> . . . </адрес>
      </гостиница>
      <гостиница>
         <название>Невзрачный Гранд Отель</название>
         <звезды>2</звезды>
         <адрес> . . . </адрес>
      </гостиница>
   </город>
   <город>
      <название>Брюгге</название>
      <описание>Красочное описание Брюгге</описание>
      <гостиница>
         <название>Центральный Отель</название>
         <звезды>5</звезды>
         <адрес> . . . </адрес>
```

```
      </гостиница>
      <гостиница>
         <название>Периферийный Отель</название>
         <звезды>2</звезды>
         <адрес> . . . </адрес>
      </гостиница>
   </город>
</города>
```

Таблица стилей

Таблица стилей `города.xsl` создает HTML-страницу, на которой прежде всего перечисляются гостиницы, а затем выводится информация о местах отдыха. Для каждой гостиницы создается гиперссылка на подробную информацию о городе. Ссылки для мест отдыха создаются путем выполнения `generate-id()` для элементов `<город>`.

Это полная таблица стилей, в которой используется упрощенный синтаксис, описанный в главе 3:

```
<html
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xsl:version="1.0"
>

<body>
   <h1>Гостиницы</h1>
   <xsl:for-each select="//гостиница">
   <xsl:sort select="звезды"
                        order="descending" data-type="number"/>
      <h2><xsl:value-of select="название"/></h2>
      <p>Адрес: <xsl:value-of select="адрес"/></p>
      <p>Звезды: <xsl:value-of select="звезды"/></p>
      <p>Город: <a href="#{generate-id(parent::город)}">
            <xsl:value-of select="parent::город/название"/></a></p>
   </xsl:for-each>

   <h1>Города</h1>
   <xsl:for-each select="//город">
      <h2><a name="{generate-id()}">
         <xsl:value-of select="название"/>
      </a></h2>
      <p><xsl:value-of select="описание"/></p>
   </xsl:for-each>
</body>
</html>
```

Обратите внимание, что `generate-id()` используется дважды – один раз для создания идентификатора места отдыха и один раз для создания ссылки.

Вывод

Приведенный ниже документ получен с помощью процессора Saxon. Я добавил отступы, чтобы отразить структуру. Различные реализации бу-

дут генерировать различные идентификаторы для элементов <a>, но на работу ссылок это никак не повлияет.

```
<html>
   <body>
      <h1>Гостиницы</h1>
        <h2>Гранд Отель</h2>
          <p>Адрес:  . . . </p>
          <p>Звезды: 5</p>
          <p>Город: <a href="#b2b1b2">Амстердам</a></p>
        <h2>Центральный Отель</h2>
          <p>Адрес:  . . . </p>
          <p>Звезды: 5</p>
          <p>Город: <a href="#b2b1b4">Брюгге</a></p>
        <h2>Невзрачный Гранд Отель</h2>
          <p>Адрес:  . . . </p>
          <p>Звезды: 2</p>
          <p>Город: <a href="#b2b1b2">Амстердам</a></p>
        <h2>Периферийный Отель</h2>
          <p>Адрес:  . . . </p>
          <p>Звезды: 2</p>
          <p>Город: <a href="#b2b1b4">Брюгге</a></p>
      <h1>Города</h1>
        <h2><a name="b2ab1">Амстердам</a></h2>
          <p>Многословное описание Амстердама</p>
        <h2><a name="b2b1b4">Брюгге</a></h2>
          <p>Красочное описание Брюгге</p>
   </body>
</html>
```

Функции, обратной для generate-id(), не существует: то есть нет прямого способа определить узел, если известен сгенерированный для него идентификатор, кроме следующего, который весьма неэффективен:

```
//node()[generate-id()=$X]
```

Однако если существует необходимость в получении такой информации, можно определить ключ следующим образом:

```
<xsl:key name="gid-key" match="*" use="generate-id()"/>
```

а затем находить элемент с заданным идентификатором с помощью выражения:

```
key('gid-key', $X)
```

Важно понимать, что генерируемые идентификаторы ничем не напоминают значения ID-атрибутов из исходного документа, поэтому исходные узлы не поддаются поиску с помощью функции id().

Кроме того, значения ID, созданные в одном сеансе, могут отличаться от создаваемых в последующих сеансах. Это следует иметь в виду, если значения

ID используются для создания гиперссылок. Если есть шанс, что преобразование будет выполняться несколько раз, небезопасно ссылаться на эти значения ID из других документов и использовать их в закладках.

Использование generate-id() для определения идентичности узлов

Второе распространенное применение `generate-id()` – определение идентичности двух узлов. А именно, если $X и $Y – это наборы узлов, каждый из которых состоит из единственного узла, выражение

```
generate-id($x) = generate-id($Y)
```

будет истинно только в случае, если $X и $Y ссылаются на один и тот же узел одного и того же документа.

Заметим, что сравнение «$X = $Y» не даст желаемых результатов, поскольку оно будет истинно для любой пары узлов, строковые значения которых совпадают.

Приведем пример случая, когда такая проверка может быть полезна: при группировке соседних узлов. Предположим, исходный XML-документ выглядит так:

```
<план>
<месяц>Апрель</месяц>
<задача>Подписать контракт</задача>
<задача>Нанять менеджера проекта</задача>
<месяц>Май</месяц>
<задача>Разместить заказы на оборудование</задача>
<задача>Нанять рабочих</задача>
<задача>Создать отдел для связи с общественностью</задача>
<месяц>Июнь</месяц>
<задача>Получить сертификат о безопасности</задача>
<задача>Завершить первую фазу строительства</задача>
<задача>Организовать церемонию открытия</task>
</план>
```

и необходимо отобразить результаты в ненумерованном HTML-списке следующим образом:

```
<h2>Задачи на апрель</h2>
    <ol>
      <li>Подписать контракт</li>
      <li>Нанять менеджера проекта</li>
    </ol>
<h2>Задачи на май</h2>
    <ol>
      <li>Разместить заказы на оборудование</li>
      <li>Нанять рабочих</li>
      <li>Создать отдел для связи с общественностью</li>
    </ol>
```

```
<h2>Задачи на июнь</h2>
    <ol>
      <li>Получить сертификат о безопасности</li>
      <li>Завершить первую фазу строительства</li>
      <li>Организовать церемонию открытия</li>
    </ol>
```

Задача упростилась бы, если бы в исходном XML-документе задания для каждого месяца были бы заключены в отдельные элементы. Но не всегда нам удается влиять на вид исходных XML-данных, приходится работать с тем, что дают.

Как и все задачи, связанные с группировкой, эта требует двух вложенных циклов. Первый производит перебор групп, а второй – элементов в группах. В нашем примере определить группы легко:

```
<xsl:for-each select="месяц">
<h2>Задачи на <xsl:value-of select="."/></h2>
<ol>
. . .
</ol>
</xsl:for-each>
```

Но что должен делать вложенный цикл? Необходимо выбрать все соседствующие элементы <задача>, для которых непосредственно предшествующий элемент <месяц> является текущим элементом <месяц>. Это можно сделать с помощью generate-id():

```
<xsl:for-each select="месяц">
<h2>Задачи на <xsl:value-of select="."/></h2>
<ol>
    <xsl:variable name="id-месяца" select="generate-id()"/>
    <xsl:for-each select="following-sibling::задача[
                          generate-id(preceding-sibling::месяц[1])
                          = $id-месяца]">
          <li><xsl:value-of select="."/></li>
    </xsl:for-each>
</ol>
</xsl:for-each>
```

В XPath лишь две операции чувствительны к идентичности узлов – функция generate-id() и оператор объединения «|». Поэтому когда возникает проблема определения идентичности узлов, на которые ссылаются различные переменные или выражения, решение связано с одной из этих операций. В простейшем случае есть выбор: если известно, что $x и $y ссылаются на единичные узлы, можно проверить идентичность этих узлов с помощью такой конструкции:

```
<xsl:if test="generate-id($x) = generate-id($y)">
```

либо эквивалентной:

```
<xsl:if test="count($x | $y ) = 1">
```

Не стоит особенно заострять внимание на выборе того или иного варианта. Тестирование производительности, произведенное Джени Теннисон (Jeni Tennison), показало, что первый вариант быстрее выполняется одними процессорами, второй – другими, а оптимизаторы – постоянно совершенствуются. В конечном итоге я рекомендую использовать `generate-id()`, поскольку этот вариант кажется мне более прозрачным. Но следует помнить, что два варианта эквивалентны только в том случае, если `$x` и `$y` содержат ровно по одному узлу.

См. также

`id()` на стр. 545

`key()` на стр. 548

id

Функция `id()` возвращает набор узлов, содержащий узел или узлы с указанным атрибутом ID.

Например, если атрибут `код` является ID-атрибутом, выражение «`id('A321-780')`» может вернуть набор узлов, содержащий единственный элемент – `<изделие код="A321-780">`.

Определена в

XPath, раздел 4.1

Формат

`id(значение)` ⇒ `набор узлов`

Аргументы

Аргумент	Тип данных	Смысл
значение	любой	Определяет значения ID, которые в каком-то смысле зависят от типа данных. Подробности приводятся в описаниях ниже

Результат

Набор узлов, содержащий узлы с указанными значениями ID.

Правила

Если аргумент не является набором узлов, он преобразуется в строку в соответствии с правилами, определенными для функции `string()`, а получившаяся строка интерпретируется как список лексем, разделенных пробелами.

Каждая лексема потенциально является значением ID: если в документе, которому принадлежит контекстный узел, существует узел с ID-атрибутом, значение которого совпадает с лексемой, этот узел включается в результирующий набор узлов.

Если аргумент является набором узлов, описанная процедура применяется для каждого из узлов набора: узел преобразуется в строку (используется строковое значение), строка интерпретируется в качестве списка лексем, элементы которого разделены пробелами, и каждая из лексем потенциально может являться значением ID. Заметим, это не эквивалентно преобразованию набора узлов в строку с помощью функции `string()`, поскольку используются все узлы набора, а не только первый.

ID-атрибут в данном контексте – любой атрибут, определенный в DTD как имеющий тип ID.

Узлы исходного набора не обязательно должны быть атрибутами типа `IDREF` или `IDREFS`, хотя функция будет возвращать ожидаемый результат и для узлов этих типов. То есть будет происходить поиск узлов, на которые ссылаются значения `IDREF` и `IDREFS` в исходном наборе узлов.

Отсутствие в документе узла с каким-либо из переданных ID-значений не является ошибкой. В такой ситуации в результирующем наборе просто не будет узла, соответствующего конкретному значению. В простейшем случае, если было передано только одно значение ID, возвращаемый набор узлов будет пуст.

Примечания

Значения ID корректно обрабатываются только в действительных документах (грубо говоря, они подчиняются правилам собственного DTD-определения). Однако XSLT и XPath спроектированы таким образом, что недействительные документы могут обрабатываться точно так же, как и действительные. Одна из существующих ошибок, нарушающих действительность документа, заключается в присутствии значений ID, которые не являются уникальными в пределах документа. Такой случай явно рассмотрен в спецификации – происходит поиск первого из узлов с таким значением ID. Прочие ошибки в спецификации не рассматриваются (скажем, что происходит, если значение ID-атрибута содержит пробелы). Разумнее всего считать поведение системы в таких случаях не определенным.

XML-анализаторы, в которых отсутствует механизм подтверждения, не обязаны читать определения атрибутов из внешнего DTD. Соответственно, XSLT-процессор будет считать, что ID-атрибуты отсутствуют, а функция `id()` будет постоянно возвращать пустой результат. В таком случае следует просто использовать другой XML-анализатор. Большинство качественных анализаторов позволяют определять тип атрибута, хотя это не является жестким требованием стандарта XML.

Применение и примеры

Функция id() является эффективным средством поиска узлов по значениям атрибутов ID.

В каком-то смысле эта функция просто облегчает жизнь; ведь если атрибут с именем id всегда имеет тип ID, выражение

```
id('B1234')
```

будет эквивалентно выражению пути:

```
//*[@id='B1234']
```

Но скорее всего в большинстве реализаций функция id() будет более эффективна, нежели прямолинейное выражение пути с предикатом, поскольку процессор, вероятно, произведет индексирование, вместо того чтобы выполнять последовательный поиск.

Возможно также использовать key() вместо id(). Главное преимущество функции id() перед использованием key() в том, что она обрабатывает списки идентфикаторов ID, разделяемых пробелами, в один проход. Функция key() на такое не способна, поскольку значение ключа может содержать пробелы.

Заметим, что id() ищет элементы в том же документе, которому принадлежит контекстный узел. Поиск элементов в другом документе потребует применения <xsl:for-each> с целью изменения контекстного узла:

```
<xsl:for-each select="document('a.xml')">
   <xsl:copy-of select="id('B1234')"/>
</xsl:for-each>
```

В случае если исходный документ содержит атрибут типа IDREFS, возможно найти элементы по всем ссылкам в одну операцию. Предположим, что элемент <книга> имеет атрибут авторы типа IDREFS, который содержит перечень идентификаторов авторов, разделенных пробелами. Соответствующие элементы <автор> могут быть найдены и обработаны следующим образом:

```
<xsl:template match="книга">
   . . .
Авторы: <xsl:for-each select="id(@авторы)">
      <xsl:value-of select="фамилия"/>
      <xsl:if test="position()!=last()">, </xsl:if>
      <xsl:if test="position()=last()-1">и </xsl:if>
</xsl:for-each>
</xsl:template>
```

См. также

key() на стр. 548

key

Функция key() используется для поиска узлов с определенным значением именованного ключа. Функция применяется совместно с элементом <xsl:key>, который описан в главе 4.

Допустим, существует такое определение ключа:

```
<xsl:key name="рег-номер" match="автомобиль" use="@номер"/>
```

Выражение «key('рег-номер', 'N498PAA')» может вычисляться в набор узлов, состоящий из одного элемента, <автомобиль номер="N498PAA">.

Определена в

XSLT, раздел 12.2

Формат

```
key(имя, значение) ⇒ набор узлов
```

Аргументы

Аргумент	Тип данных	Смысл
имя	строка	Имя ключа. Если аргумент не является строкой, он преобразуется в соответствии с правилами, определенными для функции string(). Значение строки должно быть полным именем ключа, определенным с помощью <xsl:key>.
значение	любой	Значение ключа для поиска, зависящее от типа данных (см. ниже).

Результат

Набор узлов, содержащий узлы с указанными значениями ключа.

Правила

Первый аргумент должен иметь вид полного имени: то есть имени XML, предваренного необязательным префиксом пространства имен, который связан с объявлением пространства имен, доступным в области видимости на момент вызова функции key(). Если префикс пространства имен отсутствует, соответствующий URI пространства имен пуст; пространство имен по умолчанию не используется. В таблице стилей должен присутствовать элемент <xsl:key>, определяющий ключ с таким же полным именем, причем при сравнении используются URI, а не префиксы пространств имен. Если элементов <xsl:key> для указанного имени несколько, используются все такие элементы: узел считается выбранным по ключу, если может быть выбран в соответствии с одним из определений ключа с таким именем.

Если второй аргумент не является набором узлов, его значение – при необходимости – преобразуется в строку в соответствии с правилами, определенными для функции string(). В результирующий набор узлов включаются все узлы, принадлежашие тому же документу, что и контекстный узел, значение указанного ключа для которых равно строке второго аргумента. Один узел может иметь несколько значений для одного ключа, а одному значению ключа могут соответствовать несколько узлов.

Если второй аргумент является набором узлов, та же процедура применяется к каждому из узлов набора. Для каждого узла *N* из набора в результирующий набор узлов добавляются те узлы документа, содержащего контекстный узел, значение именованного ключа для которых равно строковому значению *N*. Это не эквивалентно преобразованию исходного набора узлов в строку с помощью функции string(), поскольку она преобразует только первый узел из набора.

Заметим, что в случае нескольких исходных документов все узлы из конечного набора будут принадлежать тому документу, в который входит контекстный узел; узлы, послужившие источником значений ключей, могут принадлежать любому из документов.

Применение и примеры

Функция key() обеспечивает более удобный и эффективный ассоциативный доступ к узлам (поиск узлов по содержанию). Разумеется, степень эффективности целиком зависит от реализации, но весьма вероятно, что в большинстве реализаций используется индексирование или хеш-таблицы, ускоряющие работу функции key() и обеспечивающие ее преимущество перед использованием эквивалентных выражений маршрутов поиска.

Ключи могут также применяться для реализации эффективной группировки узлов.

Ключи: поиск узлов по значениям

Начнем с простого примера. Допустим, необходимо найти элементы <книга>, содержащие элементы <автор> с именем автора Дж. Б. Пристли:

```
<xsl:for-each select="//книга[автор='Дж. Б. Пристли']">
```

Если эта операция часто выполняется в таблице стилей, более эффективно будет определить имя автора в качестве ключа:

```
<xsl:key name="автор-книги" match="книга" use="автор"/>
. . .
<xsl:for-each select="key('автор-книги', 'Дж. Б. Пристли')"/>
```

Функция key() производит поиск элементов в документе контекстного узла. Для поиска элементов в другом документе следует применить <xsl:for-each>:

```
<xsl:for-each select="document('a.xml')">
```

```
    <xsl:copy-of select="key('автор-книги', 'Дж. Б. Пристли')"/>
</xsl:for-each>
```

Значение ключа может быть строкой или набором узлов. В первом случае значение ключа может быть вычислено перед использованием; скажем, оно может быть результатом слияния нескольких значений, полученных из различных источников. Во втором случае значение ключа всегда должно в явном виде присутствовать в исходном документе; преимущество данного варианта в том, что несколько значений для ключа могут быть переданы в одном вызове функции.

Пример: Использование ключей в качестве перекрестных ссылок

В данном примере два исходных файла: основной исходный документ содержит перечень книг, а дополнительный (доступ к которому организован с помощью функции `document()`) содержит биографии авторов. Имя автора из первого файла интерпретируется как перекрестная ссылка на биографию этого автора в дополнительном файле, примерно как в случае объединения в языке SQL.

Исходный документ

Основной документ является урезанной версией файла `книги.xml`:

```
<книги>
<книга категория="ДД">
     <название>Юные гости</название>
     <автор>Дейзи Эшфорд</автор>
</книга>
<книга категория="ДД">
     <название>Когда мы были очень молодыми</название>
     <автор>А. Милн</автор>
</книга>
</книги>
```

Дополнительный исходный документ, `авторы.xml`, выглядит следующим образом. Чтобы сократить размеры файлов, я включил информацию всего для двух авторов, но всю эффективность функции `key()` можно было бы почувствовать, если бы этих записей было несколько сотен.

```
<авторы>

<автор имя="А. Милн">
<родился>1852</родился>
<умер>1956</умер>
<биография>Алан Александр Милн закончил вестминстерскую школу и кембриджский
колледж Тринити. Этот плодовитый автор писал пьесы, романы, стихи, рассказы и
эссе. Но они все потерялись в тени его замечательных детских книг.</биография>
</автор>
<автор имя="Дейзи Эшфорд">
```

```
<родился>1881</родился>
<умер>1972</умер>
<биография>Дейзи Эшфорд (миссис Джордж Норман) написала произведение <i>Юные
гости</i>, небольшой юмористический шедевр, когда была еще совсем ребенком. Вещь
пролежала в ящике стола до 1919, а затем была отправлена издателям Чатто и
Уиндусу, которые в том же году опубликовали ее. Автором предисловия стал Дж. М.
Бэрри, который настоял на личной встрече с автором, желая убедиться, что эта
женщина действительно существует.</биография>
</автор>

</авторы>
```

Таблица стилей

В таблице стилей `биографии-авторов.xsl` объявляется ключ, который отбирает элементы `<автор>` по атрибуту `имя`. Этот ключ будет использован для файла `авторы.xml`, хотя в определении ключа на это нет намеков.

Обратите внимание, для ссылки на дополнительный исходный файл используется глобальная переменная. Можно использовать функцию `document()` при каждом доступе к файлу, и любой достойный упоминания процессор XLST произведет чтение и разбор файла лишь единожды, но, на мой взгляд, использование переменной делает происходящее более очевидным.

Внутренняя инструкция `<xsl:for-each>` не производит ни одной итерации и нужна только для переключения контекста на второй документ, поскольку функция `key()` производит поиск только в документе контекстного узла. Переключение контекста означает, что узлы основного документа перестают быть доступными напрямую, поэтому имя автора предварительно записывается в переменную.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>
<xsl:key name="биог" match="автор" use="@имя"/>
<xsl:variable name="биографии" select="document('авторы.xml')"/>

<xsl:template match="/">
  <html><body>

<xsl:variable name="все-книги" select="//книга"/>
    <xsl:for-each select="$все-книги">
    <!-- для каждой книги из файла со списком книг -->
      <h1><xsl:value-of select="название"/></h1>
      <h2>Автор<xsl:if test="count(автор)!=1">ы</xsl:if></h2>
      <xsl:for-each select="автор">
      <!-- для каждого автора данной книги -->
         <xsl:variable name="имя" select="."/>
         <h3><xsl:value-of select="$имя"/></h3>
         <xsl:for-each select="$биографии">
         <!-- сменить текущий узел на узел из файла с биографиями -->
```

```
          <!-- затем извлечь запись, относящуюся к данному автору -->
            <xsl:variable name="авт" select="key('биог', $имя)"/>
            <p><xsl:value-of
                    select="concat($авт/родился, ' - ',
$авт/умер)"/></p>
            <p><xsl:value-of select="$авт/биог"/></p>
          </xsl:for-each>
      </xsl:for-each>
    </xsl:for-each>
  </body></html>
</xsl:template>

</xsl:transform>
```

Вывод

```
<html>
   <body>
      <h1>Юные гости</h1>
      <h2>Автор</h2>
      <h3>Дейзи Эшфорд</h3>
      <p>1881 - 1972</p>
      <p>Дейзи Эшфорд (миссис Джордж Норман) написала произведение Юные гости,
небольшой юмористический шедевр, когда была еще совсем ребенком. Вещь пролежала
в ящике стола до 1919, а затем была отправлена издателям Чатто и Уиндусу,
которые в том же году опубликовали ее. Автором предисловия стал Дж. М. Бэрри,
который настоял на личной встрече с автором, желая убедиться, что эта женщина
действительно существует.</p>
      <h1>Когда мы были очень молодыми</h1>
      <h2>Автор</h2>
      <h3>А. Милн</h3>
      <p>1852 - 1956</p>
      <p>Алан Александр Милн закончил вестминстерскую школу и кембриджский
колледж Тринити. Этот плодовитый автор писал пьесы, романы, стихи, рассказы и
эссе. Но они все потерялись в тени его замечательных детских книг.</p>
</body>
</html>
```

Использование ключей для группировки

Ключи позволяют эффективным образом находить все узлы с общим значением, поэтому они могут использоваться для группировки узлов при выводе.

Эта техника иногда называется методом группировки Мюнча, поскольку была предложена Стивом Мюнчем (Steve Muench) из корпорации Oracle.

Задача группировки решается парой вложенных циклов. Внешний цикл выбирает узел, который является представителем группы, как правило, это первый, в порядке следования в документе, узел, который входит в группу. Для этого узла выводится информация о группе в целом – как правило, общее значение, используемое для группировки узлов, возможно, промежуточные результаты, вычисленные для членов группы, плюс сопутствующее

форматирование. Вложенный цикл обрабатывает по очереди все элементы группы.

Метод Мюнча заключается в определении ключа для общего значения, которое является критерием группировки. К примеру, если города одной страны составляют группу, можно определить ключ так:

```
<xsl:key name="группа-страны" match="город" use="@страна"/>
```

Внешний цикл выбирает один город для каждой страны. Для этого сначала выбираются все города, а затем из набора удаляются те, которые не занимают первых мест в списках. Определить, что город первым упомянут для страны, можно, сравнив его с первым узлом набора, возвращаемого функцией `key()` для этой страны:

```
<xsl:for-each select="//город[generate-id() =
        generate-id(key('группа-страны', @страна)[1])]">
```

В предикате используется `generate-id()` – для подтверждения идентичности узла города и первого узла набора, возвращаемого функцией `key()`. Если мы уверены, что все города имеют уникальные названия, можно просто сравнивать имена:

```
<xsl:for-each select="//город[@название =
            key('группа-страны', @страна)[1]/@название]">
```

В этот фрагмент кода можно добавить вывод конструкций форматирования, к примеру заголовков. Во внутреннем цикле обрабатываются все города страны, которые отбираются повторным использованием ключа:

```
<xsl:for-each select=key('группа-страны', @страна)">
```

Далее приведен полный пример использования данного метода.

Пример: Группировка с помощью ключей

Исходный документ

Исходный документ `города.xml` содержит перечень городов:

```
<города>
   <город название="Париж" страна="Франция"/>
   <город название="Рим" страна="Италия"/>
   <город название="Ницца" страна="Франция"/>
   <город название="Мадрид" страна="Испания"/>
   <город название="Милан" страна="Италия"/>
   <город название="Фиренце" страна="Италия"/>
   <город название="Наполи" страна="Италия"/>
   <город название="Лион" страна="Франция"/>
   <город название="Барселона" страна="Испания"/>
</города>
```

Таблица стилей

Таблица стилей `группы-городов.xsl` выглядит так:

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>
<xsl:key name="группа-страны" match="город" use="@страна"/>

<xsl:template match="/">
  <html><body>
   <xsl:for-each select="//город[generate-id()
                = generate-id(key('группа-страны', @страна)[1])]">
      <h1><xsl:value-of select="@страна"/></h1>
      <xsl:for-each select="key('группа-страны', @страна)">
         <xsl:value-of select="@название"/><br/>
      </xsl:for-each>
   </xsl:for-each>
  </body></html>
</xsl:template>
</xsl:transform>
```

Вывод

Просмотр результатов в веб-броузере (рис. 7.2):

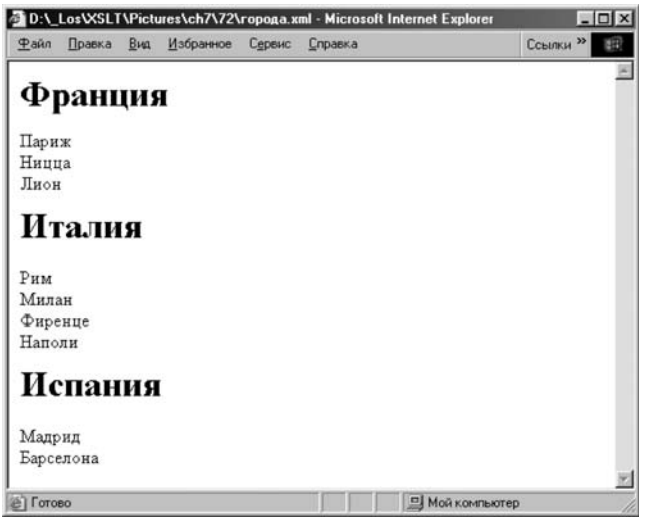


***Рис. 7.2.** Перечисление городов*

См. также

`<xsl:key>` в главе 4

`id()` на стр. 545

lang

Функция lang() позволяет определить, совпадает ли язык контекстного узла, заданный атрибутом xml:lang, с языком, идентификатор которого передается в качестве аргумента.

К примеру, если контекстный узел представлен элементом <para lang="fr-CA"> (канадский диалект французского языка), выражение «lang('fr')» возвращает значение истина.

Определена в

XPath, раздел 4.3

Формат

lang(язык) ⇒ логическое значение

Аргументы

Аргумент	Тип данных	Смысл
язык	строка	Идентификатор языка для сравнения. Если аргумент не является строкой, он преобразуется в соответствии с правилами, определенными для функции string().

Результат

Значение логического типа – истина, если язык контекстного узла совпадает с указанным либо является его подмножеством.

Правила

Язык контекстного узла определяется значением атрибута xml:lang этого узла либо, в случае отсутствия такового, значением атрибута xml:lang ближайшего узла-предка, у которого такой атрибут есть. Если атрибут xml:lang отсутствует у всех таких узлов, функция lang() возвращает ложь.

Атрибут xml:lang – один из немногих атрибутов, значение которых заранее определено спецификацией XML (вообще, можно даже сказать, что это единственный момент в спецификации XML, хоть каким-то образом связанный с непосредственной интерпретацией документа читателем). Значение атрибута может быть четырех видов:

- Двухбуквенный код языка, упомянутый в международном стандарте ISO 639. К примеру, английскому языку соответствует обозначение «en», а французскому – «fr». Обозначения могут записываться в любом регистре, хотя обычно используется нижний. Во втором издании спецификации XML также упоминается введение в обращение трехбуквенных языковых кодов в пересмотренном документе RFC 1766.

- Двухбуквенный код языка, за которым следует один или несколько дополнительных кодов: каждый дополнительный код предваряется дефисом «-». К примеру, американский английский обозначается идентификатором «en-US»; канадский французский – «fr-CA». Первый дополнительный код должен являться двухбуквенным кодом страны из международного стандарта ISO 3166 либо дополнительным кодом языка, зарегистрированным IANA (Internet Assigned Numbers Authority). Коды стран в версии ISO 3166 в основном совпадают с доменами высшего уровня сети Интернет; «DE» для Германии, «CZ» для Республики Чехии. Исключение составляет Великобритания, код страны которой в ISO 3166 (по какой-то причине) «GB», а не «UK». Эти коды преимущественно записываются в верхнем регистре. Значение дополнительных кодов после первого не определено, но они должны состоять только из букв ASCII (a–z, A–Z).
- Код языка, зарегистрированный IANA (см. *www.iana.org*), предваренный строкой «i-». К примеру «i-Navajo».
- Код языка, определенный пользователем, предваряется строкой «x-». К примеру, «x-Java» для элемента, содержащего Java-программу.

Атрибут `xml:lang` определяет язык текста, содержащегося в пределах элемента. Атрибуты `xml:lang` вложенных элементов имеют более высокий приоритет. Таким образом, если документ на английском языке содержит цитаты на немецком, код языка `xml:lang` для документа может выглядеть как «`xml:lang="en"`», а элементы с цитатами будут иметь атрибуты вида «`xml:lang="de"`».

Функция XSLT `lang()` позволяет определить, совпадает ли язык контекстного узла с указанным. «`lang('en')`» возвращает `истину` для английского языка, а «`lang('jp')`» возвращает `истину` для японского.

Выполняются следующие правила:

- Если значение `xml:lang` для контекстного узла совпадает с аргументом функции (регистр символов игнорируется), функция возвращает `истину`.
- Если значение of `xml:lang` для контекстного узла после удаления всех суффиксов, предваряемых дефисом, совпадает с аргументом функции (регистр символов игнорируется), функция возвращает `истину`.
- В остальных случаях функция возвращает `ложь`.

Применение и примеры

Данная функция обеспечивает удобный способ определения языка, используемого в исходном документе. Если исходный документ был соответствующим образом размечен с применением атрибута `xml:lang` и соблюдением требований спецификации XML, функция `lang()` позволяет реализовать специальную обработку данных на конкретном языке.

Приводимый ниже пример демонстрирует еще одно применение `lang()` – для выбора данных из таблицы соответствий в зависимости от языка.

Пример: Применение lang() для локализации дат

Исходный документ

Исходный файл даты-появления.xml содержит даты в формате ISO (ггггммдд). Приводимые даты связаны с появлением различных рабочих вариантов спецификации XSLT 1.0.

```
<даты>
<дата-iso>19991116</дата-iso>
<дата-iso>19991008</дата-iso>
<дата-iso>19990813</дата-iso>
<дата-iso>19990709</дата-iso>
<дата-iso>19990421</дата-iso>
<дата-iso>19981216</дата-iso>
<дата-iso>19980818</дата-iso>
</даты>
```

Таблица стилей

В таблице стилей форматировать-даты.xsl содержатся данные и логика, позволяющие отображать даты с использованием названий месяцев на английском, французском или немецком.

Таблица стилей должна получить глобальный параметр, определяющий язык вывода. Способ передачи параметра зависит от используемой реализации. Командная строка для процессора Saxon выглядит так:

```
saxon даты-появления.xml форматировать-даты.xsl язык=fr
```

Для процессора xt:

```
xt даты-появления.xml форматировать-даты.xsl язык=fr
```

Допустимы следующие значения параметра: «en», «de» и «fr».

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:output encoding="utf-8"/>

<данные xmlns="данные.uri">
<месяцы xml:lang="en">
   <m>January</m><m>February</m><m>March</m><m>April</m>
   <m>May</m><m>June</m><m>July</m><m>August</m>
   <m>September</m><m>October</m><m>November</m><m>December</m>
</месяцы>
<месяцы xml:lang="fr">
   <m>Janvier</m><m>Fйvrier</m><m>Mars</m><m>Avril</m>
   <m>Mai</m><m>Juin</m><m>Juillet</m><m>Aoыt</m>
   <m>Septembre</m><m>Octobre</m><m>Novembre</m><m>Dйcembre</m>
</месяцы>
```

```
<месяцы xml:lang="de">
   <m>Januar</m><m>Februar</m><m>Mдrz</m><m>April</m>
   <m>Mai</m><m>Juni</m><m>Juli</m><m>August</m>
   <m>September</m><m>Oktober</m><m>November</m><m>Dezember</m>
</месяцы>
</данные>

<xsl:param name="язык" select="'en'"/>

<xsl:template match="дата-iso">
<дата xmlns:данные="данные.uri" xsl:exclude-result-prefixes="данные">
   <xsl:value-of select="substring(., 7, 2)"/>
   <xsl:text> </xsl:text>
   <xsl:variable name="месяц" select="number(substring(.,5,2))"/>
   <xsl:value-of select="document('')/*/
            данные:данные/данные:месяцы[lang($язык)]/данные:m[$месяц]"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="substring(., 1, 4)"/>
</дата>
</xsl:template>
</xsl:transform>
```

Задача решается с помощью длинного выражения пути (оно начинается с «document('')»), позволяющего получить имя месяца. Выражение сначала находит документ таблицы стилей («document('')»), затем элемент документа («*» приводит к получению элемента <xsl:stylesheet> или <xsl:transform>), затем элемент верхнего уровня <данные> (в действительности пространством имен по умолчанию для этого элемента является «данные.uri», но в выражении XPath должен использоваться явный префикс пространства имен), затем элемент <месяцы>, соответствующий указанному языку, и наконец элемент <m> для конкретного месяца. Предикат «[lang($язык)]» истинен только для элемента <месяцы>, языковой код которого совпадает с запрошенным языком.

Более подробная информация о доступе к данным, хранящимся в таблице стилей, содержится в описании функции document() на стр. 509.

В XSLT 1.1 эту таблицу стилей можно упростить размещением таблиц соответствий в древовидном значении переменной. Элемент <данные> можно заменить элементом <xsl:variable name="данные"> (таким образом избавившись и от лишнего пространства имен), а выражение, позволяющее получить название месяца, записать следующим образом:

```
<xsl:value-of select="$данные/месяцы[lang($язык)]/m[$месяц]"/>
```

В XSLT 1.0 такой прием не сработает, поскольку невозможно обращаться к фрагментам конечного дерева с помощью оператора пути «/».

Вывод

Вот результат выполнения для английского языка:

```
<?xml version="1.0" encoding="utf-8" ?>
<дата>16 November 1999</дата>
```

```
<дата>08 October 1999</дата>
<дата>13 August 1999</дата>
<дата>09 July 1999</дата>
<дата>21 April 1999</дата>
<дата>16 December 1998</дата>
<дата>18 August 1998</дата>
```

Для французского:

```
<?xml version="1.0" encoding="utf-8" ?>
<дата>16 Novembre 1999</дата>
<дата>08 Octobre 1999</дата>
<дата>13 Aoыt 1999</дата>
<дата>09 Juillet 1999</дата>
<дата>21 Avril 1999</дата>
<дата>16 Dйcembre 1998</дата>
<дата>18 Aoыt 1998</дата>
```

Для немецкого:

```
<?xml version="1.0" encoding="utf-8" ?>
<дата>16 November 1999</дата>
<дата>08 Oktober 1999</дата>
<дата>13 August 1999</дата>
<дата>09 Juli 1999</дата>
<дата>21 April 1999</дата>
<дата>16 Dezember 1998</дата>
<дата>18 August 1998</дата>
```

Функция `lang()` позволяет лишь производить сравнение; чтобы узнать, какой язык используется в действительности, следует прямо извлечь значение атрибута `xml:lang`. Нужный атрибут можно определить с помощью выражения «`ancestor-or-self::*[@xml:lang][1]/@xml:lang`».

last

Функция `last()` возвращает значение размера контекста. При обработке списка узлов узлы нумеруются начиная с единицы. Функция `last()` возвращает число, присвоенное последнему узлу списка.

К примеру, при обработке набора узлов с помощью `<xsl:for-each>` можно выводить запятую перед каждым узлом (кроме последнего) с помощью конструкции:

```
<xsl:if test="position()!=last()">, </xsl:if>
```

Определена в

XPath, раздел 4.1

Формат

```
last() ⇒ число
```

Аргументы

Отсутствуют

Результат

Число, определяющее размер контекста. Как и следует из имени, зависит от контекста.

Правила

Спецификация XPath определяет значение last() в терминах *размера контекста*. В спецификации XSLT используется другая терминология: речь идет о *текущем списке узлов*.

При вычислении выражения верхнего уровня (выражения XPath, которое не является частью другого выражения) размер контекста определяется числом узлов в текущем списке узлов. Существует три возможных варианта:

- Когда вычисляется значение глобальной переменной, либо речь идет об одном из специальных контекстов (вроде вычисления выражения use в элементе <xsl:key>), либо при вычислении начального шаблона, обрабатывающего корневой узел, значение размера контекста устанавливается равным единице.
- При вызове <xsl:apply-templates> для обработки набора узлов этот набор является текущим набором узлов, и размер контекста в этом случае равен числу узлов, выбранных вызовом <xsl:apply-templates>.
- При вызове <xsl:for-each> для обработки набора узлов этот набор является текущим набором узлов, и размер контекста в этом случае равен числу узлов, выбранных вызовом <xsl:for-each>.

При вычислении предиката в выражении или образце размер контекста определяется числом узлов, участвующих в вычислении выражения или образца. Более подробная информация содержится в главе 5 «Выражения» и главе 6 «Образцы».

Применение и примеры

Чтобы понять результаты вызова last(), необходимо знать текущий список узлов.

Когда last() используется в качестве выражения верхнего уровня в элементе <xsl:template> (но не в пределах <xsl:for-each>), функция возвращает число узлов, выбранных соответствующим выражением выбора <xsl:apply-templates>. Дело в том, что <xsl:apply-templates> устанавливает текущий спи-

сок узлов в набор узлов, выбранных выражением, после того как узлы отсортированы в порядке обработки.

Так, следующий код может применяться для нумерации рисунков в документе. Функция last() выводит число рисунков.

```
<xsl:apply-templates select="//рисунок"/>
. . .
<xsl:template match="рисунок">
   <div align="center">
   <img src="{@href}"/>
   <p>Рисунок <xsl:value-of select="position()"/>
      <xsl:text/> из <xsl:value-of select="last()"/></p>
   </div>
</xsl:template>
```

Аналогичным образом, если last() используется в выражении верхнего уровня в пределах <xsl:for-each>, функция возвращает число узлов, выбранных соответствующим выражением выбора <xsl:for-each>. И снова дело в том, что <xsl:for-each> устанавливает текущий список узлов в набор узлов, выбранных выражением select после сортировки.

Функция last() часто применяется в целях определения того, является ли обрабатываемый узел последним в списке. Проиллюстрируем на примере:

Пример: Форматирование списка с помощью функций position() и last()

В данном примере происходит форматирование списка имен в стиле «Адам, Бетси, Чарли и Диана».

Исходный документ

В качестве исходного файла используется книги.xml. Вот фрагмент документа, относящийся к нашему примеру:

```
<книги>
<книга категория="И">
     <название>Паттерны проектирования</название>
     <автор>Эрих Гамма</автор>
     <автор>Ричард Хелм</автор>
     <автор>Ральф Джонсон</автор>
     <автор>Джон Влиссидес</автор>
</книга>
</книги>
```

Таблица стилей

Таблица стилей форматировать-имена.xsl обрабатывает книгу, собирая имена авторов в один элемент:

```
<xsl:transform
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    version="1.0"
>

<xsl:template match="книга">
<авт>
    <xsl:for-each select="автор">
        <xsl:value-of select="."/>
        <xsl:choose>
            <xsl:when test="position() = last()"/> <!-- ничего не делать -->
            <xsl:when test="position() = last()-1"> и </xsl:when>
            <xsl:otherwise>, </xsl:otherwise>
        </xsl:choose>
    </xsl:for-each>
</авт>
</xsl:template>

</xsl:transform>
```

Вывод

Вот результат для файла книги.xml:

```
<авт>Данциг</авт>
<авт>Дейзи Эшфорд</авт>
<авт>А. Милн</авт>
<авт>Эрих Гамма, Ричерд Хелм, Ральф Джонсон и Джон Влиссидес</авт>
```

<xsl:call-template> не изменяет текущий список узлов, поэтому функция last() как в вызывающем, так и в вызываемом шаблоне будет возвращать одно и то же значение.

Если функция last() применяется в выражении выбора элемента <xsl:sort>, речь идет о числе сортируемых узлов. К примеру, указание следующего ключа сортировки:

```
<xsl:sort select="position() mod (ceiling(last() div 3))"/>
```

отсортирует узлы A, B, C, D, E, F, G, H в последовательности A, D, G, B, E, H, C, F, что может быть удобно, если необходимо поместить элементы в таблицу из трех колонок.

Когда last() используется в выражении-предикате, применяемом в выражении, возвращающем набор узлов, размер контекста определяется числом узлов, выбранных на данном шаге выражения (после применения всех предшествующих фильтров). Допустим, имеется исходный документ следующего вида:

```
<страны>
<страна название="Франция" столица="Париж" континент="Европа"/>
<страна название="Германия" столица="Берлин" континент="Европа"/>
<страна название="Испания" столица="Мадрид" континент="Европа"/>
<страна название="Италия" столица="Рим" континент="Европа"/>
<страна название="Польша" столица="Варшава" континент="Европа"/>
<страна название="Египт" столица="Каир" континент="Африка"/>
```

```
<страна название="Ливия" столица="Триполи" континент="Африка"/>
<страна название="Нигерия" столица="Лагос" континент="Африка"/>
</страны>
```

Тогда:

- Выражение «страны/страна[last()]» возвращает элемент страна для Нигерии
- Выражение «страны/страна[@континент='Европа'][last()]» возвращает элемент страна для Польши
- Выражение «страны/страна[@континент='Европа'][last()-1]» возвращает элемент страна для Италии
- Выражение «страны/страна[@континент='Африка'][position() != last()]» возвращает элементы страна Египта и Ливии

Функция last() может использоваться в образце в качестве уточняющего элемента, если последний узел списка следует обрабатывать особым образом. Например:

```
<xsl:template name="обычный-p" match="p">
   <xsl:copy>
      <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>

<xsl:template match="p[last()]">
   <xsl:call-template name="обычный-p"/>
   <hr/>
</xsl:template>
```

Однако такой код может быть низкопроизводительным, поскольку необходимо проверить каждый элемент <p>, чтобы определить, является ли он последним, а это может быть связано с поиском всех дочерних узлов родителя элемента <p>. Зачастую использование <xsl:if> позволяет добиться того же с меньшими потерями:

```
<xsl:template match="p">
   <xsl:copy>
      <xsl:apply-templates/>
   </xsl:copy>
   <xsl:if test="position()=last()">
      <hr/>
   </xsl:if>
</xsl:template>
```

Но следует отметить, что эти два примера не являются строго эквивалентными. Если элементы <p> обрабатываются вызовом <xsl:apply-templates> без указания <xsl:sort>, примеры будут работать одинаково. Но если указан ключ сортировки, второй пример будет выводить элемент <hr/> для последнего из элементов <p> в порядке вывода, а в первом примере элемент <hr/> будет выведен за последним элементом <p> в порядке следования в документе.

Легко ошибиться, решив, что `last()` ***возвращает значение логического типа.*** `last()` ***можно использовать в предикате для определения последнего узла следующим образом:***

```
<xsl:value-of select="para[last()]"/>
```

Но это сокращение предиката «`[position()=last()]`»***, поскольку в предикате численное значение X равносильно проверке условия*** «`position()=X`»***. Этот механизм не действует в других контекстах, и если написать:***

```
<xsl:if test="last()"/>
```

численное значение, возвращаемое функцией `last()`***, будет преобразовано в значение логического типа, как если бы произошел вызов*** `boolean()`***. Результатом всегда будет истина, поскольку*** `last()` ***не может возвращать нуль.***

См. также

`position()` на стр. 582

`<xsl:number>` в главе 4

local-name

Функция `local-name()` возвращает локальную часть имени узла, то есть часть, которой предшествует двоеточие, либо полное имя в случае отсутствия двоеточия.

Так, для контекстного узла-элемента с именем `<заголовок>` выражение «`local-name()`» возвращает значение «`заголовок`»; для элемента `<ms:schema>` – значение «`schema`».

Определена в

XPath, раздел 4.1

Формат

```
local-name()  ⇒  строка
local-name(узел)  ⇒  строка
```

Аргументы

Аргумент	Тип данных	Смысл
узел (необязательный)	набор узлов	Узел, для которого требуется получить локальное имя. Если в наборе узлов несколько, используется первый – в порядке следования в документе. Если набор узлов пуст, функция возвращает пустую строку. Если аргумент отсутствует – функция выполняется для контекстного узла. Если аргумент не является набором узлов, возникает ошибка.

Результат

Строковое значение – локальная часть имени указанного узла.

Правила

Локальное имя зависит от типа узла следующим образом:

Тип узла	Локальное имя
корневой	Отсутствует; возвращается пустая строка
элемент	Имя элемента, часть после двоеточия
атрибут	Имя атрибута, часть после двоеточия
текстовый	Отсутствует; возвращается пустая строка
инструкция обработки	*Цель* инструкции обработки, указывающая приложение, для которого эта инструкция предназначена
комментарий	Отсутствует, возвращается пустая строка
пространство имен	Префикс пространства имен либо пустая строка в случае пространства имен по умолчанию

Применение

Функция может оказаться полезной в случае необходимости выяснить локальное имя, не трогая URI пространства имен. К примеру, можно выбрать элементы <заголовок> и <глава:заголовок> с помощью такой конструкции:

```
<xsl:apply-templates select="*[local-name()='заголовок']"/>
```

Либо можно определить шаблонное правило, выбирающее оба типа узлов:

```
<xsl:template match="*[local-name()='заголовок']">
```

В некотором смысле такой подход может считаться неправильным обращением с пространствами имен XML. Имена из одного пространства предположительно не должны быть связаны с именами другого, а похожесть имен <заголовок> и <глава:заголовок> – простая случайность.

На практике это не всегда верно. Довольно часто одно пространство имен наследует другое. К примеру, почтовая служба США может разработать схему (и связанное с ней пространство имен) для представления названий и адресов в пределах США, канадская почтовая служба может создать вариант такой системы (причем URI пространства имен будет отличаться) для канадских названий и адресов. Эти две схемы могут иметь много общих элементов, и довольно разумно постараться разработать таблицу стилей, которая будет охватывать оба случая. Есть два способа создать шаблонное правило, которое выбирает элементы <сша:адрес> и <канада:адрес>:

Указать оба варианта:

```
<xsl:template match="сша:адрес | канада:адрес">
```

выбрать по локальному имени:

```
<xsl:template match="*[local-name()='адрес']">
```

Примеры

Приводимый ниже фрагмент таблицы стилей выводит HTML-таблицу, в которой перечисляются атрибуты текущего элемента, отсортированные по пространствам имен, а затем по локальным именам:

```
<xsl:template match="*" mode="табулировать">
   <table>
      <xsl:for-each select="attribute::node()">
      <xsl:sort select="namespace-uri()"/>
      <xsl:sort select="local-name()"/>
         <tr>
         <td><xsl:value-of select="namespace-uri()"/></td>
         <td><xsl:value-of select="local-name()"/></td>
         <td><xsl:value-of select="."/></td>
         </tr>
      </xsl:for-each>
   </table>
</xsl:template>
```

См. также

`name()` в следующем разделе

`namespace-uri()` на стр. 572

name

Функция `name()` возвращает полное имя, которое является именем узла. Как правило, это имя узла в том виде, в каком оно присутствует в исходном XML-документе.

К примеру, если контекстным узлом является элемент `<ms:schema>`, реультатом выражения «`name()`» будет строка «`ms:schema`».

Определена в

XPath, раздел 4.1

Формат

`name()` ⇒ строка

`name(узел)` ⇒ строка

Аргументы

Аргумент	Тип данных	Смысл
узел (необязательный)	набор узлов	Узел, для которого требуется получить имя. Если набор содержит более одного узла, используется первый в порядке следования в документе узел. Если набор узлов пуст, функция возвращает пустую строку. Если аргумент отсутствует, в качестве исходного узла используется контекстный. Если аргумент не является набором узлов, возникает ошибка.

Результат

Строковое значение. Полное имя исходного узла.

Правила

Имя узла зависит от его типа следующим образом:

Тип узла	Имя
корневой	Отсутствует; возвращается пустая строка.
элемент	Полное имя элемента, обычно в том виде, в каком оно присутствует в исходном XML-документе, хотя может быть использован иной префикс, отображаемый в URI того же пространства имен.
атрибут	Полное имя атрибута, обычно в том виде, в каком оно присутствует в исходном XML-документе, хотя может быть использован иной префикс, отображаемый в URI того же пространства имен.
текст	Отсутствует; возвращается пустая строка.
инструкция обработки	*Цель* инструкции обработки, указывающая приложение, для которого инструкция предназначена.
комментарий	Отсутствует; возвращается пустая строка.
пространство имен	Префикс пространства имен; пустая строка – для пространства имен по умолчанию. (Результат **не** предваряется строкой `«xmlns:»`.)

За исключением случаев элементов и атрибутов, `name()` возвращает те же результаты, что и `local-name()`.

В полных именах, возвращаемых функцией, обычно используются те же префиксы, что и в исходном XML-документе. Однако гарантий такого поведения системы нет. Гарантируется только использование префикса, который отображается в тот же самый URI пространства имен. Если несколько префиксов, используемых в исходном документе, связаны с одним URI пространства имен, реализация может по своему усмотрению произвести нормализацию этих префиксов.

В XSLT 1.1 функция name() может применяться к узлам временного дерева, созданного с помощью непустого элемента <xsl:variable>. Все, сказанное выше, остается в силе; процессор волен выбирать любой префикс, связанный с правильным URI пространства имен, включая по необходимости префикс, «придуманный» на ходу. Но в таком случае более вероятно, что доступных префиксов будет несколько.

Допустим, в исходном документе присутствует такой фрагмент:

```
<товары xmlns="http://acme.com/catalog">
    <товар название="гайка" запас="2300"/>
    <товар название="болт" запас="2500"/>
    <товар название="шайба" запас="500"/>
</товары>
```

а в таблице стилей определяется следующая древовидная переменная:

```
<xsl:variable name="заканчивающиеся-товары">
  <товары xmlns:cat="http://acme.com/catalog">
      <xsl:copy-of select="//cat:товар[@запас &lt; 1000"/>
  </товары>
</xsl:variable>
```

Каково значение выражения «name($заканчивающиеся-товары/товары/товар)»?

Поскольку элемент дерева <товары> был создан прямым копированием элемента конечного дерева, он будет включать по узлу пространства имен на каждое объявление пространства имен, которое было в области действия на момент создания конечного элемента. В частности, будет присутствовать узел, содержащий отображение префикса «cat» в URI пространства имен «http://acme.com/catalog».

В пределах этого элемента дерева будет присутствовать один элемент <товар> (с атрибутом «название="шайба"»), скопированный из исходного документа. В соответствии с правилами для <xsl:copy-of> этот элемент <товар> будет включать по узлу пространства имен на каждое объявление пространства имен, которое было в области действия элемента <товар> в исходном документе. В частности, будет присутствовать узел, содержащий отображение пустого префикса «» в URI пространства имен «http://acme.com/catalog».

Таким образом в древовидной переменной будет два объявления пространств имен с различными префиксами («» и «cat»), связанных с одним URI. Это значит, что при вызове функции name() для элемента <товар> система может выбирать префикс. В результате будет получено или имя «товар», или «cat:товар»; какое именно – зависит от разработчика системы.

Применение

Функция name() полезна, если необходимо отобразить имя элемента (допустим, в сообщении об ошибке), поскольку вид имени не отличается от того, который обычно применяется пользователем.

Предположим, для корректной работы требуется, чтобы каждый элемент <книга> имел атрибут ISBN:

```
<xsl:if test="not(@ISBN)">
<xsl:message>Для элемента <xsl:value-of select="name()"/>
                не указан атрибут ISBN</xsl:message>
</xsl:if>
```

Функцию name() можно использовать и для сравнения имени узла со строкой, например, так: «doc:заголовок[name(..)='doc:раздел']». Но лучше по возможности избегать этого.

- Во-первых, результат сравнения отрицательный, если в документе для ссылки на пространство имен используется другой префикс. Нет возможности заставить систему считать «doc:раздел» полным именем, поэтому если автор документа решит использовать префикс «DOC» вместо «doc» для ссылки на пространство имен, результат сравнения будет отрицательный, хотя имена эквивалентны.
- Во-вторых, как правило, существует лучший способ решения задачи. Для данного конкретного примера можно использовать форму «doc:заголовок[parent::doc:раздел]». В большинстве случаев, если требуется установить, носит ли узел определенное имя, это можно сделать с помощью подобного предиката. Ось «self» в этом смысле наиболее удобна: чтобы выяснить, является ли текущий узел элементом <рисунок>, выполните <xsl:if test="self::рисунок">.

Существуют случаи, когда подобные конструкции не сработают. Предположим, имена структурированы, и есть необходимость в реализации более сложных проверок для имени. К примеру, в использовании «*[starts-with(name(),'private.')]» для выбора элементов, имена которых начинаются со строки «private.». Здесь мы имеем дело с плохой структурой документа, поскольку имена XML не предназначены для хранения информации посредством внутреннего формата. Для хранения информации об ограничении доступа к элементам гораздо больше подойдет не имя, начинающееся со строки «private.», а атрибут вроде «private="yes"». Но время от времени приходится иметь дело с плохо спроектированными документами, поэтому функция name() предоставляет подходящие для решения подобной задачи возможности. Но следует проявлять осторожность при работе с префиксами пространств имен: если нет уверенности, что префиксы будут использоваться последовательно, лучше остановиться на комбинации локального имени и URI пространства имен, а не на значении, возвращаемом функцией name(). Таким образом, вышеприведенный пример лучше всего записать так:

```
*[namespace-uri()='' and starts-with(local-name(),'private.')]
```

В другом случае функцию name() можно использовать для сравнения имени узла с параметром, который получен таблицей стилей. Распространенный пример – создание таблицы стилей, генерирующей сортированный список записей, и параметризация ключа сортировки. Одному пользователю нужен список книг, отсортированный по авторам, другому – по издательствам,

третьему – по цене. Поскольку выражение select в элементе <xsl:sort> должно быть жестко закодированно в таблице стилей, единственный способ достичь желаемого результата заключается в использовании следующей конструкции:

```
<xsl:sort select="*[name()=$ключ-сортировки]"/>
```

Здесь «$ключ-сортировки» – параметр таблицы стилей, в котором содержится имя поля, по которому следует производить сортировку.

И наконец, еще один случай, где нельзя использовать конструкцию «self»: выбор узлов атрибутов. Дело в том, что основной тип узла оси self составляют элементы. Если необходимо скопировать все атрибуты, кроме описание, соблазнительно использовать такой вариант:

```
<xsl:copy-of select="@*[not(self::описание)]"/>
```

Но работать он не будет, потому что «self::описание» производит поиск элементов <описание> на оси self, а когда контекстным узлом является атрибут, таких элементов нет.

Во всех этих случаях лучше всего не использовать конструкцию «name()!='описание'». Лучше подойдет «local-name()!="'описание' and namespace-uri()=''».

Избегайте применения функции name() с целью создания имен в конечном документе, скажем, с помощью конструкций вроде «<xsl:element name="{name()}">». Каждый префикс в name() интерпретируется в свете объявлений пространств имен, присутствующих в таблице стилей, а не объявлений исходного документа. С целью решения задачи правильнее всего будет использовать <xsl:copy>. Существуют случаи, когда <xsl:copy> не поможет: скажем, если необходимо использовать имя атрибута исходного документа для создания имени элемента в конечном документе. В этом случае лучше воспользоваться функциями local-name() и namespace-uri():

```
<xsl:element name="{local-name()}" namespace="{namespace-uri()}">
```

Примеры

Приводимая ниже таблица стилей создает HTML-таблицу с перечислением элементов, которые присутствуют в исходном документе.

Пример: Перечисление имен элементов исходного документа

Исходный документ

Таблица стилей может использоваться для любого исходного документа. Результат более интересен, если в исходном документе используются пространства имен. Для проверки попытайтесь в качестве исходного документа использовать эту же таблицу стилей.

Таблица стилей

Приводимая ниже таблица стилей вывести-элементы.xsl обрабатывает все узлы элементов исходного документа, сортирует их по URI пространств имен, а затем и по локальным именам, и для каждого элемента отображает имя, префикс, локальное имя и URI пространства имен. Дублирующиеся элементы не удаляются.

Единственный способ получить префикс пространства имен – выполнить name() и извлечь из результата подстроку до двоеточия (если таковое присутствует).

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:template match="/">
<html><body>
<h1>Список элементов</h1>
<table border="1" cellpadding="5">
<tr><td>Элемент</td><td>Префикс</td>
<td>Локальное имя</td><td>URI пространства имен</td></tr>
    <xsl:apply-templates select="//*">
         <xsl:sort select="namespace-uri()"/>
         <xsl:sort select="local-name()"/>
    </xsl:apply-templates>
</table></body></html>
</xsl:template>

<xsl:template match="*">
     <xsl:variable name="префикс">
        <xsl:choose>
        <xsl:when test="contains(name(), ':')">
           <xsl:value-of select="substring-before(name(),':')"/>
        </xsl:when>
        <xsl:otherwise/>
        </xsl:choose>
     </xsl:variable>
<tr>
     <td><xsl:value-of select="name()"/></td>
     <td><xsl:value-of select="$префикс"/></td>
     <td><xsl:value-of select="local-name()"/></td>
     <td><xsl:value-of select="namespace-uri()"/></td>
     </tr>
</xsl:template>

</xsl:transform>
```

Вывод

Вывод на рис. 7.3 получен для таблицы стилей вывести-города.xsl из примера к функции concat() (см. стр. 500), использованной в качестве исходного документа.

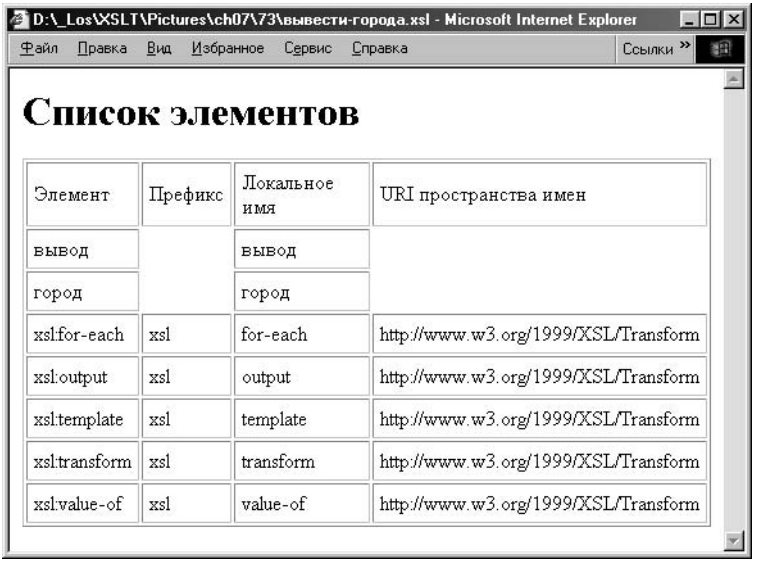


Рис. 7.3. *Перечень элементов*

См. также

`local-name()` на стр. 564

`namespace-uri()` на стр. 572

namespace-uri

Функция `namespace-uri()` возвращает строку, которая является URI пространства имен расширенного имени узла. Как правило, это идентификатор, использованный в объявлении пространства имен, то есть значение атрибута `xmlns` или `xmlns:*`.

Если, например, вызвать эту функцию для элемента высшего уровня таблицы стилей с помощью выражения «`namespace-uri(document('')/*)`», результатом будет строка «`http://www.w3.org/1999/XSL/Transform`».

Определена в

XPath, раздел 4.1

Формат

```
namespace-uri()  ⇒  строка
namespace-uri(узел)  ⇒  строка
```

Аргументы

Аргумент	Тип данных	Смысл
узел (необязательный)	набор узлов	Указывает узел, для которого необходимо определить пространство имен. Если узлов в наборе несколько, используется первый (в порядке следования в документе) узел из набора. Если набор узлов пуст, функция возвращает пустую строку. Если функция вызывается без аргументов, происходит определение пространства имен для контекстного узла. Если аргумент не является набором узлов, это приводит к возникновению ошибки.

Результат

Строковое значение: URI пространства имен для полного имени указанного узла.

Правила

URI пространства имен для конкретного узла зависит от типа этого узла следующим образом:

Тип узла	URI пространства имен
корневой	Отсутствует, в качестве результата возвращается пустая строка.
элемент	Если имя элемента, приводимое в исходном XML-документе, содержит двоеточие, возвращаемым значением будет URI из объявления пространства имен, соответствующего префиксу элемента. В противном случае будет получено значение URI пространства имен по умолчанию. Если таковое отсутствует, результатом будет пустая строка.
атрибут	Если имя атрибута, приводимое в исходном XML-документе, содержит двоеточие, возвращаемым значением будет URI из объявления пространства имен, соответствующего префиксу атрибута. В противном случае результатом будет пустая строка.
текст	Отсутствует, в качестве результата возвращается пустая строка.
инструкция обработки	Отсутствует, в качестве результата возвращается пустая строка.
комментарий	Отсутствует, в качестве результата возвращается пустая строка.
пространство имен	Отсутствует, в качестве результата возвращается пустая строка.

`namespace-uri()` возвращает пустую строку для всех типов узлов, кроме элементов и атрибутов.

Применение

Начнем с рассмотрения некоторых ситуаций, в которых эта функция **не нужна**.

Если необходимо выяснить, принадлежит ли текущий узел определенному пространству имен, это проще всего сделать с помощью критерия имени вида «префикс:*». Так, проверить, принадлежит ли текущий элемент пространству имен «http://ibm.com/ebiz», можно с помощью конструкции:

```
<xsl:if test="self::ebiz:*" xmlns:ebiz="http://ibm.com/ebiz">
```

Лучшим решением для определения URI пространства имен для конкретного префикса является использование узлов пространства имен. Необходимость в этом способе может возникнуть, если префиксы пространств имен используются в значениях атрибутов: такой прием используется и в самом стандарте XSLT – в атрибутах вроде extension-element-prefixes, и, разумеется, он может быть использован и в прочих типах документов XML. Если известно, что значение атрибута «@value» имеет вид полного имени (с указанием пространства имен), получить соответствующий URI пространства имен можно с помощью следующего фрагмента кода:

```
<xsl:variable name="префикс" select="substring-before(@value, ':')"/>
<xsl:variable name="ns-uri"  select="string(namespace::*[name()=$префикс])"/>
```

Функция namespace-uri(), напротив, полезна, когда есть необходимость отобразить URI пространства имен текущего узла либо более подробно изучить данное свойство элемента. Допустим, известно, что существует целое семейство пространств имен, URI-идентификаторы которых начинаются с urn:schemas.biztalk, и необходимо проверить, принадлежит ли текущий узел одному из этих пространств. Этого можно добиться с помощью следующей конструкции:

```
<xsl:if test="starts-with(namespace-uri(), 'urn:schemas.biztalk')>
```

Примеры

В следующем фрагменте таблицы стилей определяется URI пространства имен текущего элемента, а затем предок самого высокого уровня, в котором объявлено это пространство имен:

```
<xsl:template match="*">
   <xsl:variable name="uri" select="namespace-uri()"/>
   URI пространства имен: <xsl:value-of select="$uri"/>
   Объявлен в элементе <xsl:value-of
        select="name(ancestor-or-self::*[namespace::*=$uri][last()])"/>
</xsl:template>
```

См. также

local-name() на стр. 564

name() на стр. 566

normalize-space

Функция normalize-space() удаляет начальные и конечные пробелы из строки и заменяет пробельные последовательности внутри строки единичными пробелами.

К примеру, результатом вычисления «normalize-space(' x	 y ')» является строка «x y».

Определена в

XPath, раздел 4.2

Формат

normalize-space() ⇒ строка

normalize-space(значение) ⇒ строка

Аргументы

Аргумент	Тип данных	Смысл
значение (необязательный)	строка	Исходная строка. Если аргумент не является строкой, он преобразуется в соответствии с правилами, определенными для функции string(). При вызове без аргумента используется строковое значение контекстного узла.

Результат

Строка, полученная удалением начальных и конечных пробелов из исходной, а также заменой внутренних последовательностей пробелов единичными пробелами.

Правила

Пробел в определении спецификации XML – это последовательность из пробелов, табуляций, символов новой строки и символов возврата каретки (#x9, #xA, #xD и #x20).

Применение

Нередко бывает полезно применить функцию normalize-space() ко всем строкам исходного документа перед обработкой содержимого, поскольку многие пользователи считают начальные и конечные пробелы незначимыми, а несколько пробелов или символов табуляции в пределах одной строки – эквивалентными одному пробелу.

Не следует думать, что элемент <xsl:strip-space> сделает все необходимое за вас. Этот элемент удаляет только текстовые узлы, состоящие целиком из пробелов.

Существует ситуация, когда использование normalize-space() не безопасно – при обработке смешанного содержимого элемента, включающего атрибуты форматирования отдельных символов. Допустим, мы обрабатываем узлы такого элемента:

```
<p>Немного <i>весьма</i> традиционного HTML</p>
```

Пробелы после слова «Немного» и «традиционного» значимы, хотя, казалось бы, являются всего лишь начальными и конечными пробелами текстового узла.

Примеры

Следующее объявление ключа индексирует названия книг с нормализованными пробелами:

```
<xsl:key name="название-книги" match="книга" use="normalize-space(название)"/>
```

Ключ может использоваться для поиска книг по названиям следующим образом:

```
<xsl:for-each select="key('название-книги', normalize-space($название))">
```

В результате будет возможно, не зная, сколько пробелов и переводов строки встречается в названии, найти книгу, название которой в исходном документе выглядит как-нибудь так:

```
<книга>
   <название>Объектно-ориентированные языки-
                основные принципы и приемы программирования</название>
</книга>
```

Функция normalize-space() может пригодиться при обработке списка значений, разделенных пробелами. Такие списки применяются в некоторых документах, а также могут создаваться во время выполнения кода таблицы стилей, поскольку в XSLT 1.0 не существует другого способа хранения вычисленных значений в переменной (разумеется, в XSLT 1.1 для этих целей можно использовать временные деревья). После нормирования пробелов в строке можно воспользоваться функцией substring-before() для поочередного получения лексем. Чтобы дополнительно упростить себе жизнь, после

нормирования я обычно добавляю пробел в конец строки. Таким образом, за каждой лексемой следует единичный пробел.

В следующем примере функция `normalize-space()` используется для определения количества слов в строке.

Пример: Использование normalize-space() для подсчета слов

Данная таблица стилей содержит шаблон общего назначения, позволяющий подсчитать количество слов в строке. Работа шаблона демонстрируется подсчетом слов в строковых значениях элементов документа.

Исходный документ

Эта таблица стилей может использоваться с любым исходным XML-документом. К примеру, попробуйте использовать ее для файла `авторы.xml`, который приводится в примере на стр. 550.

Таблица стилей

Таблица стилей `подсчет-слов.xsl` содержит шаблон с именем «счетчик-слов», который производит подсчет слов в параметре «текст». Функция `normalize-space()` используется для замены всех последовательностей пробелов единичными пробелами. Если после обработки строка пуста, возвращается нулевое значение, в противном случае шаблон вызывается рекурсивно (в качестве параметра передается подстрока после первого пробела), и к результату добавляется единица.

Шаблонное правило для корневого узла – просто пример, который показывает, как использовать шаблон «счетчик-слов» для определения количества слов в каждом из элементов документа.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:template name="счетчик-слов">
    <xsl:param name="текст"/>
    <xsl:variable name="нтекст" select="normalize-space($текст)"/>
    <xsl:choose>
    <xsl:when test="$нтекст">
        <xsl:variable name="остаток">
            <xsl:call-template name="счетчик-слов">
                <xsl:with-param name="текст"
                      select="substring-after($нтекст, ' ')"/>
            </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$остаток + 1"/>
    </xsl:when>
    <xsl:otherwise>0</xsl:otherwise>
```

```
        </xsl:choose>
    </xsl:template>

    <xsl:template match="/">
        <xsl:for-each select="//*">
            <xsl:variable name="длина">
                <xsl:call-template name="счетчик-слов">
                    <xsl:with-param name="текст" select="."/>
                </xsl:call-template>
            </xsl:variable>
            <элемент имя="{name()}" слова="{$длина}"/>;
        </xsl:for-each>
    </xsl:template>

    </xsl:transform>
```

Вывод

Для файла `авторы.xml` результат будет следующим:

```
<элементы имя="авторы" слова="93"/>;
<элементы имя="автор" слова="32"/>;
<элементы имя="родился" слова="1"/>;
<элементы имя="умер" слова="1"/>;
<элементы имя="биография" слова="30"/>;
<элементы имя="автор" слова="61"/>;
<элементы имя="родился" слова="1"/>;
<элементы имя="умер" слова="1"/>;
<элементы имя="биография" слова="59"/>;
<элементы имя="i" слова="2"/>;
```

См. также

`concat()` на стр. 498

`substring-after()` на стр. 599

`substring-before()` на стр. 601

not

Функция `not()` возвращает логическое отрицание аргумента: если аргумент имеет значение `истина`, возвращается значение `ложь`, и наоборот.

К примеру, выражение «`not(true())`» всегда возвращает `ложь`.

Определена в

XPath, раздел 4.3

Формат

`not(условие)` ⇒ `логическое значение`

Аргументы

Аргумент	Тип данных	Смысл
условие	логический	Исходное условие. Если аргумент не является значением логического типа, он преобразуется в соответствии с правилами, определенными для функции `boolean()`.

Результат

Логическое отрицание значения аргумента: `истина` для аргумента `ложь` и `ложь` для аргумента `истина`.

Правила

Если аргумент не является значением логического типа, он подвергается преобразованию в соответствии с правилами, определенными для функции `boolean()`.

Если значение (после всех преобразований) истинно, `not()` возвращает `ложь`; если значение ложно, `not()` возвращает `истина`.

Применение

Заметим, что запись «`not($A=2)`» не эквивалентна записи «`$A!=2`». Разница обнаруживается, если `$A` является набором узлов: «`not($A=2)`» будет принимать значение `истина`, если `$A` не содержит узла, который равен 2, тогда как «`$A!=2`» принимает значение `истина`, если в `$A` входит узел, который не равен 2. К примеру, если `$A` – пустой набор узлов, «`not($A=2)`» возвращает `истина`, но «`$A!=2`» возвращает `ложь`.

Об этом легко забыть при рассмотрении значений атрибутов: скажем, следующие два примера работают одинаково, если атрибут `go` присутствует (если значение отличается от «`no`», выводится строка «`go`»), но если атрибут отсутствует – появляются различия и в поведении. Второй пример выводит «`go`», тогда как первый не выводит ничего.

```
1: <xsl:if test="@go!='no'">go</xsl:if>
2: <xsl:if test="not(@go='no')">go</xsl:if>
```

Когда речь заходит о наборах узлов, операторы отношений, такие как «`=`» и «`!=`» подвергаются неявному переводу в форму «если существует»: «`$X=$Y`» означает «если существует пара узлов (x из `$X`, y из `$Y`) такая, что строковые значения узлов идентичны». А чтобы получить форму «для всех» – «у всех узлов набора атрибут `size` установлен в значение 0», можно использовать отрицание условия и всего выражения в целом: «`not(@size!=0)`».

Примеры

Следующая проверка дает положительный результат, если у текущего узла нет потомков:

```
<xsl:if test="not(node())">
```

Следующая проверка дает положительный результат, если у текущего узла нет родительского, то есть если он является корневым узлом:

```
<xsl:if test="not(parent::node())">
```

Следующий оператор <xsl:for-each> обрабатывает все дочерние элементы текущего узла, кроме элементов <заметки>:

```
<xsl:for-each select="*[not(self::заметки)">
```

Следующая проверка дает положительный результат, если строковое значение текущего узла имеет нулевую длину:

```
<xsl:if test="not(.)">
```

Следующая проверка завершается положительно, если атрибут name текущего узла отсутствует либо имеет значение нулевой длины:

```
<xsl:if test="not(string(@name))">
```

Следующая проверка завершается положительно, если атрибут name первого узла из набора $ns отличается от значений атрибута name всех последующих узлов набора (предполагается, что атрибут присутствует во всех узлах набора):

```
<xsl:if test="not($ns[1]/@name = $ns[position()!=1]/@name)">
```

См. также

boolean() на стр. 495
false() на стр. 528
true() на стр. 613

number

Функция number() производит преобразование аргумента в число.
К примеру, выражение «number(' -17.3')» возвращает число –17.3.

Определена в

XPath, раздел 4.4

Формат

```
number()  ⇒  число
number(значение)  ⇒  число
```

Аргументы

Аргумент	Тип данных	Смысл
значение (необязательный)	любой	Значение, которое необходимо преобразовать. В отсутствие аргумента используется строковое значение контекстного узла.

Результат

Значение аргумента, преобразованное в число.

Правила

Правила преобразования связаны с типами данных поступающих значений и перечислены в приведенной ниже таблице.

Исходный тип данных	Правила преобразования
логический	`ложь` преобразуется в нуль; `истина` в единицу
число	Значение не изменяется
строка	Начальные и конечные пробелы удаляются. Если полученная строка состоит из необязательного минуса и числа в формате XPath, она вычисляется как выражение XPath; в противном случае результатом является не-число (NaN)
набор узлов	Набор узлов преобразуется в строку с использованием правил, определенных для функции `string()`, получившаяся строка преобразуется в число таким же образом, как аргумент-строка (см. выше)

Применение

В большинстве ситуаций происходит неявное преобразование в число, поэтому нет необходимости вызывать функцию `number()`.

Но есть один важный случай, когда преобразование должно быть явным, а именно – в предикате. Смысл предиката зависит от типа данных значения, в частности численный предикат интерпретируется относительно позиции в контексте. Если значение имеет тип, отличный от числового, оно преобразуется в логический тип.

Так, к примеру, если значение, хранимое атрибутом или временным деревом, используется в качестве численного предиката, следует явным образом преобразовать его в число.

```
<xsl:apply-templates select="$обьем-продаж[number(@месяц)]"/>
```

Чтобы определить, является ли значение (скажем, атрибута) численным, можно воспользоваться функцией number() и преобразовать его в число, а затем сравнить результат со значением NaN (не-число). Самый простой способ это сделать:

```
<xsl:if test="string(number(@значение))='NaN'"/>
```

Примеры

Выражение	Результат
number(12.3)	12.3
number("12.3")	12.3
number(true())	1.0
number("xyz")	NaN
number("")	NaN

См. также

boolean() на стр. 495
format-number() на стр. 531
string() на стр. 591
<xsl:number> в главе 4

position

Функция position() возвращает значение положения в контексте. При обработке списка узлов position() возвращает номер, присвоенный текущему узлу списка, нумерация узлов при этом начинается с единицы.

Определена в

XPath, раздел 4.1

Формат

```
position()  ⇒  число
```

Аргументы

Отсутствуют

Результат

Число, значение контекстной позиции. Как и следует из названия, позиция зависит от контекста.

Правила

Спецификация XPath определяет значение `position()` в терминах *контекстной позиции* (*context position*). В спецификации XSLT используется другая терминология: речь идет о *текущем узле* (*current node*) и *текущем списке узлов* (*current node list*).

При вычислении выражения верхнего уровня (то есть выражения, которое не является частью другого), контекстная позиция устанавливается в позицию текущего узла в текущем списке узлов. Нумерация узлов начинается с единицы (1). Существует несколько вариантов:

- Если вычисляется значение глобальной переменной либо речь идет об одном из специальных контекстов (вроде вычисления выражения `use` в элементе `<xsl:key>`), текущий список узлов содержит единственный узел (корневой), поэтому контекстная позиция всегда равна 1. Аналогично и в случае, когда для обработки корневого узла применяется первое шаблонное правило.
- При вызове `<xsl:apply-templates>` для обработки набора узлов этот набор является текущим набором узлов, соответственно узлы в наборе имеют тот же порядок, а контекстная позиция последовательно принимает значения из интервала 1, 2, ... *n* по мере обработки узлов. Позиция отражает конечный порядок узлов, другими словами – порядок после сортировки, но совершенно необязательно, что это будет порядок узлов в исходном документе.
- При вызове `<xsl:for-each>` для обработки набора узлов этот набор является текущим набором узлов, узлы в наборе имеют тот же порядок, а контекстная позиция последовательно принимает значения из интервала 1, 2, ... *n* по мере обработки узлов. Позиция отражает конечный порядок узлов, другими словами – порядок после сортировки, но совершенно необязательно, что это будет порядок узлов в исходном документе.
- Если функция `position()` применяется в выражении `select` элемента `<xsl:sort>`, она возвращает позицию узла до сортировки. Так что если, например, необходимо отсортировать узлы в порядке, обратном порядку следования в документе, можно выполнить следующее:

  ```
  <xsl:sort select="position()" data-type="number"
            order="descending">
  ```

  В описании функции `last()` на стр. 559 приводится более сложный пример, в котором узлы сортируются по колонкам с целью отображения в таблице.

При вычислении предиката в выражении или образце контекстная позиция является относительной позицией обрабатываемого узла (на данном шаге вычисления выражения или образца), причем подсчет происходит в порядке следования в документе либо в обратном порядке, в зависимости от направления выбранной оси. Более подробная информация содержится в главе 5 «Выражения» и главе 6 «Образцы».

Применение и примеры

У функции position() есть две области применения – *отображение* текущей позиции и *определение* текущей позиции.

Отображение текущей позиции

В этой ипостаси функция position() является альтернативой инструкции <xsl:number> и может использоваться для реализации простой нумерации параграфов, разделов и рисунков.

Нумерация гораздо более гибко реализуется с помощью элемента <xsl:number>, но у функции position() есть два важных преимущества:

- Она быстрее выполняется
- Нумерация элементов происходит в порядке вывода, тогда как <xsl:number> присваивает номера на основе позиции узла в исходном документе. Это значит, что инструкция <xsl:number> бесполезна или практически бесполезна в случае списка, отсортированного с помощью <xsl:sort>.

В случае использования функции position() возможности форматирования инструкции <xsl:number> по-прежнему доступны. Пример:

```
<xsl:number value="position()" format="(a)"/>
```

Здесь определяется позиция узла и происходит форматирование результата по заданному образцу: полученная последовательность будет состоять из номеров «(a)», «(b)», «(c)» и т. д.

Определение текущей позиции

Позицию текущего элемента можно определить с помощью логического выражения в элементе <xsl:if> или <xsl:when> либо в предикате выражения для набора узлов или в образце.

Обычно бывает необходимо специальным образом обрабатывать первый и последний элементы списка. Допустим, необходимо вставлять горизонтальную линейку после каждого элемента списка, кроме последнего. Можно воспользоваться следующим подходом:

```
<xsl:for-each select="товар">
<xsl:sort select="@название"/>
   <p><xsl:value-of select="@название"/>:
      <xsl:value-of select="описание"/></p>
   <xsl:if test="position() != last()">
      <hr/>
   </xsl:if>
</xsl:for-each>
```

В предикате (в выражении или в образце) численное значение подразумевает неявное сравнение с результатом выполнения функции position(). Например, «item[1]» эквивалентно «item[position()=1]», а «item[last()]» эквивалентно «item[position()=last()]».

Это сокращение можно использовать только в предикате, то есть – в пределах квадратных скобок. Если использовать численное значение в контексте, где предполагается значение логического типа, число будет преобразовано в логическое значение по следующему правилу: 0 преобразуется в ложь, все прочие значения – в истину. Итак, `<xsl:if test="1">` ***не означает*** `<xsl:if test="position()=1">`***; а означает то же, что и*** `<xsl:if test="true()">`**.**

В более сложных случаях бывает необходимо выделить первый узел набора узлов.

Пример: Применение функции position() для рекурсивной обработки набора узлов

Исходный документ

Исходный файл `фигуры.xml` содержит коллекцию геометрических фигур:

```
<фигуры>
<прямоугольник ширина="10" высота="30"/>
<квадрат сторона="15"/>
<прямоугольник ширина="3" высота="80"/>
<круг радиус="10"/>
</фигуры>
```

Таблица стилей

Таблица стилей `площадь.xsl` производит вычисление суммарной площади фигур. Для вычисления площади фигуры каждого типа существует отдельное правило. Шаблон `«суммарная-площадь»` вычисляет площадь первой фигуры с помощью вызова `<xsl:apply-templates>`, а затем добавляет к результату суммарную площадь оставшихся фигур с помощью рекурсивного вызова для всех элементов, кроме первого. Если набор узлов пуст, шаблон возвращает нулевое значение.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:template match="прямоугольник" mode="площадь">
    <xsl:value-of select="@ширина * @высота"/>
</xsl:template>

<xsl:template match="квадрат" mode="площадь">
    <xsl:value-of select="@сторона * @сторона"/>
</xsl:template>

<xsl:template match="круг" mode="площадь">
    <xsl:value-of select="3.14159 * @радиус * @радиус"/>
</xsl:template>

<xsl:template name="суммарная-площадь">
    <xsl:param name="множество-фигур"/>
```

```
    <xsl:choose>
    <xsl:when test="$множество-фигур">
        <xsl:variable name="первая">
            <xsl:apply-templates select="$множество-фигур[1]"
                                 mode="площадь"/>
        </xsl:variable>
        <xsl:variable name="остальные">
            <xsl:call-template name="суммарная-площадь">
                <xsl:with-param name="множество-фигур"
                    select="$множество-фигур[position()!=1]"/>
            </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$первая + $остальные"/>
    </xsl:when>
    <xsl:otherwise>0</xsl:otherwise>
    </xsl:choose>
</xsl:template>

<xsl:template match="фигуры">
    <xsl:call-template name="суммарная-площадь">
        <xsl:with-param name="множество-фигур" select="*"/>
    </xsl:call-template>
</xsl:template>

</xsl:transform>
```

Вывод

При использовании процессора Saxon получается следующий результат:

```
1079.159
```

Чтобы округлить значение до первой цифры после десятичной запятой, можно воспользоваться функцией `format-number()` с форматным образцом «0.0».

См. также

`last()` на стр. 559
`<xsl:number>` в главе 4

round

Функция `round()` возвращает целое число, наиболее приближенное к значению аргумента.

К примеру, выражение «`round(4.6)`» возвращает значение 5.

Определена в

XPath, раздел 4.4

Формат

`round(значение) ⇒ число`

Аргументы

Аргумент	Тип данных	Смысл
значение	число	Исходное значение. Если значение имеет тип, отличный от числового, оно преобразуется в соответствии с правилами, определенными для функции `number()`.

Результат

Целочисленное значение: результат округления первого аргумента до ближайшего целого числа.

Правило

Если значение аргумента не числовое, аргумент сначала преобразуется в число. Более подробно правила преобразования приводятся в описании функции `number()` на стр. 580. Если значение является набором узлов, правила применяются к первому из узлов набора, в порядке обхода документа.

Спецификация XPath скрупулезно определяет результат выполнения `round()`. Правила приводятся в следующей таблице. Понятия положительного и отрицательного нуля, положительной и отрицательной бесконечности объясняются в разделе «Типы данных» главы 2.

Если аргумент...	результатом будет...
Целое число N	N
Число из интервала от N до $N+0.5$	N
Ровно $N+0.5$	$N+1$
Число из интервала от $N+0.5$ до $N+1$	$N+1$
Число из интервала от -0.5 до нуля	Отрицательный нуль
Положительный нуль	Положительный нуль
Отрицательный нуль	Отрицательный нуль
Положительная бесконечность	Положительная бесконечность
Отрицательная бесконечность	Отрицательная бесконечность
NaN (не-число)	NaN

Применение

Функция round() полезна, если необходимо найти ближайшее целое, скажем, при вычислении среднего значения либо при выборе геометрических координат для отображения объекта.

Примеры

Пример: Применение round() для упорядочивания данных в таблице

В следующем примере происходит создание HTML-таблицы, число колонок в которой является параметром. Первая колонка занимает половину ширины страницы, а оставшееся место делится поровну между остальными.

Исходный документ

Исходный файл продажи.xml:

```
<продажи>
   <продукт>Windows 98
      <период название="K1">82</период>
      <период название="K2">64</период>
      <период название="K3">58</период>
   </продукт>
   <продукт>Windows NT
      <период название="K1">17</период>
      <период название="K2">44</период>
      <период название="K3">82</период>
   </продукт>
</продажи>
```

Таблица стилей

Таблица стилей таблица-продаж.xsl приведена ниже. Она вычисляет ширину столбцов на основе числа колонок:

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:template match="продажи">
<html><body>
   <h1>Продажи продуктов</h1>
   <xsl:variable name="столбцы" select="count(продукт[1]/период)"/>
   <table border="1" cellpadding="5" width="100%">
      <tr>
      <th width="50%">Продукт</th>
      <xsl:for-each select="продукт[1]/период">
         <th width="{round(50 div $столбцы)}%">
```

```
            <xsl:value-of select="@название"/>
          </th>
        </xsl:for-each>
      </tr>
      <xsl:for-each select="продукт">
      <tr>
        <td><xsl:value-of select="text()"/></td>
        <xsl:for-each select="период">
          <td>
            <xsl:value-of select="."/>
          </td>
        </xsl:for-each>
      </tr>
      </xsl:for-each>
      </table>
    </body></html>
    </xsl:template>

    </xsl:transform>
```

Вывод

***Рис. 7.4.** Продажи продуктов*

См. также

ceiling() на стр. 497
floor() на стр. 529

starts-with

Функция starts-with() определяет, является ли одна строка началом другой. К примеру, выражение «starts-with('$17.30', '$')» имеет значение истина.

Определена в

XPath, раздел 4.2

Формат

`starts-with(значение, подстрока) ⇒ логическое значение`

Аргументы

Аргумент	Тип данных	Смысл
значение	строка	Строка, в которой производится поиск. Если аргумент не является строковым значением, он преобразуется в соответствии с правилами, определенными для функции `string()`.
подстрока	строка	Подстрока, с которой предположительно начинается строка. Если аргумент не является строковым значением, он преобразуется в соответствии с правилами, определенными для функции `string()`.

Результат

Значение логического типа: `истина`, если `значение` начинается с `подстроки`, в противном случае `ложь`.

Правила

Строки сравниваются посимвольно, начиная с первого. Если вторая строка заканчивается, прежде чем найдена пара различных символов, возвращается значение `истина`; в противном случае возвращается значение `ложь`. Символы считаются одинаковыми, если совпадают их Unicode-значения.

Если вторая строка пуста, результатом всегда будет значение `истина`. Если пуста первая строка, результатом будет значение `истина` только в том случае, когда вторая строка также пуста. Если вторая строка длиннее первой, результатом всегда будет значение `ложь`.

Применение и примеры

Функция `starts-with()` полезна для работы с текстовыми значениями либо значениями атрибутов, которые обладают определенной внутренней структурой. Так, следующее шаблонное правило выбирает все элементы `<link>`, у которых есть атрибут `href` со значением, начинающимся с символа «#»:

```
<xsl:template match="link[starts-with(@href, '#')]">
. . .
</xsl:template>
```

Примечание

Не существует функции `ends-with()`. Вот самый простой способ проверить, заканчивается ли строка `$A` строкой `$B`:

```
substring($A, string-length($A) - string-length($B) + 1) = $B
```

См. также

contains() на стр. 501
string-length() на стр. 593

string

Функция string() преобразует аргумент в строковое значение.

К примеру, выражение «string(4.00)» возвращает строку «4».

Определена в

XPath, раздел 4.2

Формат

```
string()  ⇒  строка
string(значение)  ⇒  строка
```

Аргументы

Аргумент	Тип данных	Смысл
значение (необязательный)	любой	Исходное значение. Если аргумент отсутствует, используется набор узлов, в который входит только контекстный узел.

Результат

Строковое значение: результат преобразования аргумента в строку.

Правила

Значение любого типа может быть преобразовано в строку. Для преобразования применяются следующие правила.

Тип данных	Правила преобразования
логический	Логическое значение ложь преобразуется в строку «ложь». Значение истина преобразуется в строку «истина».
число	Не-число преобразуется в строку «NaN». Положительный и отрицательный нули преобразуются в строку «0». Положительная бесконечностью преобразуется в строку «Infinity». Отрицательная бесконечность преобразуется в строку «-Infinity». Целочисленное значение представляется в виде десятичного числа без десятичной запятой. Начальные нули удаляются; если число отрица-

Тип данных	Правила преобразования
	тельное, в результат включается минус. Результат является корректным числом XPath. Любое другое число представляется в десятичном формате, в котором присутствует по меньшей мере одна цифра перед запятой и одна цифра после запятой. Отрицательные числа предваряются минусом, начальные нули удаляются, если они не предшествуют непосредственно десятичной запятой. Число цифр после запятой должно быть достаточным, чтобы данное числовое значение можно было отличить от других числовых значений IEEE 754. Результат является корректным числом XPath.
строка	Значение не изменяется.
набор узлов	Если набор узлов пуст, результатом является пустая строка. В противном случае результатом является строковое значение первого, в порядке следования в документе, узла. Строковым значением текстового узла является его текстовое содержание. Строковым значением комментария является комментарий. Строковым значением инструкции обработки является часть данных. Строковым значением узла пространства имен является URI этого пространства. Строковым значением узла атрибута является значение атрибута. Строковым значением корневого узла или узла элемента является объединение содержания всех текстовых узлов-потомков, взятых в порядке обхода документа.

Переменная, значением которой является дерево, вроде `<xsl:variable name="v">Нью-Йорк</xsl:variable>`, технически является набором узлов, состоящим из корневого узла, у которого один текстовый дочерний узел. Строковым значением этой переменной является строковое значение корневого узла, которое, в свою очередь, определено как объединение значений всех текстовых узлов-потомков. Для рассматриваемого случая получаем строковое значение «Нью-Йорк».

В данном случае, несмотря на то, что переменная является набором узлом, она может использоваться во всех контекстах и целях в качестве строки, содержащей значение «Нью-Йорк». Это верно для версий XSLT 1.0 и 1.1; спецификация XSLT 1.0 явным образом оговаривает, что фрагмент конечного дерева может использоваться везде, где может использоваться строка.

Применение и примеры

Обычно нет необходимости явным образом вызывать функцию `string()`, поскольку она во многих случаях вызывается автоматически – если контекст требует строкового значения, а действительное значение таковым не является.

Вот пример ситуации, когда явное преобразование является уместным. Необходимо принудительно сравнить строковые значения, а не наборы узлов. Так, следующая проверка завершается положительно, если существует непосредственный потомок текущего узла <автор> со значением «Дж. Б. Пристли».

```
<xsl:if test="автор='Дж. Б. Пристли'">
```

А эта проверка завершается положительно только в том случае, если первый непосредственный потомок <автор> (в порядке следования в документе) имеет такое значение:

```
<xsl:if test="string(автор)='Дж. Б. Пристли'">
```

Однако в данном случае более прозрачна такая конструкция:

```
<xsl:if test="автор[1]='Дж. Б. Пристли'">
```

См. также

boolean() на стр. 495
number() на стр. 580

string-length

Функция string-length() возвращает число символов в строке.

Например, выражение «string-length('Бетховен')» возвратит значение 8.

Определена в

XPath, раздел 4.2

Формат

```
string-length()  ⇒  число
string-length(значение)  ⇒  число
```

Аргументы

Аргумент	Тип данных	Смысл
значение (необязательный)	строка	Строка, длину которой необходимо определить. Если значение имеет тип, отличный от строкового, оно преобразуется в соответствии с правилами, определенными для функции string(). При вызове без аргумента используется строковое значение контекстного узла.

Результат

Число символов в значении аргумента.

Правила

Символами считаются экземпляры порождающего правила Символ (Char) XML. Это значит, что суррогатная пара Unicode (пара 16-битных значений, используемых для представления символа Unicode из диапазона от #x10000 до #x10FFFF) считается за один символ.

Здесь важно число символов в строке, а не способ записи символов в исходном документе. Символ, записанный с помощью ссылки на символ вроде «ÿ» или ссылки на сущность вроде «&», считается за один символ.

Комбинирующие и не пробельные символы Unicode считаются отдельными символами, за исключением случаев, когда они нормализуются реализацией системы. Реализациям позволено приводить строки к канонической форме, но они не обязаны это делать. В канонической форме акценты и диакритические символы, как правило, объединяются с буквами, которые модифицируют. В таких случаях результат выполнения string-length() однозначно не может быть определен.

Применение

Функция string-length() может успешно применяться для определения объема создаваемых данных. К примеру, если список выводится в несколько колонок, число колонок может быть определено с помощью алгоритма, основанного на максимальной длине отображаемой строки.

В случае когда требуется определить, пуста ли строка, **нет** необходимости вызывать string-length(), поскольку преобразование строки в значение логического типа – явное, посредством функции boolean(), или неявное, посредством контекста, – приводит к получению значения истина только в случае, когда длина строки не меньше одного символа. По той же причине обычно нет необходимости вызывать string-length() в процессе обработки символов в строке с помощью рекурсивных итераций, поскольку условие окончания обработки – получение пустой строки – может быть проверено преобразованием строки в значение логического типа.

Примеры

В следующей таблице приведены примеры выполнения string-length() для различных строк:

Выражение	Результат
string-length('абв')	3
string-length('')	0
string-length('<>')	2
string-length('&#xFFFD;')	1
string-length('𠀀')	1

См. также

`substring()` в следующем разделе

substring

Функция `substring()` возвращает часть строкового значения, определенную позициями символов в строке. Нумерация позиций начинается с единицы. К примеру, выражение «`substring('Гольдфарб', 6, 3)`» возвращает строку «фар».

Определена в

XPath, раздел 4.2

Формат

```
substring(значение, начало)  ⇒  строка
substring(значение, начало, длина)  ⇒  строка
```

Аргументы

Аргумент	Тип данных	Смысл
значение	строка	Исходная строка. Если значение имеет тип, отличный от строкового, оно преобразуется в соответствии с правилами, определенными для функции `string()`.
начало	число	Позиция первого символа исходной строки, который следует включать в результат. Если значение имеет тип, отличный от числового, оно преобразуется в соответствии с правилами, определенными для функции `number()`.
длина (необязательный)	число	Число символов, которые следует включить в результат. Если значение имеет тип, отличный от числового, оно преобразуется в соответствии с правилами, определенными для функции `number()`. Если этот аргумент отсутствует, в результате возвращаются все символы от указанного в аргументе `начало` до последнего символа исходной строки.

Результат

Строка: запрошенная подстрока исходной строки.

Правила

Если говорить неформально, функция возвращает строку, которая состоит из символов строки *значение*, начиная с символа *начало*; если указана *длина*, возвращаемая строка содержит указанное число символов, в противном случае возвращаются все символы до конца строки *значение*.

Символы строки нумеруются числами 1, 2, 3 ... *n*. Это знакомо программистам на Visual Basic, но не тем, кто привык к языкам C и Java, поскольку в этих языках нумерация начинается с нуля.

Символами считаются экземпляры порождающего правила `Символ` (`Char`) XML. Это значит, что суррогатная пара Unicode (пара 16-битных значений, используемых для представления символа Unicode из диапазона от `#x10000` до `#x10FFFF`) считается за один символ.

Комбинирующие и непробельные символы Unicode считаются отдельными символами, за исключением случаев, когда они нормализуются реализацией системы. Реализациям позволено приводить строки к канонической форме, но они не обязаны это делать. В канонической форме акценты и диакритические символы, как правило, объединяются с буквами, которые модифицируют.

Формальные правила неожиданно сложны, поскольку они учитывают такие вещи, как отрицательные длины и позиции, значения вроде NaN, дробные, а также бесконечные (описание числового типа данных находится в разделе «Типы данных» главы 2). Правила таковы:

> ***Пусть*** начало ***– численное значение второго аргумента. Пусть*** длина ***– численное значение третьего аргумента (может отсутствовать).***
>
> ***Если присутствует аргумент*** длина***, возвращаемая строка содержит те символы, для которых позиция*** p ***в исходной строке удовлетворяет следующим условиям:***
>
> ```
> p >= round(начало)
> ```
>
> ***и***
>
> ```
> p < round(начало) + round(длина).
> ```
>
> ***Если*** длина ***не указана, возвращаемая строка содержит те символы, для которых позиция*** p ***в исходной строке удовлетворяет условию:***
>
> ```
> p >= round(начало)
> ```

Округление выполняется с помощью функции `round()`, которая описана на стр. 586. Сравнения и арифметические вычисления выполняются с помощью арифметических механизмов IEEE 754, что может иметь любопытные последствия, если используются значения вроде бесконечности и не-чисел либо любые дробные значения. Правила арифметики IEEE 754 описаны в главе 2.

Формальное правило гласит, что если аргумент *начало* меньше единицы, строка результата начинается с первого символа исходной строки, а если этот аргумент больше длины строки, результат будет пустой строкой. Если аргумент *длина* меньше нуля, он считается нулем, а в качестве результата возвращается пустая строка. Если аргумент *длина* превышает число доступных символов, возвращаются все символы до конца исходной строки.

Применение и примеры

Функция substring() полезна для посимвольной обработки строки. Она часто используется для извлечения первого символа строки:

```
<xsl:variable name="имя-диска">
   <xsl:if test="substring($имя-файла,2,1)=':'">
      <xsl:value-of select="substring($имя-файла,1,1)"/>
   </xsl:if>
</xsl:variable>
```

А также при работе с принятым в США форматом имени (имя, инициал, фамилия):

```
<xsl:variable name="отобразить-имя">
   <xsl:value-of select="имя"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="substring(второе-имя, 1, 1)"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="фамилия"/>
</xsl:variable>
```

Чтобы определить, кончается ли строка определенным суффиксом, можно использовать substring() в паре с функцией string-length(). Вспомним, что нумерация символов начинается с единицы:

```
<xsl:variable name="связанный-документ">
   <xsl:if test="substring(@href, string-length(@href)-3) = '.xml'">
      <xsl:value-of select="document(@href)"/>
   </xsl:if>
</xsl:variable>
```

В следующем примере мы определяем, содержит ли исходная строка последовательность вида «#x#», где «x» – произвольный символ:

```
<xsl:if test="contains($s, '#') and
                      substring(substring-after($s, '#'), 2, 1)='#'">
```

Функция substring() также может применяться в качестве своего рода условного выражения. Предположим, $b – значение логического типа. Рассмотрим выражение:

```
substring("xyz", 1, $b * string-length("xyz"))
```

Поскольку $b используется в численном выражении, происходит преобразование значения в число: 0 для случая ложь, 1 для случая истина. Итак, значение третьего аргумента 0, если $b ложно, и 3, если $b истинно. Окончательный результат, возвращаемый функцией substring(), таким образом, будет пустой строкой для случая, когда $b ложно, либо строкой "xyz", если $b истинно. Это выражение эквивалентно конструкции «($b ? "xyz" : "")» в языках Java и C.

По сути дела, чтобы это работало, третий аргумент вовсе не обязательно должен быть равен длине строки. Достаточно любого значения, которое больше, чем длина строки. Можно с тем же успехом написать:

```
substring("xyz", 1, $b * 1000)
```

или даже:

```
substring("xyz", 1, $b * (1 div 0))
```

пользуясь тем фактом, что деление «1 div 0» дает бесконечность.

Такой трюк может пригодиться в ситуации определения таблицы HTML. Довольно часто требуется создать ячейку таблицы, содержащую конкретное значение (скажем «item»), если значение существует, либо неразрывный пробел ( ) в противном случае. Можно решить задачу с помощью <xsl:choose>, но вот более лаконичный способ:

```
concat(item, substring(' ', 1, not(item))
```

Эта концепция также может оказаться полезной в контекстах вроде определения ключа (<xsl:key>) или ключей сортировки (<xsl:sort>), когда все вычисления ограничиваются XPath и нет возможности использовать XSLT для создания переменных. Предположим, необходимо отсортировать набор цен, причем некоторые цены в долларах, а некоторых в фунтах:

```
<цены>
  <товар валюта="GBP" цена="14.00"/>
  <товар валюта="GBP" цена="18.00"/>
  <товар валюта="USD" цена="23.50"/>
  <товар валюта="GBP" цена="12.00"/>
  <товар валюта="USD" цена="17.80"/>
</цены>
```

Задачу решает следующий код:

```
<xsl:variable name="курс" select="1.53"/>
  <xsl:for-each select="товар">
    <xsl:sort data-type="number"
         select="concat(
                 substring(@цена * $курс, 1, (@валюта='USD') *
                           1000),
                 substring(@цена, 1, (@валюта='GBP') * 1000))"/>
  </xsl:for-each>
</xsl:variable>
```

Я советую в случае использования подобных конструкций создавать подробные комментарии, касающиеся механизма работы, и не очень ругать эту книгу за то, что здесь они отсутствуют.

См. также

`substring-after()` в следующем разделе
`substring-before()` на стр. 601
`string-length()` на стр. 593
`contains()` на стр. 501

substring-after

Функция `substring-after()` возвращает ту часть строкового значения, которая следует за первым вхождением указанной подстроки.

К примеру, выражение «`substring-after('print=yes', '=')`» возвращает строку «`yes`».

Определена в

XPath, раздел 4.2

Формат

`substring-after(значение, подстрока)` ⇒ `строка`

Аргументы

Аргумент	Тип данных	Смысл
значение	строка	Исходная строка. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции `string()`.
подстрока	строка	Подстрока-разграничитель. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции `string()`.

Результат

Строка, содержащая символы, следующие за первым вхождением указанной подстроки в исходную строку.

Правила

Если исходная подстрока не содержит подстроку-разграничитель, функция возвращает пустую строку. Заметим, что получение пустой строки в качестве результата может указывать и на то, что исходная строка заканчивается подстрокой-разграничителем. Эти два случая можно отличить с помощью функции `contains()`.

Если исходная строка содержит подстроку-разграничитель, функция возвращает строку, состоящую из всех символов исходной строки, следующих за первым вхождением подстроки-разграничителя.

Если подстрока-разграничитель пуста, функция возвращает исходную строку.

Если исходная строка пуста, функция возвращает пустую строку.

Применение и примеры

Функция `substring-after()` полезна, когда необходимо обработать строку, содержащую символы-разделители. К примеру, если строка является списком лексем, элементы которого разделены пробелами, первую лексему можно получить с помощью вызова

```
substring-before($s, ' ')
```

а оставшуюся часть строки так:

```
substring-after($s, ' ')
```

Нелишне использовать функцию `normalize-space()`, чтобы каждый разделитель состоял из единственного пробела, а также функцию `concat()` для добавления одного дополнительного пробела к концу строки, чтобы функции корректно работали даже в случае, когда список содержит всего одну лексему.

В следующем примере лексемы из списка, элементы которого разделены пробелами, перечисляются с новым разделителем, пустым элементом `<br/>`.

```
<xsl:template name="вывести-лексемы">
   <xsl:param name="список"/>
   <xsl:variable name="нсписок"
      select="concat(normalize-space($список),' ')"/>
   <xsl:variable name="первый" select="substring-before($нсписок, ' ')"/>
   <xsl:variable name="остальные" select="substring-after($нсписок, ' ')"/>
   <xsl:value-of select="$первый"/>
   <xsl:if test="$остальные">
      <br/>
      <xsl:call-template name="вывести-лексемы">
         <xsl:with-param name="список" select="$остальные"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

См. также

`contains()` на стр. 501

`substring()` на стр. 595

`substring-before()` в следующем разделе

substring-before

Функция substring-before() возвращает часть строкового значения, которая предшествует первому вхождению указанной подстроки.

Например, выражение «substring-before('print=yes', '=')» возвращает строку «print».

Определена в

XPath, раздел 4.2

Формат

substring-before(значение, подстрока) ⇒ строка

Аргументы

Аргумент	Тип данных	Смысл
значение	строка	Исходная строка. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции string().
подстрока	строка	Подстрока-разграничитель. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции string().

Результат

Строка, содержащая символы, которые предшествуют первому вхождению подстроки-разграничителя в исходную строку.

Правила

Если исходная строка не содержит подстроку-разграничитель, функция возвращает пустую строку. Заметим, что это может указывать и на то, что исходная строка начинается подстрокой-разграничителем. Эти два случая можно отличить с помощью функции starts-with().

Если исходная строка содержит подстроку-разграничитель, функция возвращает строку, состоящую из всех символов исходной строки, предшествующих первому вхождению разграничителя.

Если исходная строка или подстрока-разграничитель пуста, функция возвращает пустую строку.

Применение и примеры

Пример использования функций substring-after() и substring-before() для обработки списка лексем, разделенных пробелами, приводится в описании функции substring-after() на стр. 599.

Если поводом для использования substring-before() является необходимость определить присутствие определенного префикса в строке, следует воспользоваться функцией starts-with(). Можно написать:

```
<xsl:if test="substring-before($url, ':')='https'">
```

но такой вариант проще:

```
<xsl:if test="starts-with($url, 'https:')">
```

Функции substring-before() и substring-after() могут использоваться для замены фрагмента строки. Функция translate() не может применяться для замены одного слова другим. Чтобы решить задачу, необходимо использовать сочетание функций contains(), substring-before(), substring-after() и, возможно, concat(). Проиллюстрируем следующим примером.

Пример: Замена всех вхождений строки

Приведенная ниже таблица стилей заменяет все вхождения указанной строки другой строкой в пределах любого текстового узла исходного документа. Заменяемая строка передается через глобальный параметр «заменить», строка подстановки – через параметр «на».

Исходный документ

Можно использовать любой исходный документ. К примеру, файл авторы.xml с параметрами заменить=автор и на=писатель.

Таблица стилей

Таблица стилей замена.xsl копирует все элементы и атрибуты без изменений, но обрабатывает текстовые узлы с помощью именованного шаблона «выполнить-замену», который заменяет все вхождения строки «заменить» строкой «на». Первая замена производится напрямую, затем функция вызывается рекурсивно для обработки оставшегося текста.

Чтобы таблица стилей заработала, она должна получить глобальные параметры. Способ их передачи зависит от используемой реализации. В случае процессора Saxon задача решается с помощью командной строки:[1]

```
saxon исходный-файл.xml замена.xsl заменить=xxx на=yyy
```

В случае процессора Xalan следует выполнить (одной строкой):

```
java org.apache.xalan.xslt.Process -in исходный-файл.xml -xsl
        замена.xsl -param заменить xxx -param на yyy
```

[1] Необходимо иметь в виду, что если записать русское название параметра непосредственно в командной строке, то оно будет в текущей кодировке системы, которая может отличаться от кодировки таблицы стилей, что приведет к тому, что параметр не будет передан в таблицу стилей. Поэтому либо следует воспользоваться другим способом передачи параметра таблице стилей, либо использовать в именах параметров только символы ASCII. – *Примеч. науч. ред.*

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:param name="заменить"/>
<xsl:param name="на"/>

<xsl:template name="выполнить-замену">
   <xsl:param name="текст"/>
   <xsl:choose>
   <xsl:when test="contains($текст, $заменить)">
      <xsl:value-of select="substring-before($текст, $заменить)"/>
      <xsl:value-of select="$на"/>
      <xsl:call-template name="выполнить-замену">
         <xsl:with-param name="текст"
                         select="substring-after($текст, $заменить)"/>
      </xsl:call-template>
   </xsl:when>
   <xsl:otherwise>
      <xsl:value-of select="$текст"/>
   </xsl:otherwise>
   </xsl:choose>

</xsl:template>

<xsl:template match="*">
    <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
    </xsl:copy>
</xsl:template>

<xsl:template match="text()">
    <xsl:call-template name="выполнить-замену">
        <xsl:with-param name="текст" select="."/>
    </xsl:call-template>
</xsl:template>

</xsl:transform>
```

Вывод

Результат обработки файла `авторы.xml` **с параметрами** «заменить=автор на=***» **приводится ниже:**

```
<авторы>

<автор имя="А. Милн">
<родился>1852</родился>
<умер>1956</умер>
<биография> Алан Александр Милн закончил вестминстерскую школу и кембриджский
колледж Тринити. Этот плодовитый *** писал пьесы, романы, стихи, рассказы и
эссе. Но они все потерялись в тени его замечательных детских книг.</биография>
```

```
</автор>

<автор имя="Дейзи Эшфорд">
<родился>1881</родился>
<умер>1972</умер>
<биография> Дейзи Эшфорд (миссис Джордж Норман) написала произведение <i>Юные
гости</i>, небольшой юмористический шедевр, когда была еще совсем ребенком. Вещь
пролежала в ящике стола до 1919, а затем была отправлена издателям Чатто и
Уиндусу, которые в том же году опубликовали ее. ***ом предисловия стал Дж. М.
Бэрри, который настоял на личной встрече с ***ом, желая убедиться, что эта
женщина действительно существует.</биография>
</автор>
</авторы>
```

См. также

contains() на стр. 501
starts-with() на стр. 589
substring() на стр. 595
substring-after() на стр. 599

sum

Функция sum() вычисляет сумму численных значений, находящихся в наборе узлов.

Если, например, контекстным узлом является элемент <rect x="20" y="30"/>, выражение «sum(@*)» возвращает 50. (Выражение «@*» — это набор узлов, содержащий все атрибуты контекстного узла.)

Определена в

XPath, раздел 4.4

Формат

sum(узлы) ⇒ число

Аргумент

Аргумент	Тип данных	Смысл
узлы	набор узлов	Набор узлов, для которого следует производить сложение. Если аргумент не является набором узлов, возникает ошибка.

Результат

Число. Результат сложения строковых значений узлов набора, преобразованных в численные.

Правила

Преобразование строковых значений выполняется в соответствии с правилами, определенными для функции number().

Сложение численных значений выполняется по арифметическим правилам, определенным в IEEE 754.

Как следствие, если строковое значение одного из узлов не может быть преобразовано в число, результатом выполнения функции sum() будет не-число.

Для пустого набора узлов возвращается нулевой результат.

Применение

Функция sum() может использоваться для подведения итогов и промежуточных итогов в отчетах, а также для вычисления геометрических размеров на выходной странице.

Время от времени возникает проблема, связанная с необходимостью получить сумму для значений, которые напрямую не присутствуют в исходном файле, но вычисляются на основе данных, в нем содержащихся. К примеру, если исходный документ содержит элементы <книга> с атрибутами цена и продажи, как подсчитать общий доход от продаж, который представляет сумму произведений цена на продажи для всех книг? Как сложить числа, если каждое предваряется символом «$», подлежащим предварительному удалению? Короче говоря, функция sum() не может использоваться для решения этих задач.

На деле существует два ограничения применению sum():

- Все суммируемые значения должны явно присутствовать в исходном документе в виде узлов; предварительная (к примеру, специальное преобразование строк в числа) обработка значений невозможна.
- Если одно из значений не является числом (либо значение отсутствует), результатом будет не-число.

Если исходные данные не удовлетворяют этим условиям, следует использовать другой подход. Как вариант, можно обрабатывать набор узлов явным образом с помощью рекурсивно вызываемого шаблона. Пример такой обработки – суммирования площадей фигур – приводится на стр. 585. Другой способ (связанный с использованием XSLT 1.1 либо дополнительной функции node-set(), предоставляемой многими процессорами XSLT 1.0) заключается в том, чтобы обрабатывать данные в два прохода. На первом проходе создается временное дерево, содержащее вычисленные значения, которые следует сложить, на втором проходе функция sum() вызывается для узлов дерева с целью получения окончательного ответа.

Примеры

Пример: Сетка игр

В этом примере функции count() и sum() используются для выполнения различных вычислений по результатам набора футбольных матчей (матчей группы A финала Кубка мира 1998 года).

Исходный документ

Файл футбол.xml:

```
<результаты группа="A">
<матч>
<дата>10-Jun-98</дата>
   <команда очки="2">Бразилия</команда>
   <команда очки="1">Шотландия</команда>
</матч>
<матч>
   <дата>10-Jun-98</дата>
   <команда очки="2">Марокко</команда>
   <команда очки="2">Норвегия</команда>
</матч>
<матч>
   <дата>16-Jun-98</дата>
   <команда очки="1">Шотландия</команда>
   <команда очки="1">Норвегия</команда>
</матч>
<матч>
   <дата>16-Jun-98</дата>
   <команда очки="3">Бразилия</команда>
   <команда очки="0">Марокко</команда>
</матч>
<матч>
   <дата>23-Jun-98</дата>
   <команда очки="1">Бразилия</команда>
   <команда очки="2">Норвегия</команда>
</матч>
<матч>
<дата>23-Jun-98</дата>
   <команда очки="0">Шотландия</команда>
   <команда очки="3">Марокко</команда>
</матч>
</результаты>
```

Таблица стилей

Таблица стилей хранится в файле лига.xsl.

В таблице происходит создание двух глобальных переменных: «команды», в которой хранится набор уникальных элементов <команда> из документа (элементы <команда>, строковые значения которых пересекаются со стро-

ковыми значениями уже прочитанных, не включаются в список), и «матчи», в которой хранятся все элементы <матч>.

Для каждой команды вычисляется число проведенных матчей, число побед, поражений, ничьих, общее число голов на счету команды и общее число голов, забитых командами соперников.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
>

<xsl:variable name="команды" select="//команда[not(.=preceding::команда)]"/>
<xsl:variable name="матчи" select="//матч"/>

<xsl:template match="результаты">
<html><body>
   <h1>Результаты в группе <xsl:value-of select="@группа"/></h1>

   <table cellpadding="5">
      <tr>
        <td>Команда</td>
        <td>Сыграла</td>
        <td>Победила</td>
        <td>Ничья</td>
        <td>Проиграла</td>
        <td>Забила голов</td>
        <td>Число голов противника</td>
     </tr>
   <xsl:for-each select="$команды">
        <xsl:variable name="эта" select="."/>
        <xsl:variable name="сыграла" select="count($матчи[команда=$эта])"/>
        <xsl:variable name="победы" select="count($матчи[команда[.=$эта]/
              @очки &gt; команда[.!=$эта]/@очки])"/>
        <xsl:variable name="проигрыши" select="count($матчи[команда[.=$эта]/
              @очки &lt; команда[.!=$эта]/@очки])"/>
        <xsl:variable name="ничьи" select="count($матчи[команда[.=$эта]/@очки =
              команда[.!=$эта]/@очки])"/>
       <xsl:variable name="за" select="sum($матчи/команда[.=current()]/@очки)"/>
        <xsl:variable name="против" select="sum($матчи[команда=current()]/
              команда/@очки) - $за"/>
        <tr>
        <td><xsl:value-of select="."/></td>
        <td><xsl:value-of select="$сыграла"/></td>
        <td><xsl:value-of select="$победы"/></td>
        <td><xsl:value-of select="$ничьи"/></td>
        <td><xsl:value-of select="$проигрыши"/></td>
        <td><xsl:value-of select="$за"/></td>
        <td><xsl:value-of select="$против"/></td>
        </tr>
   </xsl:for-each>
   </table>
</body></html>
```

```
</xsl:template>
</xsl:transform>
```

Вывод

***Рис. 7.5.** Результаты в группе А*

См. также

`count()` на стр. 502

system-property

Функция `system-property()` позволяет получить информацию о среде, в которой выполняется обработка.

К примеру, при использовании процессора XSLT версии 1.1 выражение «`system-property('xsl:version')`» возвращает значение `1.1`.

Определена в

XSLT, раздел 12.4

Формат

`system-property(имя)` ⇒ `любой тип`

Аргументы

Аргумент	Тип данных	Смысл
имя	строка	Имя системного свойства, для которого необходимо получить информацию. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции `string()`. Значение строки должно являться уточненным именем, указывающим свойство системы. Если свойство с таким именем отсутствует, функция возвращает пустую строку.

Результат

Тип данных результата, как и значение, зависит от свойства, информация для которого запрошена.

Правила

Аргумент функции преобразуется в полное имя на основе объявлений пространств имен в области видимости того элемента таблицы стилей, который содержит вызов `system-property()`.

Существует три системных свойства, которые обязана поддерживать каждая реализация, все три входят в пространство имен XSLT.

Имя системного свойства	Значение
`xsl:version`	Число, определяющее версию XSLT, которая реализована в процессоре. В существующих сегодня процессорах это число 1.0 или 1.1. Заметим, что это число не является строкой, поэтому при отображении с помощью `<xsl:value-of>` результатом будет «1» или «1.1».
`xsl:vendor`	Строка, определяющая поставщика процессора. На практике иногда содержит также и название продукта, но собственно значение определяется разработчиком. В MSXML3 от Microsoft значение свойства – строка «Microsoft».
`xsl:vendor-url`	Строка: URL веб-сайта поставщика. К примеру, MSXML3 возвращает «http://www.microsoft.com».

Все прочие системные свойства и их значения определяются разработчиком. Все свойства, определенные разработчиком, обычно принадлежат специальному пространству имен, уникальному для этого разработчика.

В одном из ранних вариантов спецификации XSLT было сказано, что эта функция должна предоставлять доступ к переменным окружения операционной системы. Будет неудивительно обнаружить реализацию, обладающую такими возможностями, но их наличие не является обязательным.

Применение

Функция `system-property()` может применяться для получения информации о процессоре, который выполняет таблицу стилей, либо с целью отображения этой информации (скажем, для создания комментария в выводе), либо с целью реализации логики ветвления.

В целом рекомендуется избегать применения этой функции в целях проверки доступности тех или иных возможностей. Этой цели существенно лучше служат функции `function-available()` и `element-available()`, а также инструкция `<xsl:fallback>`, а для обеспечения работы таблиц стилей с более старыми процессорами, реализующими иные диалекты XSLT, могут применяться

механизмы совместимости с последующими версиями, описанные в соответствующем разделе главы 3.

Но есть также случаи, когда вызов «system-property('xsl:version')» – единственный практичный способ выяснить, доступна ли возможность. Текущий вариант стандарта XSLT 1.1, например, содержит возможность использовать переменную с древовидным значением (фрагмент конечного дерева в терминах XSLT 1.0) в качестве набора узлов в таких контекстах, как <xsl:for-each> и <xsl:apply-templates>. Поскольку эта возможность не связана с появлением новых функций или элементов XSLT, единственный практичный способ определить ее доступность – проверить, какая версия XSLT поддерживается реализацией. Во многих процессорах XSLT 1.0 существуют фирменные эквиваленты такой возможности, реализуемые, как правило, с помощью дополнительной функции вроде «msxml:node-set()». Это позволяет создавать код следующего характера:

```
<xsl:variable name="дерево">
   . . .
</xsl:variable>
<xsl:choose>
<xsl:when test="system-property('xsl:version') &gt;= 1.1">
   <xsl:apply-templates select="$дерево"/>
</xsl:when>
<xsl:when test="function-available('msxml:node-set')">
   <xsl:apply-templates select="msxml:node-set($дерево)"/>
</xsl:when>
<xsl:when test="function-available('xt:node-set')">
   <xsl:apply-templates select="xt:node-set($дерево)"/>
</xsl:when>
<xsl:otherwise>
   <xsl:message terminate="yes">Невозможно преобразовать дерево в набор узлов</
xsl:message>
</xsl:otherwise>
</xsl:choose>
```

Примеры

Следующий фрагмент кода создает комментарий в конечном HTML-документе:

```
<HTML>
   <xsl:comment>Документ создан с помощью таблицы стилей XSLT abc.xsl
                с помощью <xsl:value-of select="system-property('xsl:vendor')"/>
                Версия XSLT
      <xsl:value-of select="system-property('xsl:version')"/>
   </xsl:comment>
. . .
```

См. также

element-available() на стр. 522

function-available() на стр. 535

<xsl:fallback> в главе 4

translate

Функция translate() заменяет указанные символы исходной строки предложенными символами. Также может использоваться для удаления указанных символов из строки.

К примеру, результатом вычисления «translate('ABC-123', '-', '/')» является строка «ABC/123».

Определена в

XPath, раздел 4.2

Формат

translate(значение, из, в) ⇒ строка

Аргументы

Аргумент	Тип данных	Смысл
значение	строка	Исходная строка. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции string().
из	строка	Список символов, подлежащих замене. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции string().
в	строка	Список символов для замены. Если аргумент имеет тип, отличный от строкового, он преобразуется в соответствии с правилами, определенными для функции string().

Результат

Строка, полученная модификацией исходной. Символы из второго аргумента заменяются соответствующими символами из третьего аргумента либо удаляются в случае отсутствия пары.

Правила

Для каждого из символов исходной строки выполняется одно из трех перечисленных действий:

- Если символ отсутствует в списке символов, подлежащих замене, он копируется в строку результата без изменений.
- Если символ присутствует в позиции P списка символов, подлежащих замене, и список заменяющих символов имеет длину, большую или равную P, в строку результата копируется символ из позиции P списка заменяющих символов.
- Если символ присутствует в позиции P списка символов, подлежащих замене, а список заменяющих символов короче, чем P, никакой символ не копируется в строку результата.

Обратите внимание, что третий аргумент является обязательным, хотя может быть пустой строкой. В этом случае из строки будут удалены все символы, входящие во второй аргумент.

Если символ более одного раза встречается в списке символов, подлежащих замене, второе и все последующие вхождения игнорируются, как и символы в соответствующих позициях списка замены (третьего аргумента).

Если третий аргумент длиннее второго, лишние символы игнорируются.

Под *символом* в этих правилах понимается XML-символ, а не 16-битное Unicode-значение. Таким образом, суррогатная пара Unicode (пара 16-битных значений, используемых для представления символа Unicode из диапазона от `#x10000` до `#x10FFFF`) считается за один символ – в любой из трех строк.

Применение и примеры

Функция `translate()` может использоваться для реализации простого преобразования регистров в случаях, когда алфавит известен и имеет разумные размеры. К примеру, если данные состоят только из латинских букв без акцентов, преобразование в верхний регистр может осуществляться с помощью такой конструкции:

```
translate($X,
     'abcdefghijklmnopqrstuvwxyz',
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Функция `translate()` может применяться для удаления лишних символов пунктуации или пробелов: к примеру, чтобы удалить все пробельные символы, дефисы и скобки из телефонного номера, можно выполнить:

```
translate($X, '&#x20;&#x9;&#xA;&#xD;()-', '')
```

Другое применение `translate()` – проверка наличия определенного символа или диапазона символов. Следующая конструкция позволяет определить, что строка содержит последовательность из трех или более цифр ASCII:

```
contains(translate($X, '0123456789', '9999999999'), '999')
```

Аналогичным образом функция может использоваться для нормализации разделителей до применения substring-after() или substring-before() в целях извлечения части строки. Если имя файла состоит из последовательности имен, разделяемых символами «/» или «\», следующее выражение извлекает первый сегмент имени (подстроку до первого разделителя):

```
substring-before(translate($X, '\', '/'), '/')
```

См. также

contains() на стр. 501
substring() на стр. 595
substring-after() на стр. 599
substring-before() на стр. 601

true

Функция возвращает логическое значение истина.

Определена в

XPath, раздел 4.3

Формат

```
true() ⇒ логическое значение
```

Аргументы

Отсутствуют

Результат

Значение логического типа истина.

Правила

В выражениях XPath не могут использоваться логические константы, но в качестве замены доступны функции true() и false().

Применение

Константы логического типа могут пригодиться в нескольких случаях. Наиболее распространенная ситуация – передача параметра шаблону с помощью <xsl:with-param>. Ниже приводится пример.

Запись <xsl:when test="true()"> может послужить полезной уловкой, если существует необходимость принудительно выбрать одну из ветвей кода (скажем, если остальные ветви еще не реализованы).

Пример

В следующем фрагменте кода происходит вызов именованного шаблона с передачей параметра «подробный-отчет», установленного в значение истина:

```
<xsl:call-template name="выполнить-работу">
   <xsl:with-param name="подробный-отчет" select="true()"/>
</xsl:call-template>
```

См. также

false(), на стр. 528

unparsed-entity-uri

Функция unparsed-entity-uri() предоставляет доступ к определениям неанализируемых сущностей из DTD исходного документа.

Если, например, DTD содержит объявление:

```
<!ENTITY карта-погоды SYSTEM "погода.jpeg" NDATA JPEG>
```

то выражение «unparsed-entity-uri('карта-погоды')» возвращает URI «weather.jpeg», который преобразуется в абсолютный идентификатор с помощью базового URI для DTD.

Определена в

XSLT, раздел 12.4

Формат

unparsed-entity-uri(имя) ⇒ строка

Аргументы

Аргумент	Тип данных	Смысл
имя	строка	Указывает имя неанализируемой сущности, для которой необходимо получить информацию. Если аргумент не является строкой, он преобразуется в соответствии с правилами, определенным для функции string(). Строка должна являться *именем* XML.

Результат

Строка, содержащая URI (системный идентификатор) неанализируемой сущности с указанным именем, если таковая существует. В противном случае возвращается пустая строка.

Правила

Если документ, которому принадлежит контекстный узел, содержит неанализируемую сущность, имя которой совпадает с передаваемой строкой, возвращается URI для этой анализируемой сущности. Если сущность с таким именем не существует, возвращается пустая строка.

Если системный идентификатор является относительным URI, XSLT-процессор должен преобразовать его в абсолютный URI и вернуть именно абсолютный идентификатор.

Применение

Неанализируемые сущности определяются в DTD с помощью объявлений вида:

```
<!ENTITY карта-погоды SYSTEM "погода.jpeg" NDATA JPEG>
```

Именно слово NDATA («не XML-данные») делает сущность неанализируемой. А неанализируемая сущность не может использоваться с помощью обычной ссылки вида «`&карта-погоды;`», вместо этого должна применяться ссылка по имени в атрибуте типа ENTITY или ENTITIES, к примеру такая: `<прогноз карта="карта-погоды">`.

Автор таблицы стилей, разумеется, должен знать, что атрибут `карта` имеет тип ENTITY (XSLT не предоставляет способа получить такую информацию), и использовать значение атрибута в вызове вроде «`unparsed-entity-uri(@карта)`». Этот вызов приводит к получению абсолютного URI конкретного ресурса, скажем, строки вида «`file:\c:\документы\прогнозы\карта.jpeg`». Этим можно воспользоваться, например, для создания HTML-элементов `<IMG>` в конечном файле.

В XSLT не существует способа узнать имя нотации (в нашем примере это «`JPEG`») или URI нотации, а также невозможно определить открытый идентификатор сущности, если таковой присутствует.

> *Это не совсем соответствует букве раздела 4.4.6 спецификации XML, в котором сказано: «Если имя неанализируемой сущности используется в качестве лексемы в атрибуте типа ENTITY или ENTITIES, подтверждающий процессор должен уведомить приложение о существующих системных и открытых идентификаторах и связанных с ними обозначениями». При этом нельзя сказать, что неанализируемые сущности являются очень популярной возможностью XML, так что неудивительно наличие в XSLT лишь минимальной ее поддержки.*

Правила, определенные спецификацией XSLT, не разрешают этого явным образом, но на практике, в случае использования не подтверждающего анализатора для обработки исходного документа, анализатор не обязан передавать XSLT-процессору информацию о неанализируемых сущностях, определенных во внешней части DTD, и в этом случае весьма вероятно, что функция `unparsed-entity-uri()` будет возвращать пустую строку. В качестве решения можно предложить применение подтверждающего XML-анализатора – разумеется, если исходный документ является действительным.

Примеры

Для определения сущности

```
<!ENTITY карта-погоды SYSTEM "погода.jpeg" NDATA JPEG>
```

и ссылки на сущность

```
<ПРОГНОЗ КАРТА="карта-погоды"/>
```

следующий код производит вставку элемента `<IMG>` в HTML-вывод:

```
<xsl:template match="ПРОГНОЗ">
   <IMG HREF="{unparsed-entity-uri(@КАРТА)}"/>
</xsl:template>
```

См. также

Раздел «Деревья, а не документы» главы 2.

Резюме

Исследование библиотеки функций, доступных XSLT-программисту, завершает справочную часть этой книги. Надеюсь, что в руках читателей оказалось полное справочное руководство по языку XSLT, охватывающее, в том числе, и язык выражений XPath. Последующие главы помогут научиться применять язык XSLT в разработке реальных приложений. В главе 8 мы рассмотрим правила создания функций расширения на языках Java и JavaScript.

8

Разработка функций расширения

В предыдущих четырех главах мы рассмотрели все средства языков XSLT и XPath. Вместе они предоставляют почти все средства, которые могут понадобиться при трансформации XML-документов. Тем не менее, иногда может возникнуть необходимость вызова из таблицы стилей кода, написанного на других языках, и назначение этой главы – объяснить, как это делается.

Черновой вариант спецификации XSLT 1.1 определяет общий механизм для вызова функций расширения, написанных на любом языке, а затем определяет точные интерфейсы для языков Java и JavaScript (также называемого ECMAScript). Интерфейсы для других языков, таких как C++, Visual Basic или Perl, могут определяться производителями, но пока не включены в стандарт.

Большинство процессоров XSLT 1.0 также предоставляет средства для вызова функций расширения. Все они слегка отличаются друг от друга, хотя не настолько, как можно было бы ожидать, так как все разработчики вместе работают над стандартом и свободно обмениваются идеями. В свою очередь, эти привязанные к производителю средства оказали сильное влияние на архитектуру стандартного интерфейса в XSLT 1.1. Я приведу краткое описание данных нестандартных механизмов в приложении для каждого продукта.

Стандарт XSLT 1.1 определяет механизмы для написания функций расширения на языках Java и JavaScript. Он не требует от разработчиков поддержки этих языков, поэтому средства, описанные в данной главе, можно найти не в каждом XSLT-процессоре. Тем не менее, если разработчик предоставляет интерфейс для Java или JavaScript, то, скорее всего, он будет соответствовать спецификациям. А это дает надежду на то, что можно будет разрабатывать функции расширения на Java, которые можно будет без изменений использовать с процессорами Saxon, Xalan или Oracle XSL, или расширения на JavaScript, переносимые между процессорами Microsoft, Xalan и Unicorn.

Это создает для сторонних разработчиков возможность создания библиотек функций расширения, которые могут быть использованы с любым из этих XSLT-процессоров.

Начнем, однако, с предупреждения об этой главе. На время написания данной книги стандарт XSLT 1.1 существует только в виде Рабочего проекта, и пока не существует соответствующих ему продуктов. Спецификация может претерпеть изменения в своей конечной версии. Вопрос определения привязок для внешних языков всегда является предметом жарких споров, и здесь также существуют возможности для коммерческих маневров, так как у каждого производителя есть свои взгляды на значимость различных языков программирования. Кроме того, так как мы работаем с предварительной спецификацией, которая еще не доступна в программных продуктах, ни один из примеров данной главы не протестирован полностью. Они могут быть легко адаптированы для работы с одним из процессоров XSLT 1.0, но это не означает, что они будут работать так, как здесь написано.

Когда необходимо применять функции расширения?

Существует несколько причин, по которым может понадобиться вызывать функцию расширения из таблицы стилей:

- Желание хранить данные во внешнем источнике, возможно, базе данных или приложении.
- Необходимость доступа к системным службам, недоступным непосредственно в XSLT или XPath. Например, может возникнуть потребность в определении текущей даты, доступе к генератору случайных чисел или добавлении записи к файлу журнала. При генерировании графики в формате SVG могут понадобиться тригонометрические функции, такие как `sin()` и `cos()`.
- Необходимость выполнения сложных вычислений, которые сложно выразить в XSLT или которые будут выполняться в нем слишком медленно. В качестве типичных примеров можно привести манипуляции со строками и датами. Энтузиасты XSLT часто записывают такие вычисления при помощи рекурсивных именованных шаблонов, но если у вас мало времени, то обращение к более привычному языку не будет преступлением.
- Сомнительной практикой является использование внешних функций для обхода правила «нет побочных эффектов» в XSLT, например, чтобы обновить значение счетчика. Избегайте этого всеми силами – если вам требуются такие средства, значит, вы еще не научились решать проблемы естественным для XSLT способом. Подробнее об этом написано в следующей главе.

Хотя в заголовке данной главы говорится о *разработке* функций расширения, очень часто случается так, что нужный код уже кем-нибудь написан. Интерфейсы для языков Java и JavaScript были тщательно спроектирова-

ны, чтобы исчерпывающие библиотеки классов, доступные в обоих языках, были доступны непосредственно в таблице стилей, без какого-либо дополнительного программирования. Это, безусловно, относится к математическим функциям, функциям манипуляции строками и датами.

Какой язык выбрать для разработки функции расширения, зависит от ваших предпочтений. При использовании процессора, написанного на Java, естественным выбором для написания функций расширения является Java. При использовании процессора MSXML3 от Microsoft естественным выбором является язык сценариев Microsoft. Если вы используете процессор 4XSLT, то вы, вероятно, предпочитаете язык Python. У процессоров, написанных на языках C и C++, обычно более сложная процедура привязки функций расширения, если они вообще поддерживаются.

Вызов функций расширения

Функции расширения всегда вызываются из выражения XPath, а синтакс вызова был описан в главе 5. Типичный вызов функции выглядит следующим образом:

```
my:function($arg1, 23, string(title))
```

В названии функции расширения всегда присутствуют префикс пространства имен и двоеточие. Префикс (например, «my») должен быть объявлен обычным способом в объявлении пространства имен какого-либо охватывающего элемента таблицы стилей. Функция может принимать любое число аргументов (ноль или более), а скобки требуются даже в том случае, когда аргументы не передаются. Аргументы могут быть любыми выражениями XPath. В нашем примере первый аргумент – это ссылка на переменную, второй – число, а третий – вызов функции. Аргументы передаются в функцию по значению, то есть функция не может изменить значения аргументов. Функция всегда возвращает значение.

Со временем я расскажу гораздо больше о типах данных аргументов и типе результата.

Какой язык выбрать?

Многие процессоры предлагают только один язык для написания функций расширения (если в них вообще существует поддержка функций расширения), поэтому выбор уже может быть сделан за вас. Некоторые процессоры предлагают нескольких языков на выбор, например Xalan поддерживает Java и JavaScript, а процессор Microsoft поддерживает любые распространенные языки сценариев, например JScript и VBScript.

В общем случае я бы посоветовал использовать «родной» язык для выбранного процессора – например, Java для Oracle, Saxon и Xalan, JScript для MSXML3, Python для 4XSLT. Если предполагается использовать таблицу

стилей с несколькими процессорами, напишите по одной версии функции расширения для каждого языка.

Если есть возможность выбора, то проще всего писать несложные функции расширения на JavaScript, потому что они могут записываться непосредственно в таблице стилей, и сам язык, как и XPath, не является статически типизированным. С другой стороны, в Java есть богатая библиотека предопределенных функций, которые несложно вызывать из таблицы стилей.

Привязка функций расширения

Когда из выражения XPath вызывается функция с именем «my:function», XSLT-процессор должен найти для вызова подходящую функцию. Этот процесс называется **привязкой (binding)**. Существует специальный самый внешний элемент <xsl:script>, назначение которого – указать процессору, где необходимо искать функцию. Подробное описание элемента <xsl:script> находится в главе 4.

Как всегда, можно выбрать любой префикс пространства имен – значение имеет только URI пространства имен. Процессор ищет элемент <xsl:script>, атрибут implements-prefix которого соответствует тому же URI пространства имен, что и у префикса, использованного в вызове функции. Если подходящий элемент <xsl:script> не был найден, то это необязательно является ошибкой: разработчику разрешается предоставлять другие механизмы для привязки функций расширения, помимо определенных в стандарте.

Если для данного пространства имен существует несколько элементов <xsl:script> и они задают разные языки, например, у одного атрибут «language» равен «java», а у другого – «javascript», то они считаются альтернативными реализациями одних и тех же функций, и система может выбрать одну из них по своему усмотрению. Это потенциально увеличивает переносимость таблиц стилей, потому что позволяет написать таблицу стилей, которая, например, будет работать с двумя процессорами, один из которых поддерживает функции расширения, написанные только на Java, а другой – только на JavaScript. Безусловно, именно вы должны обеспечить одинаковый результат двух различных реализаций.

В общем случае пространство имен соответствует совокупности родственных функций, а локальное имя функции (часть имени после двоеточия) определяет, какие из этих функций вызываются на самом деле. На этом этапе подробности начинают зависеть от языка, потому что правила для областей действия имен довольно сильно отличаются в разных языках. Для Java каждое пространство имен соответствует классу, а локальное имя – методу, определенному в этом классе. Для JavaScript пространство имен больше напоминает модуль (с технической точки зрения, это глобальный объект), а локальное имя обозначает функцию из этого модуля.

Мы опишем подробные правила привязки отдельно для каждого языка. Но перед тем как мы окунемся в разделы Java и JavaScrit, рассмотрим еще одну

область, общую для обоих языков, а именно отображение деревьев XPath в объектную модель документа (DOM).

Деревья XPath и объектная модель документа

В этой книге у нас нет места для подробного описания интерфейса DOM (Document Object Model, объектная модель документа), но большинство читателей уже наверняка сталкивалось с ним, и он подробно описан в большинстве хороших книг по XML. Интерфейс DOM предоставляет объектную модель (и, следовательно, API) для перемещения по XML-данным в древовидной форме и манипулирования ими.

Если нужно, чтобы функции расширения получили доступ к исходному дереву XSLT или чтобы было загружено дополнительное исходное дерево при помощи функции `document()`, или, даже, чтобы было создано временное дерево во время XSLT-трансформации, то это можно сделать, передав в качестве одного из аргументов функции набор узлов. Функция расширения может манипулировать этими узлами как объектами в структуре DOM. У функции расширения также есть возможность создавать новое дерево и возвращать его (обычно в виде DOM-объекта Document) вызвавшему ее выражению XPath, где им можно манипулировать так же, как и дополнительным исходным деревом, полученным в результате вызова функции XSLT `document()`.

Единственная проблема у такого подхода – как мы видели во второй главе – это то, что существуют небольшие, но существенные отличия между моделью дерева, используемой в XSLT и XPath, и моделью дерева, используемой в спецификации DOM. Например, модель DOM предоставляет доступ к ссылкам на сущности и разделам CDATA, а модель XPath – нет.

Положение усложняется еще и тем фактом, что существует два различных подхода к реализации, выбранных производителями XSLT-процессоров: оба являются совершенно допустимыми и оба необходмио принимать во внимание. Некоторые продукты, такие как MSXML3 от Microsoft, ориентированы на DOM. В Microsoft в качестве внутренней модели дерева используется DOM, а модель XPath предоставляется в виде виртуальной структуры данных (обертки) поверх нее. Это означает, например, что разделы CDATA физически присутствуют в дереве, и операции XPath, такие как `following-sibling`, динамически объединяют содержимое разделов CDATA с окружающими их текстовыми узлами. При вызове функции расширения такой продукт предоставит ей внутреннюю структуру DOM вместе с узлами CDATA и так далее. Однако другие продукты (например, Saxon) используют внутреннюю структуру данных, которая почти совпадает с моделью XPath, описанной во второй главе. В этой структуре данных не хранится информация, которая не нужна для обрабокти XPath, такая как разделы CDATA и ссылки на сущности. При вызове функции расширения ситуация меняется на противоположную: такой продукт предоставит интерфейс DOM в качестве обертки поверх внутренней модели XPath.

Невозможно скрыть все различия между этими двумя подходами. Например, ориентированный на DOM продукт, вероятно, не удалит из дерева узлы, состоящие из пробельных символов, там, где это требуется спецификацией XPath, а просто скроет их от XPath. Продукт, использующий дерево XPath, скорее всего удалит из дерева ненужные узлы, состоящие из пробельных символов, при построении дерева. Это означает, что при одном подходе удаленные узлы, состоящие из пробельных символов, будут присутствовать в модели DOM, предоставляемой функциям расширения, а при другом – отсутствовать.

Другое отличие состоит в том, что в дереве XPath прилегающие друг к другу текстовые узлы будут объединены (или нормализованы) в один узел, тогда как в дереве DOM они могут оставаться ненормализованными. (На самом деле во время написания этой книги MSXML3 не всегда правильно нормализует текстовые узлы, даже в предоставляемом XPath отображении.)

Все это означает, что если вы хотите, чтобы функция расширения была переносима между различными процессорами, необходимо быть в курсе существующих различий и обходить их. Спецификация XSLT 1.1 перечисляет области возможных отличий между представлением DOM и представлением XPath следующим образом:

Узел XPath	Узел DOM	Соответствие
Корневой узел	Document	Один к одному.
Элемент	Element	Один к одному.
Атрибут	Attr	В дереве XPath объявления пространств имен никогда не представляются как атрибуты `xmlns` или `xmlns:*`. В дереве DOM такие узлы `Attr` могут присутствовать, а могут и не присутствовать. Если в исходном документе использовались ссылки на сущности в значении атрибута, они могут остаться в модели DOM, а могут и не остаться. Значение свойства `getSpecified` в модели DOM не определено.
Текст	Text	В дереве DOM текстовые узлы могут быть или не быть нормализованными. Если в исходном документе присутствовали разделы CDATA, узлы CDATA могут или присутствовать, или нет в модели DOM. Если в исходном документе использовались ссылки на сущности в текстовом значении, они могут остаться в модели DOM, а могут и не остаться. Узлы, состоящие из пробельных символов, которые должны быть удалены по требованиям XSLT, могут присутствовать или не присутствовать в виде текстовых узлов в модели DOM.

Узел XPath	Узел DOM	Соответствие
Инструкция обработки	Инструкция обработки	Один к одному.
Комментарий	Комментарий	Один к одному.
Пространство имен	Недоступно	В DOM не существует структуры, непосредственно эквивалентной узлам пространств имен XPath. В модели DOM элементы и атрибуты могут использовать URI пространств имен, которые не объявлены где-либо в дереве.
Недоступно	CDATASection (Секция CDATA)	Узлы CDATASection могут присутствовать в дереве DOM, если оно является внутренней структурой данных, используемой процессором, но они вряд ли будут в нем присутствовать, если процессор конструирует дерево DOM из дерева XPath.
Недоступно	Entity Reference (Ссылка на сущность)	Узлы EntityReference могут присутствовать в дереве DOM, если оно является внутренней структурой данных, используемой процессором, но они вряд ли будут в нем присутствовать, если процессор конструирует дерево DOM из дерева XPath.

При вызове методов, определенных в DOM, результат будет следовать правилам DOM, а не правилам XPath. Например, в XPath строковое значение узла элемента является объединением всего текстового содержимого элемента, а в DOM подобный метод `nodeValue()` возвращает `null`.

Функции расширения, которым передаются ссылки на узлы из дерева DOM, не должны пытаться изменить его через них, потому что результат таких действий не определен. На практике могут случиться три вещи:

- Попытка изменения дерева DOM может вызвать ошибку.
- Если дерево DOM является копией дерева XPath, обновление может выполниться успешно, но не изменить дерева, видимого из таблицы стилей.
- Если деревья DOM и XPath являются различными представлениями одних и тех же данных, то обновление может повлиять на последующую обработку XSLT. Это может вызвать ошибки, например, если будут удалены узлы, ссылки на которые еще используются XSLT-процессором.

Функция расширения может сконструировать новое дерево DOM и вернуть его в таблицу стилей как значение функции.

Если возвращаемое значение является объектом Node или NodeList модели DOM, то каждый узел обрабатывается следующим образом:

- Сначала процессор определяет местонахождение корня дерева DOM, в котором расположен узел, который должен быть объектом `Document` или `DocumentFragment`.
- Затем процессор решает, является ли данный корневой узел корневым узлом существующего документа XPath, который уже известен таблице

стилей, потому что рассматриваемое дерево DOM было получено из документа XPath или потому что оно было возвращено одним из предыдущих вызовов какой-либо функции расширения.

- Если документ уже известен, процессор находит в нем узел XPath, соответствующий объекту `Node`, возвращенному функцией расширения.
- Если же документ неизвестен таблице стилей, процессор пытается загрузить документ как дополнительный исходный документ, как если бы он загружался при помощи функции `document()`. Процессору позволяется отвергнуть документ, если он был создан неподдерживаемой реализацией DOM (например, процессор Oracle может отвергнуть дерево DOM, созданное при помощи Xerces). Это правило формулируется следующим образом: успешное выполнение операции гарантируется только в том случае, если корнем дерева является объект `DocumentFragment`, свойство `ownerDocument` которого возвращает тот же самый объект `Document`, что и свойство `ownerDocument` объекта `XSLTContext`.
- Когда таким образом импортируется новый документ, создаются узлы пространств имен, соответствующие всем используемым пространствам имен для имен элементов и атрибутов, так же как и для любых пространств имен, объявленных при помощи атрибутов `xmlns:*`. Узлы, состоящие из пробельных символов, не удаляются. После того как документ был передан XSLT-процессору, он не должен изменяться, хотя процессор не обязан определять такую ошибочную ситуацию.

Данные правила отображения деревьев XPath кажутся довольно запутанными, и они, конечно, являются источником многих возможных ошибок. Мой совет – избегать этой области XSLT, если это возможно. Перемещение по дереву является задачей, с которой вполне можно справиться средствами XSLT и XPath, и нет необходимости прибегать для этого к другому языку программирования. Обычно проще и вполне достаточно передавать в функцию расширения обычные строки и числа.

При вызове функций JavaScript, аргументы которых нетипизированы, может оказаться полезным применение функций XPath для преобразования аргументов функции к требуемому типу. Например, если функция `my:toUpperCase()` принимает в качестве аргумента строку, пишите не `«my:toUpperCase(@desc)»`, а `«my:toUpperCase(string(@desc))»`. Это позволяет избежать применения правил для наборов узлов. Хотя в Java приведение типов выполнится автоматически.

Если нужно написать функцию расширения, которая создает и возвращает новое дерево, более простой альтернативой может оказаться применение функции `document()` и реализация интерфейса `URIResolver`, который принимает URI, переданный этой функции, и возвращает подходящий источник данных. Интерфейс TrAX, описанный в приложении F, определяет стандартный способ выполнения таких операций.

Однако если это нужно сделать именно в функции расширения, необходимо создавать узлы DOM при помощи свойства `ownerDocument` объекта `XSLTContext`.

Создайте объект DocumentFragment, который будет корнем нового дерева, а затем добавляйте к нему следующие узлы. Наиболее вероятным действием по завершении является возвращение созданного объекта DocumentFragment, хотя можно вернуть любой объект Node или NodeList из нового дерева.

Привязки для языка Java

В этом разделе подробно описывается интерфейс между вызовами функций XPath и классами и методами Java. Это позволяет вызывать методы Java из таблицы стилей.

Процесс вызова метода, написанного на Java, делится на четыре стадии:

- Использование префикса пространства имени функции XPath для определения класса Java.
- Использование локальной части имени функции XPath и, если это необходимо, типов данных переданных аргументов для определения метода, принадлежащего этому классу.
- Преобразование переданных аргументов в объекты Java.
- Обработка возвращаемого значения метода и любых ошибок, которые он может сгенерировать.

Мы рассмотрим все эти четыре стадии по очереди.

Привязка для языка Java была спроектирована для удовлетворения двух слегка различающихся целей:

- Во-первых, для упрощения вызова методов из стандартной библиотеки классов Java непосредственно из таблицы стилей. Это позволяет выполнять типичные задачи, такие как получение текущей даты, случайного числа, выполнение тригонометрических вычислений или проверку существования файла.
- Во-вторых, для упрощения написания новых методов Java, вносящих в таблицу стилей логику, характерную для приложений. Например, метода, которому дается код товара, а он возвращает описание товара, выполняя SQL-запрос к базе данных товаров.

Результатом воплощения первой цели явился интерфейс, позволяющий вызвать почти любой метод Java. Главным ограничением этого интерфейса является то, что если метод сгенерирует ошибку, то выполнение таблицы стилей завершится. Результатом воплощения второй цели явились средства, позволяющие коду на Java узнавать подробности контекста обработки XSLT и XPath, к чему мы вернемся на стр. 642.

Определение класса Java

Как мы уже видели, при вызове функции расширения всегда используется имя с префиксом, например «my:function()». Префикс должен быть объявлен в объявлении пространства имен, находящемся в области действия данного

места таблицы стилей, и процессор использует данное объявление пространства имен для определения соответствующего URI пространства имен. Например, если используется объявление пространства имен «xmlns:my="http://schmidt.de/functions"», то это и будет URI пространства имен, который мы ищем.

> *Я бы посоветовал использовать обычно URI пространства имен, задающее непосредственно имя класса, например* «xmlns:math=java:java.lang.Math»*. Однако класс и пространство имен – это разные вещи, поэтому присваивание им одинакового имени запутает данное объяснение. Нельзя перегружать имена подобным образом, пока еще нет четкого понимания того, что происходит.*

Процессор ищет элемент <xsl:script>, атрибут implements-prefix которого задает префикс, соответствующий тому же URI пространства имен. Конечно, в большинстве случаев префиксы будут совпадать, однако значение имеет только URI пространства имен.

Если процессор находит для данного пространства имен элемент <xsl:script> и его атрибут language равен «java», то атрибут src элемента <xsl:script> определяет полностью заданное имя класса Java, с префиксом «java:» для того, чтобы он являлся допустимым URI.

Например, для следующего вызова функции:

```
<xsl:value-of select="dt:new()"/>
```

может существовать следующий элемент <xsl:script>:

```
<xsl:script language="java" implements-prefix="dt" src="java:java.util.Date"/>
```

Это объявление означает, что требуемым Java-классом является класс java.util.Date. Это класс обычно находится в пути к классам. В качестве альтернативного варианта можно задавать путь явно, используя атрибут archive, это особенно полезно при использовании процессора, встроенного в веб-броузер, потому что позволяет загружать классы с сервера по мере необходимости.

Если для одного и того же пространства имен существует несколько элементов <xsl:script>, процессор выбирает элемент с наибольшим приоритетом импортирования. Если для одного элемента определено несколько элементов <xsl:script> с одинаковым приоритетом, то это является ошибкой.

Выбор метода Java

Локальная часть имени в вызове функции XPath используется для определения вызываемого метода.

Если в вызове используется имя «new», то процессор ищет открытые конструкторы, а не открытые методы, во всех остальных случаях он ищет метод с соответствующим именем. Поиск соответствия происходит следующим образом:

Прежде всего, из имени удаляются все дефисы, а любой символ, следующий за дефисом, преобразуется к верхнему регистру. Например, если вызов

функции XPath записан как `acct:get-account-number()`, то процессор будет искать метод Java, названный `getAccountNumber()`. Это косметическая операция, позволяющая использовать в обоих языках привычные соглашения именования методов. Если вы знаете, что хотите вызвать метод `getAccountNumber()`, то можно записать вызов как `acct:getAccountNumber()`, если это вам нравится.

Далее процессор находит все открытые методы с данным именем в выбранном классе. Если ни одного не найдено, то это является ошибкой. Безусловно, в Java-классе может находиться несколько методов с одинаковым именем и разными аргументами (эта возможность обычно называется перегрузкой методов), поэтому, когда возникает такая ситуация, процессор должен также принять во внимание аргументы вызова.

Затем система исключает из рассмотрения все методы с неподходящим числом аргументов. Это необязательно означает, что у него должно быть столько же аргументов, сколько у вызова функции. Существует два фактора, усложняющих обработку:

- Если первым аргументом в сигнатуре метода Java является объект класса `org.w3c.xsl.XSLTContext`, то данный аргумент ни на что не влияет. Процессор подставит значение данного аргумента, как объясняется на стр. 642, поэтому он не соответствует никакому аргументу вызова функции XPath.
- Если рассматриваемый метод Java является методом экземпляра (в отличие от статического метода или конструктора), то вызов функции XPath должен передать дополнительный первый аргумент, который не используется при подсчете аргументов. Данный дополнительный аргумент должен быть внешним объектом, который используется как цель вызова метода.

Любые методы с неподходящим числом аргументов будут исключены из рассмотрения после выполнения данных подсчетов.

Если после этого осталось более одного подходящего метода, выбор начинает зависеть от типов данных аргументов вызова функции XPath. Так как XPath является динамически типизируемым языком, типы данных аргументов не всегда могут быть определены заранее, так что данная заключительная стадия может совершаться во время выполнения. Потенциально один и тот же вызов функции XPath может в разных случаях вызывать различные методы Java.

Основным принципом является выбор метода, объявленные типы аргументов которого наилучшим образом соответствуют типам переданных аргументов. Фактический алгоритм похож на используемый в самом языке Java. Хотя в спецификации правила даются иначе, мне нравится объяснять действие данного алгоритма, показывая, что он выбирает метод, требующий наименьших усилий по преобразованию аргументов.

На преобразование между типами данных XPath и классами Java необходимы некоторые затраты: затраты равны нулю, если типы точно соответствуют друг другу, и бесконечности, если преобразование не разрешено. Так, для каждого подходящего метода можно вычислить затраты преобразования для

каждого аргумента (фактические значения «затрат» даны в следующем разделе). Если тип данных XPath переданного аргумента – *A*, а Java-класс требуемого аргумента – *P*, то затраты на преобразование являются функцией *A* и *P*: скажем, *затраты(A, P)*. Таблица, начинающаяся на стр. 629, показывает, что, например, *затраты(число, double)*=0 и *затраты(число, String)*=16.

Представьте, что типы данных XPath переданных аргументов – это A_1, A_2 и A_3, и что мы выбираем из двух методов Java: метода *P* с объявленными аргументами P_1, P_2 и P_3 и метода *Q* с аргументами Q_1, Q_2, и Q_3.

Мы не просто подсчитываем суммарные затраты, складывая затраты преобразования для каждого аргумента. Скорее, мы должны найти метод, у которого для каждого аргумента по отдельности затраты на преобразование для данного аргумента не хуже, чем у любого другого метода.

Если для каждого аргумента с номером n=1..3 (в нашем примере) либо P_n является тем же классом, что и Q_n, либо *затраты*(A_n, P_n) < *затраты*(A_n, Q_n), то метод *P* предпочтительнее метода *Q*. (В обоих методах должен существовать по крайней мере один отличающийся аргумент, в противном случае компилятор Java отвергнет методы, как повторяющиеся.) Если существует один аргумент, для которого *затраты*(A_n, P_n) < *затраты*(A_n, Q_n), и другой, для которого *затраты*(A_m, P_m) > *затраты*(A_m, Q_m), то оба метода, *P* и *Q*, не являются предпочтительными.

Если при использовании данных правил один подходящий метод является более предпочтительным, чем другой, то он и будет выбран. Если не один из методов не является предпочтительным, то выводится сообщение об ошибке.

Если говорить конкретно, представим, что у нас есть вызов функции

```
date:compare('20010405', '20011011')
```

и что у нас есть четыре подходящих метода в соответствующем классе:

A	compare(int, int)
B	compare(int, String)
C	compare(String, int)
D	compare(String, String)

Интуитивно метод *D* является наиболее подходящим, так как оба переданных аргумента являются строками. Оказывается, что он также является наиболее подходящим, если выполнить вычисления, потому что затраты на преобразование (взятые из таблицы на стр. 629), следующие:

Метод	Затраты на преобразование первого аргумента	Затраты на преобразование второго аргумента
A	4	4
B	4	0

Метод	Затраты на преобразование первого аргумента	Затраты на преобразование второго аргумента
C	0	4
D	0	0

Метод *D*, и только *D*, удовлетворяет правилу для каждого аргумента, затраты на преобразование меньше или равны затратам на преобразование того же аргумента в других методах. Поэтому выбран метод *D*.

Если бы мы могли выбирать только из методов *A*, *B* и *C*, то не существовало бы однозначного победителя, потому что *C* был бы лучшим по первому аргументу, тогда как *B* был бы лучшим по второму аргументу. Поэтому, если метод *D* был бы недоступен, то вызов функции вызвал бы ошибку.

Таблица затрат на преобразование

Нижеследующая таблица используется для вычисления затрат на преобразование значения XPath в объект (или значение) Java. Типы данных XPath передаваемого аргумента находятся в первой строке таблицы, классы Java требуемого аргумента находятся в первом столбце. Числа в таблице являются, конечно же, относительными показателями стоимости преобразования. Мы заинтересованы только в знании того, что преобразовать логическое значение в объект типа String дешевле, чем в объект типа Float, а фактические значения несущественны.

	логический	числовой	строковый	набор узлов	внешний
boolean	0	14	6	8	–
Boolean	0	15	7	9	–
byte	3	12	4	6	4
Byte	4	13	5	7	5
char	–	10	2	4	2
Character	–	11	3	5	3
double	3	0	4	6	4
Double	4	1	5	7	5
float	3	2	4	6	4
Float	4	3	5	7	5
int	3	6	4	6	4
Integer	4	7	5	7	5
long	3	4	4	6	4
Long	4	5	5	7	5

	логический	числовой	строковый	набор узлов	внешний
Node (примечание 1)	–	–	–	1	–
NodeList	–	–	–	0	–
Object	1	17	1	3	0 (примечание 2)
short	3	8	4	6	4
Short	4	9	5	7	5
String	2	16	0	2	1

Примечание 1: это класс DOM `org.w3c.dom.Node` или любой из его подклассов. В строке под ним тип `NodeList` является классом DOM `org.w3c.dom.NodeList`.

Примечание 2: при условии, что класс переданного объекта можно присвоить объекту требуемого класса.

Там, где в таблице находится прочерк, преобразование невозможно, поэтому затраты на преобразование равны бесконечности.

Правила преобразования аргументов

У нас есть таблица, показывающая относительную стоимость затрат на любой возможный вариант преобразования типов, но нам необходимо объяснить, как происходят сами преобразования. Что означает, например, преобразование логического значения XPath в тип byte Java?

Правила преобразования приведены в следующих разделах, упорядоченные в соответствии с типом данных XPath переданного аргумента.

Преобразование из логического типа XPath

Конечный тип данных	Правила преобразования
`boolean` или `Boolean`	В преобразовании нет необходимости.
`Object`	Переданное значение передается в виде значения типа `Boolean`: либо `Boolean.TRUE`, либо `Boolean.FALSE`.
`String`	Переданное значение преобразуется в строку «`true`» или «`false`».
`double, float, long, int, short, byte, Double, Float, Long, Integer, Short, Byte`	Ложь преобразуется в `0`, истина в `1`.

Преобразование из числового типа XPath

Конечный тип данных	Правила преобразования
`double, Double`	В преобразовании нет необходимости.

Конечный тип данных	Правила преобразования
`float, Float`	Значение округляется до точности, подходящей для значения типа float. Значения, находящиеся вне диапазона типа float, преобразуются в плюс или минус бесконечность.
`long, Long, int, Integer, short, Short, char, Character, byte, Byte`	Значение приводится к целому типу путем отбрасывания дробной части (с округлением в сторону нуля). Если значение находится вне диапазона целевого типа, возникает ошибка. Ошибка также возникает в случае, если переданное значение является не-числом.
`boolean, Boolean`	Число преобразуется как при использовании функции `boolean()` XPath – ноль становится значением `false`, а любое другое число – `true`.
`String`	Число преобразуется в строку как при использовании функции `string()` XPath.
`Object`	Число передается в виде объекта типа `Double`.

Преобразование из строкового типа XPath

Целевой тип данных	Правила преобразования
`String`	В преобразовании нет необходимости.
`Object`	Число передается в виде объекта типа `String`.
`char, Character`	Если строка состоит из одного символа, то в качестве значения передается именно этот символ. Если строка нулевой длины или длиннее одного символа, возникает ошибка.
`double, float, long, int, short, byte, Double, Float, Long, Integer, Short, Byte`	Строка XPath преобразуется в число XPath с использованием функции `number()`, а затем получившееся число преобразуется в нужный тип Java, используя правила из предыдущей таблицы.
`boolean, Boolean`	Строки нулевой длины преобразуются в значение `false`, все остальные – в значение `true`.

Преобразование набора узлов XPath

Целевой тип данных	Правила преобразования
`org.w3c.dom.NodeList`	Узлы в наборе узлов представляются в виде объекта `NodeList` DOM. Список будет упорядочен в порядке документа. Может возникнуть ошибка, если в наборе узлов находятся узлы пространств имен.

`org.w3c.dom.Node`
`org.w3c.dom.Attr`
`org.w3c.dom.CDATASection`
`org.w3c.dom.CharacterData`
	Если набор узлов пуст, то возникает ошибка. В противном случае используется первый узел набора (в порядке документа), а остальные игнорируются. Если это узел неподходящего типа, возникает ошибка (например, если предается узел элемен-

Целевой тип данных	Правила преобразования
org.w3c.dom.Comment org.w3c.dom.Document org.w3c.dom.Element org.w3c.dom.ProcessingInstruction org.w3c.dom.Text	та, а метод принимает в качестве аргумента объект типа Document).
String	Набор узлов преобразуется в строку по правилам функции string() XPath. (Результатом является строковое значение первого узла, если в наборе есть хотя бы один узел, или пустая строка, если набор пуст.)
Object	Переданный объект будет иметь тип org.w3.dom.NodeList.
char, Character	Набор узлов преобразуется в строку по правилам функции string() XPath. Если в строке находится ровно один символ, передается именно он, в противном случае возникает ошибка.
double, float, long, int, short, byte, Double, Float, Long, Integer, Short, Byte	Набор узлов преобразуется в число при помощи функции number() XPath, а затем оно преобразуется в требуемый тип Java так же, как если бы в качестве аргумента передавалось число. Это вызовет ошибку, если значение не является числом.
boolean, Boolean	Переданное значение станет значением false, если набор узлов пуст, и true, если в нем находится хотя бы один узел.

Преобразование внешнего объекта

Целевой тип данных	Правила преобразования
Любой Java-класс C, такой, что выражение «OBJ instanceof C» истинно, где OBJ – упакованный внешний объект.	Распаковать внешний объект.
String	Вызвать метод toString() упакованного объекта.
char, Character	Вызвать метод toString() упакованного объекта. Использовать данное значение, если его длина равна ровно одному символу, в противном случае вывести сообщение об ошибке.
double, float, long, int, short, byte, Double, Float, Long, Integer, Short, Byte	Вызвать метод toString() упакованного объекта, затем преобразовать полученную строку в число по правилам функции number() XPath. Преобразовать полученное число в требуемый тип Java, как если бы в качестве аргумента было передано числовое значение XPath.

Обработка возвращаемого значения

Если метод Java генерирует исключение, возникает ошибка. У таблицы стилей не существует возможности перехватить эту ошибку и продолжить обработку. На практике это означает, что не стоит вызывать методы, генерирующие исключения, за исключением следующих случаев: (а) вы можете предотвратить возникновение исключения, гарантируя передачу правильных аргументов, или (б) в любом случае возобновление работы таблицы стилей после данной ошибки невозможно.

Если это является проблемой, ее можно обойти, написав метод-оболочку на Java, перехватывающий исключение и сигнализирующий об ошибке возвращаемым значением, например, возвращая специальное значение, такое как –1.

Если метод не возвращает значения, то результатом функции XPath является пустой набор узлов. (Единственное назначение вызова ничего не возвращающего метода – это использование побочных эффектов вызова, поэтому будьте осторожны, смотрите замечания о функциях с побочными эффектами на стр. 640.)

Если метод Java возвращает значение null, то результатом функции XPath является внешний объект, являющийся значением null. К сожалению, не существует простого способа проверить внешний объект на равенство null, кроме как передавая его другой функции расширения. Более того, в Java нет простого метода, который можно вызвать, чтобы проверить, является ли объект значением null. Я изучил документацию по Java API вдоль и поперек, и единственное, что я смог найти, – это статический метод `identityHashCode()` класса `java.lang.System`, который возвращает ноль, если аргумент является значением null. Поэтому можно написать:

```
<xsl:stylesheet...
  xmlns:sys="java:java.lang.System"
  xmlns:my="java:name.joe-programmer.ExtensionClass">
<xsl:script language="java" implements-prefix="sys"
            src="java:java.lang.System"/>
      . . .
<xsl:variable name="result" select="my:method($x)"/>
<xsl:if test="sys:identityHashCode($result) = 0">
   <xsl:message>метод my:method() вернул null</xsl:message>
</xsl:if>
```

Простые возвращаемые значения преобразуются в значения XPath следующим образом:

Значение, возвращенное Java-методом	Результат вызова функции XPath
`String`	Эквивалентная строка. Спецификация намеренно не определяет, что происходит, если в строке находятся символы, допустимые в Java, но не в XML, например, уп-

Значение, возвращенное Java-методом	Результат вызова функции XPath
	равляющий символ NUL ASCII (`#x00`). Если в строке находятся такие символы, то это ошибка, однако процессор не обязан сообщать о ней.
`double`, `float`, `long`, `int`, `short`, `byte`, `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`	Эквивалентное число XPath.
`boolean`, `Boolean`	Эквивалентное логическое значение XPath.

Если Java-метод возвращает объект `NodeList` DOM или просто узел `Node`, то правила усложняются, и во время написания книги они были определены не очень четко. Вот некоторые из затруднений:

- Многие продукты могут обрабатывать только те структуры данных DOM, которые были созданы с использованием продукта конкретного производителя. В общем случае нельзя ожидать, что процессор Oracle XSLT обработает переданную ему модель DOM, созданную анализатором XML Xerces: он примет только модель DOM, созданную при помощи анализатора Oracle.
- Что случится, если вы вызовете последовательность функций расширения и несколько функций вернут один и тот же узел DOM? Вернет ли каждый вызов один и тот же узел XPath (с одинаковым значением, возвращаемым для него функцией `generate-id()`) или разные узлы XPath? Для продукта, использующего DOM в качестве внутренней структуры данных, это, вероятно, не является большой проблемой, так что вы будете получать каждый раз один и тот же узел. Но для продукта, выполняющего отображение или преобразование между структурой DOM и собственным внутренним представлением, существуют проблемы сохранения тождественности узлов, особенно из-за того, что спецификация Java DOM не гарантирует, что один и тот же объект в структуре DOM будет всегда представляться одним и тем же объектом Java – спецификация DOM намеренно написана так, чтобы узлы можно было создавать по требованию, например путем извлечения данных из базы данных.

Спецификация XSLT позволяет функциям расширения возвращать объекты `NodeList` или отдельные узлы `Node`, но не отвечает на приведенные выше вопросы, поэтому если использовать эту возможность, то нельзя быть уверенным, что таблица стилей окажется безболезненно переносимой.

Использование функций расширения Java

Простейшими функциями расширения, которые необходимо использовать всегда, когда это возможно, являются методы без сохранения состояния, которые не получают дополнительной информации из своего окружения и не оставляют после себя ничего, кроме возвращенного значения. Лучше всего записывать их как статические методы, и мы рассмотрим их первыми.

Для более сложных функций может возникнуть необходимость сохранения информации между вызовами: например, если вы хотите сгенерировать последовательность случайных чисел. Обычно это требует написания метода экземпляра, поэтому мы обсудим их позже.

Вызов статических методов

Для вызова статического метода необходимо знать имя класса, имя метода и ожидаемые аргументы.

Типичным примером является метод `max(a, b)`, находящийся в классе `java.lang.Math`. Он принимает два числа в качестве аргументов и возвращает наибольшее из них.

Данную функцию можно вызвать следующим образом:

```
<xsl:script implements-prefix="Math" xmlns:math="java:java.lang.Math"
            language="java" src="java:java.lang.Math"/>
. . .
<xsl:template match="price">
   <xsl:value-of select="Math:max(number(@net-price), number(@gross-price))"
                 xmlns:Math="java:java.lang.Math"/>
</xsl:template>
```

Я начал использовать свое обычное соглашение об использовании короткой формы имени класса Java в качестве префикса пространства имен, а длинной формы (с префиксом «`java:`») в качестве URI-пространства имен. Помните, что это всего лишь соглашение, вы можете использовать любой префикс и любой URI, какой пожелаете.

На самом деле в классе `java.lang.Math` существует четыре метода с названием `max()`, которые являются альтернативными версиями с аргументами типа `double`, `float`, `int` и `long`, соответственно. В данном случае будет вызываться метод с аргументами типа `double` и возвращающий значение типа `double`, потому что это минимизирует затраты на преобразование.

Заметьте, что оба переданных нами аргумента являются числами XPath. Если бы мы не использовали функцию `number()` для преобразования их в числа, то они были бы наборами узлов. Глядя на таблицу на стр. 629, видно, что затраты на преобразование набора узлов XPath в тип Java `double`, `float`, `long` или `int` равны. Это означает, что ни один из методов не был бы предпочтительнее другого, и возникла бы ошибочная ситуация.

Статические методы, такие как `max()`, хорошо укладываются в модель обработки XSLT, потому что у них нет побочных эффектов. Не имеет значения, как часто они вызываются, одна и та же функция с одними и теми же аргументами всегда возвращает один и тот же результат. Поэтому нет необходимости беспокоиться о порядке выполнения инструкций в таблице стилей. Как мы уже видели, это можно сказать не о всех функциях расширения.

Вызов конструкторов и методов экземпляров

Можно вызвать конструктор Java для создания нового экземпляра класса, как если бы он был статическим методом с именем `new()`.

Результатом вызова конструктора является объект Java, упакованный в значение XPath специального типа, – **внешний объект**. Единственное, что можно сделать с внешним объектом – это передать его другой функции расширения. Можно передавать его в виде аргумента функции или вызывать его методы экземпляра (то есть не статические). Чтобы вызвать метод экземпляра, необходимо передать объект в качестве первого аргумента вызову функции XPath, а затем значения обычных аргументов, если они есть.

Предположим, например, что нужно вывести текущую дату. У класса `java.util.Date` есть конструктор по умолчанию, создающий объект `Date`, инициализированный текущими временем и датой, а также метод экземпляра `toString()`, выводящий дату в формате «`Sat Jan 06 14:32:29 GMT 2001`». Можно использовать функцию `substring()` XPath для извлечения частей даты, которые необходимо показать.

Вот код таблицы стилей, выполняющий эту задачу:

```
<xsl:script language="java" implements-prefix="Date"
   src="java:java.util.Date" xmlns:Date="java:java.util.Date"/>
...
<xsl:template match="/">
  <xsl:variable name="date" select="Date:toString(Date:new())"
                            xmlns:Date="java:java.util.Date"/>
  <xsl:comment>Вывод сгенерирован в <xsl:value-of select="
        concat(substring($date, 12, 5), ' ',
               substring($date, 9, 2), ' ',
               substring($date, 5, 3), ' ',
               substring($date, 25, 4))"/></xsl:comment>
  . . .
</xsl:template>
```

Этот код вставит в выходной файл комментарий вида:

```
<!-- Вывод сгенерирован в 14:32 06 Jan 2001-->
```

В листинге присутствуют два вызова функций: вызов `Date:new()`, вызывающий конструктор класса `java.util.Date` с нулевым количеством аргументов для создания объекта `Date`, и вызов `Date:toString()`, принимающий в качестве единственного аргумента данный объект. Поскольку метод `Date:toString()` является методом экземпляра, первым аргументом вызова функции XPath должен быть объект `Date`, и данный объект `Date` становится целью вызова метода.

Значение, возвращенное первым вызовом функции и переданное во второй, не является стандартным типом данных XPath – это внешний объект. В данном примере мы использовали внешний объект сразу, напрямую передавая его другому вызову функции, но мы также могли бы сохранить его в пере-

менной для дальнейшего использования или передать его в качестве параметра шаблону. Однако при сохранении внешнего объекта в переменной надо позаботиться о том, чтобы над ней не выполнялись никакие операции XPath: очевидно, что такие безобидные операции, как сравнение ее с другим значением или преобразование в строку или число, приведут к ошибке.

Хотя данный пример требует расширения множества типов данных XPath для включения в него понятия внешнего объекта, он все еще хорошо вписывается в модель обработки XSLT, потому что в обоих вызовах функций отсутствуют побочные эффекты. В данном конкретном случае утверждение о том, что мы получаем каждый раз один и тот же результат, неверно – цель данной функции возвращать результат, зависящий от времени ее вызова. Однако мы пока еще в безопасности. В следующем примере мы окажемся на более опасной территории.

Вызов внешних функций в цикле

В следующем примере мы будем использовать объект Java `BufferedReader` для чтения внешнего файла и построчного копирования его в выходное дерево, причем за каждой строчкой будет выводиться пустой элемент `<br/>`.

Пример: Вызов внешних функций в цикле

Исходный документ

Таблица стилей не обращается к первичному исходному документу, поэтому нам подойдет любой XML-файл. Например, можно использовать в качестве исходного документа даже саму таблицу стилей.

Настоящий исходный документ является последовательным файлом, которым может быть любой текстовый файл. Например, следующий `hiawatha.txt`:[1]

```
Лук возьми свой, Гайавата,
Острых стрел возьми с собою,
Томагаук, Поггэвогон,
Рукавицы, Минджикэвон,
И березовую лодку.
Желтым жиром Мише-Намы
Смажь бока ее, чтоб легче
Было плыть ей по болотам,
И убей ты чародея,
Отомсти врагу Нокомис,
Отомсти врагу народа!
```

Таблица стилей

Таблица стилей называется `reader.xsl`.

[1] В русском издании книги использовался перевод И. А. Бунина. – *Примеч. ред.*

Сначала мы объявим необходимые пространства имен. Так как пространства имен используются в элементе <xsl:script> и в инструкции XSL, вызывающей функцию расширения, часто бывает проще объявить эти пространства имен в самом элементе <xsl:stylesheet>. Я буду использовать свое соглашение об использовании одного и того же URI «java:*» для задания местоположения класса Java и в качестве его URI пространства имен, а также использовании сокращенного имени класса в качестве префикса пространства имен. Вряд ли необходимо, чтобы эти префиксы появлялись в выходном документе, поэтому их можно выключить при помощи атрибута exclude-result-prefixes.

```
<xsl:stylesheet version="1.1"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:FileReader="java:java.io.FileReader"
   xmlns:BufferedReader="java:java.io.BufferedReader"
xmlns:System="java:java.lang.System"
       exclude-result-prefixes="FileReader BufferedReader System">
<xsl:script language="java" implements-prefix="FileReader"
       src="java:java.io.FileReader"/>
<xsl:script language="java" implements-prefix="BufferedReader"
       src="java:java.io.BufferedReader"/>
<xsl:script language="java" implements-prefix="System"
       src="java:java.lang.System"/>
```

Имя исходного файла будет передаваться в параметре таблицы стилей:

```
<xsl:param name="filename"/>
```

Когда мы готовы начинать чтение файла, мы создаем в переменной объект BufferedReader. Затем мы вызываем шаблон для построчного чтения файла:

```
<xsl:template match="/">
<out>
    <xsl:variable name="reader"
         select="BufferedReader:new(FileReader:new($filename))"/>
    <xsl:call-template name="read-lines">
         <xsl:with-param name="reader" select="$reader"/>
    </xsl:call-template>
</out>
</xsl:template>
```

Именованный шаблон читает и выводит первую строку файла, а затем рекурсивно вызывает сам себя для обработки оставшихся строк. Единственной проблемой является то, что метод readLine() класса BufferedReader возвращает null, когда достигает конца файла, а в XPath нет простого способа проверить, является ли внешний объект Java значением null. Поэтому мы будем использовать трюк с System.identityHashCode(), о котором я говорил ранее: эта функция ноль, если ее аргумент – null.

```
<xsl:template name="read-lines">
   <xsl:param name="reader"/>
```

```
    <xsl:variable name="line" select="BufferedReader:readLine($reader)"/>
    <xsl:if test="System:identityHashCode($line) != 0">
         <xsl:value-of select="$line"/><br/>
         <xsl:call-template name="read-lines">
             <xsl:with-param name="reader" select="$reader"/>
         </xsl:call-template>
     </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Конечный документ

При запуске данной таблицы стилей ей необходимо передать параметр, являющийся именем файла. Например:

```
somexsltproc reader.xsl reader.xsl filename=hiawatha.txt
```

(Я не написал, какой XSLT-процессор используется в данном примере, так как во время написания этой книги не существовало XSLT-процессора с достаточным уровнем поддержки XSLT 1.1, необходимым для запуска данного примера, для всех существующих сейчас процессоров его необходимо слегка изменять.)

Конечный документ должен выглядеть так:

```
<out>
Лук возьми свой, Гайавата,<br/>
Острых стрел возьми с собою,<br/>
Томагаук, Поггэвогон,<br/>
Рукавицы, Минджикэвон,<br/>
И березовую лодку.<br/>
Желтым жиром Мише-Намы<br/>
Смажь бока ее, чтоб легче<br/>
Было плыть ей по болотам,<br/>
И убей ты чародея,<br/>
Отомсти врагу Нокомис,<br/>
Отомсти врагу народа!<br/>
</out>
```

В данном примере у вызова функции есть побочные эффекты, потому что переменная `$reader` является внешним объектом Java, который хранит информацию о позиции в читаемом файле и увеличивает позицию при каждом чтении строки. В общем случае функции с побочными эффектами опасны, потому что XSLT не определяет порядок выполнения операторов. Но в данном случае логика таблицы стилей такова, что XSLT-процессор должен быть очень изощренным, чтобы выполнять операторы в порядке, отличном от очевидного. Тот факт, что рекурсивный вызов шаблона `read-lines` находится внутри инструкции `<xsl:if>`, проверяющей значение переменной `$line`, означает, что процессор обязан прочитать строку, проверить результат и затем, если это необходимо, выполнить рекурсивный вызов для чтения остальных строк.

В следующем примере побочные эффекты менее управляемы, и его результаты будут отличаться от процессора к процессору.

Функции с неуправляемыми побочными эффектами

Для демонстрации опасностей применения функций с побочными эффектами мы включили в книгу пример с непредсказуемым результатом.

Пример: Функция с неуправляемыми побочными эффектами

Исходный документ

Как и в предыдущем примере, таблица стилей не обращается к первичному исходному документу, поэтому нам подойдет любой XML-файл. Например, можно использовать в качестве исходного документа даже саму таблицу стилей.

В данном примере мы будем считывать исходный файл, содержащий имена и адреса, например `addresses.txt`. Мы будем считать, что файл был создан старым приложением и содержит группы из пяти строк. В каждой группе на первой строке находится номер клиента, на второй – имя клиента, на третьей и четвертой – адрес, на пятой – телефон. Мы будем считать, что последней строкой файла является строка «`****`», поскольку таким образом устроены большинство файлов старых приложений.

```
15668
Mary Cousens
15 Birch Drive
Wigan
01367-844355
17796
John Templeton
17 Spring Gardens
Wolverhampton
01666-932865
19433
Jane Arbuthnot
92 Mountain Avenue
Swansea
01775-952266
****
```

Таблица стилей

Мы могли бы захотеть написать таблицу стилей следующим образом (`addresses.xsl`), изменив предыдущий пример:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.1"
   xmlns:FileReader="java:java.io.FileReader"
```

```
    xmlns:BufferedReader="java:java.io.BufferedReader"
    xmlns:System="java:java.lang.System"
    exclude-result-prefixes="FileReader BufferedReader System">

<xsl:script language="java" implements-prefix="FileReader"
            src="java:java.io.FileReader"/>
<xsl:script language="java" implements-prefix="BufferedReader"
            src="java:java.io.BufferedReader"/>
<xsl:script language="java" implements-prefix="System"
            src="java:java.lang.System"/>

<xsl:template match="/">
    <xsl:variable name="reader"
         select="BufferedReader:new(FileReader:new($filename))"/>
    <xsl:call-template name="read-addresses">
        <xsl:with-param name="reader" select="$reader"/>
    </xsl:call-template>
</xsl:template>

<xsl:template name="read-addresses">
   <xsl:param name="reader"/>
   <xsl:variable name="line1" select="BufferedReader:readLine($reader)"/>
   <xsl:if test="$line1 != '****'">
     <xsl:variable name="line2" select="BufferedReader:readLine($reader)"/>
     <xsl:variable name="line3" select="BufferedReader:readLine($reader)"/>
     <xsl:variable name="line4" select="BufferedReader:readLine($reader)"/>
     <xsl:variable name="line5" select="BufferedReader:readLine($reader)"/>
     <label>
       <address>
         <xsl:value-of select="$line3"/><br/>
         <xsl:value-of select="$line4"/><br/>
       </address>
         <recipient>Attn: <xsl:value-of select="$line2"/></recipient>
     </label>
     <xsl:call-template name="read-lines">
         <xsl:with-param name="reader" select="$reader"/>
     </xsl:call-template>
   </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

В чем разница? На этот раз мы предполагаем, что четыре переменных, `$line2`, `$line3`, `$line4` и `$line5`, будут вычисляться в порядке, в котором мы их записали. Это не гарантируется, и существуют процессоры, для которых эта таблица стилей вернет неверный результат. Например, процессор имеет право не вычислять переменную до того, как она использована, а это означает, что переменная `$line3` может быть вычислена *раньше*, чем переменная `$line2`, или даже хуже, `$line5` (потому что она никогда не используется) может никогда не быть вычислена, то есть вместо чтения группы из пяти строк шаблон будет читать только четыре строки при каждом вызове.

Поэтому данная таблица стилей может работать с одними XSLT-процессорами и не работать с другими.

Конечный документ

Конечный документ целиком и полностью зависит от последовательности выполнения, выбранной конкретным XSLT-процессором.

Как и для предыдущего примера, я не смог найти во время написания книги XSLT-процессор, способный выполнять данную таблицу стилей.

На практике я, вероятно, подошел бы к этой проблеме с другой стороны. С процессором, основанным на TrAX, таким как Saxon или Xalan, я бы написал приложение на Java, читающее строки файла и выводящее их как набор узлов элементов, как если бы приложение было анализатором SAX2. Я бы задал класс `URIResolver`, выполняющий данное приложение, когда URI принимает заданное значение, а затем я бы вызвал из таблицы стилей функцию `document()` с данным URI в качестве аргумента для получения содержимого файла с данными как вторичного исходного документа.

С MSXML3 я бы, вероятно, принял бы подобный подход, но с использованием DOM. Я бы написал функцию расширения на JavaScript, которая читала бы файл и создавала дерево DOM, а затем возвращала его в функцию расширения в качестве результата вызова функции.

Оба этих решения безопасней, потому что они используют один вызов функции в таблице стилей для чтения всего файла, поэтому нет зависимости от порядка выполнения.

Объект XSLTContext

Разработчики языка XSLT стремились разрешить пользователям создавать библиотеки функций расширения, отражающие возможности и соглашения о вызовах из основных функций, предоставляемых XPath и XSLT, поэтому вызов функции расширения будет выглядеть так же, как и вызов встроенной функции. Многие из основных функций должны знать контекст XPath; например, если такие функции, как `name()` и `generate-id()`, вызываются без аргументов, то они работают с контекстным узлом. Поэтому было решено, что функции расширения должны иметь доступ к информации о контексте. Обеспечивается это при помощи дополнительного неявного параметра, передаваемого методу Java, если в его сигнатуре определено, что первым аргументом должен быть класс `org.w3c.xsl.XSLTContext`.

Объект `XSLTContext` предоставляет следующие методы. Ни один из них не определяет каких-либо особенных исключительных условий, впрочем, как любые методы в Java, они могут генерировать такие стандартные исключения, как `java.lang.IllegalArgumentException` и `java.lang.NullPointerException`.

```
org.w3c.dom.Node getContextNode()
```

Этот метод возвращает контекстный узел XPath, представленный в виде узла DOM `Node`: это тот же узел, который можно получить при помощи выражения XPath «`.`». Для перехода от этого узла к другим узлам дерева можно воспользоваться методами DOM. Не определяется, что произойдет в редкой ситуации, когда контекстный узел является узлом пространства имен (узлы пространства имен нельзя непосредственно представить в DOM).

```
int getContextPosition()
```

Этот метод возвращает контекстную позицию XPath, и это то же значение, которое возвращает функция XPath `position()`.

```
int getContextSize()
```

Этот метод возвращает контекстный размер XPath, и это то же значение, которое возвращает функция XPath `last()`.

```
org.w3c.dom.Node getCurrentNode()
```

Этот метод возвращает текущий узел XSLT, представленный в виде узла DOM `Node`, и это тот же узел, который возвращается функцией XPath `current()`.

```
org.w3c.dom.Document getOwnerDocument()
```

Этот метод возвращает объект DOM `Document`, который можно применить для создания новых узлов. Он не обязательно связан с каким-либо документом, используемым в таблице стилей. К этому объекту можно применить методы `createDocumentFragment`, `createElement`, `createElementNS`, `createAttribute`, `createAttributeNS`, `createTextNode`, `createComment`, `createProcessingInstruction` и `importNode`. Результат вызова других методов не определен.

```
Object systemProperty(String namespaceURI, String localName)
```

Этот метод возвращает то же, что и функция XPath `system-property()`. Вместо того чтобы передавать имя свойства как полное имя, как было бы сделано в XPath, два компонента расширенного имени передаются отдельно. Тип результата зависит от запрашиваемого свойства.

```
String stringValue(org.w3c.dom.Node node)
```

Этот метод возвращает строковое значение указанного узла, следуя правилам XPath. Это удобный метод, который доступен потому, что на удивление сложно написать эквивалентную функцию, использующую лишь интерфейсы DOM.

Пример

В приведенном ниже примере кода показан метод, используемый для реализации функции расширения, возвращающей тип контекстного узла, например `"root"`, `"element"`, `"text"` и т. д. Это можно определить и в XPath, но это не просто.

```
package name.joe_programmer.xslt;
import org.w3c.dom.*;
```

```
import org.w3c.xsl.XSLTContext;

public abstract class Extensions {

    public static String nodeType(XSLTContext context) {
        Node node = context.getContextNode();
        if (node instanceof Document) return "root";
        if (node instanceof Element) return "element";
        if (node instanceof Attr) return "attribute";
        if (node instanceof ProcessingInstruction)
                                    return "processing-instruction";
        if (node instanceof Comment) return "comment";
        if (node instanceof Text) return "text";
        return "unknown node type";
    }
}
```

Затем этот метод можно вызвать из таблицы стилей следующим образом:

```
<xsl:stylesheet. . .
    xmlns:my="http://joe-programmer.name/xslt">
<xsl:script implements-prefix="my" language="java"
            src="java:name.joe_programmer.xslt.Extensions"/>
. . .
<xsl:template name="a-template">
  <xsl:if test="my:node-type() = 'attribute'">
    <xsl:message terminate="yes">Контекстный узел является атрибутом</xsl:message>
  </xsl:if>
    . . .
</xsl:template>
```

Конечно же, каждый XSLT-процессор предоставит собственную реализацию объекта `XSLTContext`. Зачастую это предоставляет дополнительную информацию и возможности помимо определенных выше, к которым можно получить доступ, приведя переданный объект `XSLTContext` к классу реализации производителя. Например, в Saxon контекстный объект предоставляет информацию о пространстве имен из области действия выражения в таблице стилей, которую необходимо знать, если функция расширения (наподобие `key()` или `format-number()`) принимает строковый аргумент, который должен содержать полное имя. Таким образом, объект `XSLTContext` часто открывает двери к целому ряду возможностей, специфичных для продукта.

Используя приведенную ниже структуру, можно сохранить переносимые функции расширения и пользоваться при этом возможностями, специфичными для производителя:

```
public static String methodName(XSLTContext context) {
    if (context instanceof com.icl.saxon.Context) {
        com.icl.saxon.Context saxonContext = (com.icl.saxon.Context)context;
        // Код для Saxon
    } else if (context instanceof
                            org.apache.xalan.extensions.ExpressionContext){
        org.apache.xalan.extensions.ExpressionContext  xalanContext =
```

```
                (org.apache.xalan.extensions.ExpressionContext)context;
            // Код для Xalan
        } else {
            return "*** неизвестный процессор XSLT: " +
                        context.systemProperty(
                            "http://www.w3.org/1999/XSL/Transform", "vendor");
        }
    }
```

Привязки для языка JavaScript

Рабочий проект спецификации XSLT 1.1 определяет привязки для двух языков: Java и JavaScript. До сих пор мы говорили о привязках для Java, а теперь пришло время посмотреть на JavaScript. Формально в стандарте для этого языка используется название ECMAScript, охватывающее и продукт JavaScript от Netscape, и JScript от Microsoft.

JavaScript гораздо проще, чем Java, а потому не удивительно, что привязки XSLT для JavaScript также гораздо проще, чем привязки для Java. Они проще по трем основным причинам:

- JavaScript, как и XSLT, – это динамически типизированный язык, что означает, что значения имеют тип данных времени выполнения и они не ограничиваются объявлениями переменных.
- Поскольку JavaScript – это интерпретируемый язык, появляется возможность включить исходный код в сами таблицы стилей.
- В отличие от Java, где всегда существуют два уровня имен – классы и методы, в JavaScript имена функций могут иметь глобальную область действия.

Очень важно различать код на JavaScript, вызываемый при выполнении преобразований процессором XSLT, и код на JavaScript на стороне клиента, который должен составлять часть генерируемого HTML-кода. Чтобы еще больше все усложнить, скажу, что если преобразования XSLT запускаются в броузере, то само преобразование может быть запущено кодом на JavaScript из HTML-страницы. Когда вы будете отлаживать код с веб-сайта поздно ночью, эти три типа JavaScript могут легко перепутаться на экране...

JavaScript, используемый в процессе преобразования, написан либо в таблице стилей как содержимое элемента <xsl:script>, либо в отдельном файле (обычно с расширением «.js»), на который указывает атрибут src элемента <xsl:script>. Так же, конечно, можно воспользоваться элементами <xsl:include> или <xsl:import> для ссылки на другой модуль таблицы стилей, содержащей требуемый элемент <xsl:script>. Такой подход очень полезен, когда необходимо использовать одни и те же функции JavaScript во многих таблицах стилей.

Поскольку имена функций JavaScript могут иметь глобальную область действия, можно определить все необходимые функции JavaScript в одном

элементе <xsl:script> и использовать один префикс пространства имен для указания на любые из них. Это отличается от привязок для Java, когда необходимо иметь по одному элементу <xsl:script> для каждого класса, на который вы ссылаетесь.

Пример JavaScript

Показать, как работают привязки для JavaScript, проще всего посредством примера. Ниже приведена таблица стилей, использующая функции JavaScript для проверки, соответствует ли строка регулярному выражению, заданном другой строкой.

```
<xsl:stylesheet version="1.1"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:js="javascript:code">

<xsl:script language="javascript" implements-prefix="js">
   function matches(str, regexp) {
       return str.match(new RegExp(regexp));
   }
</xsl:script>

<xsl:template match="*[js:matches(name(), 'address$')]">
    . . .
</xsl:template>

</xsl:stylesheet>
```

Элемент <xsl:template> определяет шаблонное правило, которое будет соответствовать любому элементу, чье имя заканчивается на «address», например «billing-address» или «delivery-address».

> *Вызов функции расширения внутри предиката шаблона поиска, вероятно, потребует значительных затрат процессорного времени; но если она достигает желаемого эффекта, это может оказаться лучшим способом выполнения данной задачи.*

В качестве префикса пространства имен я использую значение «js», а в качестве URI пространства имен – значение «javascript:code», но такой выбор случаен. Единственное, что имеет значение, так это то, что URI пространства имен для префикса пространства имен, используемый в атрибуте implements-prefix элемента <xsl:script>, совпадает с URI пространства имен для префикса пространства имен, используемым в вызове функции.

Выбор запускаемой функции

Когда в процессе обработки XSLT встречается вызов функции, наподобие js:matches(), первым делом ищутся все элементы <xsl:script>, имеющие атрибут implements-prefix, идентифицирующий нужное пространство имен.

Затем проверяется значение атрибута language у этих элементов <xsl:script>. Процессор может проигнорировать элементы <xsl:script> для неподдерживаемых им языков и произвольно выбрать язык для элементов <xsl:script>,

предназначенных для нескольких языков. Если два элемента <xsl:script> имеют один и тот же атрибут implements-prefix, но различные атрибуты language, тогда считается, что они предоставляют альтернативные, но эквивалентные реализации одних и тех же функций расширения.

Давайте предположим, что один или более элементов <xsl:script> предоставляют реализации на JavaScript требуемого пространства имен функций расширения, и что именно эту реализацию выбрал процессор.

Локальное имя, используемое в вызове функции XPath, применяется для определения требуемой функции JavaScript. Если в ее имени встречаются дефисы, то они удаляются, и символ, следующий непосредственно за дефисом, преобразуется в верхний регистр. Поэтому, например, вызывая функцию XPath js:contains-same-nodes(), процессор будет искать функцию JavaScript под названием containsSameNodes().

Рассматриваются только те функции, которые являются свойствами глобального объекта. Эквивалента механизму в привязках для Java для вызова функций, не являющихся глобальными, не существует.

Если требуемому имени соответствует несколько функций, то выбирается та из них, которая имеет более высокий приоритет импортирования. В спецификации ничего не сказано о том, что произойдет, если существует несколько функций с одинаковым приоритетом импортирования, но, скорее всего, будет выбрана та, которая определяется последней.

В отличие от Java, тут не существует правила, определяющего количество совпадающих аргументов. Дело в том, что в JavaScript, по существу, нет необходимости объявлять все аргументы, которые может обработать функция, так как она всегда может получить доступ к переданным аргументам при помощи массива arguments.

Преобразование аргументов функций

Поскольку в определениях функций JavaScript нет необходимости объявлять типы аргументов, правила соответствия очень просты и основываются лишь на типах значений XPath, переданных вызовам функций.

Различные типы данных XPath преобразуются в типы данных JavaScript следующим образом:

Тип данных XPath	Тип данных JavaScript
строковый	string
числовой	number
логический	Boolean
набор узлов	DOM NodeList
внешний объект	JavaScript object

Если в качестве аргумента передается внешний объект, он должен оборачивать объект JavaScript, а этот обернутый объект JavaScript и будет объектом, переданным функции JavaScript.

Эти правила означают, что передаваемые функциям JavaScript аргументы не будут автоматически преобразовываться в ожидаемые типы так, как это было для определенных в системе функций. Например, если функция ожидает строку, тогда нельзя передать значение, являющееся набором узлов; вместо вызова «`my:func(@value)`» вызовите «`my:func(string(@value))`».

Получение контекстной информации XSLT и XPath

Специального механизма для передачи контекстной информации в качестве дополнительного аргумента функции, как это было в привязках для Java, тут не существует. Вместо этого в качестве свойства глобального объекта доступен объект `XSLTContext`.

Объект `XSLTContext` имеет следующие свойства:

Имя	Тип	Описание
`contextNode`	`Node`	Контекстный узел XPath (узел, возвращаемый выражением «`.`»).
`contextPosition`	`число`	Контекстная позиция XPath (значение функции `position()`).
`contextSize`	`число`	Контекстный размер XPath (значение функции `last()`).
`currentNode`	`Node`	Текущий узел XSLT (узел, возвращаемый функцией `current()`).
`ownerDocument`	`Document`	Объект DOM `Document`, который можно использовать для создания новых узлов. Он не обязательно связан с каким-либо документом, используемым в таблице стилей. К этому объекту можно применить методы `createDocumentFragment`, `createElement`, `createElementNS`, `createAttribute`, `createAttributeNS`, `createTextNode`, `createComment`, `createProcessingInstruction` и `importNode`. Результат вызова других методов не определен.

Методы объекта `XSLTContext` перечислены ниже:

Имя	Аргументы	Результат	Описание
`systemProperty`	`String namespaceURI` `String localName`	`Object`	Эквивалент функции XSLT `system-property()`
`stringValue`	`Node node`	`строка`	Возвращает строковое значение для заданного узла в соответствии с правилами XPath

Преобразование возвращаемого значения

Правила преобразования возвращаемых значений JavaScript в значения XPath гораздо сложнее, чем соответствия для аргументов из-за сложностей с обработкой возвращаемых узлов DOM, о чем говорилось на стр. 634.

Вот эти правила:

Тип данных JavaScript	Тип данных XPath
`string`	строковый
`number`	числовой
`boolean`	логический
`DocumentFragment`, `Document`, `Node`, `NodeList`, `Attr`, `Element`, `Text`, `Comment`, `CDATASection`, `EntityReference`, `ProcessingInstruction`	набор узлов
`null`	внешний объект
не определенный	пустой набор узлов
любой другой объект	внешний объект

Если возвращается строка, то она не должна содержать недопустимых в XML символов, таких как ASCII-символ `NUL` (`#x00`); однако XSLT-процессор не обязан сообщать об этой ошибке из-за возможных ограничений производительности. То же самое касается и строк, содержащихся в возвращаемых узлах DOM.

Не существует простого способа проверить из XPath, возвращает ли функция значение null. Нужно предоставить другую внешнюю функцию, вызываемую для этой проверки.

Функция function-available()

До тех пор пока процессор поддерживает вызов внешних функций JavaScript, функция `function-available()` будет возвращать значение `истина`, если после запуска всех сценариев из элементов `<xsl:script>` глобальный объект для пространства имен, используемого в имени функции, будет иметь функцию, имя которой соответствует локальной части имени функции.

Резюме

Внешние функции полезны для расширения возможностей таблиц стилей XSLT. Они позволяют таблицам стилей обращаться к внешним системным возможностям и выполнять вычисления, которые сложно или неэффективно выполнять в «чистом» XSLT и XPath.

Привязки для языков, описанные в этой главе, приведены в том виде, в котором они описаны в Рабочем проекте спецификации XSLT 1.1 Recommendation, определяющем стандартные привязки к языкам для вызова внеш-

них функций, написанных на Java или JavaScript. Когда писалась эта книга, ни один процессор XSLT не обеспечивал выполнения этой спецификации. Однако многие из них предлагают нечто подобное, так что если вы поняли представленную здесь спецификацию, у вас не должно возникнуть сложностей с различными вариантами, реализованными в различных продуктах XSLT 1.0. Даже когда выйдет XSLT 1.1, производители могут предоставить другие привязки для языков либо альтернативные привязки для этих двух языков. В частности, многие производители, вероятно, будут продолжать поддерживать собственные механизмы для привязок, доступные в их продуктах с версии 1.0, которые разрабатывались до того, как были определены стандартные механизмы привязок. Подробности о некоторых из этих механизмов представлены в приложениях с информацией о продуктах. Однако когда производители перейдут на стандартные механизмы, описанные в этой главе, было бы неплохо перейти к этим интерфейсам, поскольку это сделает ваши функции расширения переносимыми между процессорами XSLT, если они, конечно, поддерживают выбранный язык.

В следующей главе мы отойдем от подробных спецификаций интерфейсов и посмотрим на то, как можно использовать возможности XSLT для создания хорошо спроектированных таблиц стилей.

9

Образцы проектирования таблиц стилей

В этой главе рассмотрены четыре часто встречающихся **образца проектирования** (design patterns) для таблиц стилей XSLT.

Концепция образцов проектирования введена Эрихом Гаммой (Erich Gamma), Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидесом (John Vlissides) в их классической книге «Design Patterns: Elements of Reusable Object-Oriented Software», Addison Wesley, ISBN 0201633612 («Приемы объектно-ориентированного проектирования: Паттерны проектирования», «Питер», ISBN 5-272-00355-1). Их идея состояла в том, что некоторый набор методик оказывается полезным неоднократно. Они представили 23 различных образца проектирования для объектно-ориентированного программирования, не претендуя на то, что это полный перечень возможных моделей, но утверждая, что в огромном большинстве программ, написанных опытными проектировщиками, задействована одна или несколько этих моделей.

Что касается таблиц стилей XSLT, большинство из них основано на одном из четырех образцов проектирования:

- Таблице стилей для заполнения бланков
- Навигационных таблицах стилей
- Таблицах стилей, основанных на правилах
- Вычислительных таблицах стилей

Опять же, это не означает, что это единственные способы написания таблиц стилей, но также это не означает, что каждая создаваемая таблица стилей должна соответствовать только одному из этих четырех образцов, исключая три других. Просто очень многие таблицы стилей, написанные опытными программистами, используют один из этих образцов, и поэтому знакомство с ними обеспечивает хороший набор методов, которые можно применять при решении любых задач.

Первые три образца проектирования будут описаны довольно кратко, потому что они не очень трудны. Четвертый образец проектирования для вычислений будет рассмотрен более подробно – не потому, что к нему придется обращаться чаще, а потому что он требует иного подхода к алгоритмам, чем тот, который принят в традиционных процедурных языках программирования.

Таблица стилей для заполнения бланков

Многие частные языки шаблонов были созданы для HTML. Шаблоны выглядят в значительной степени подобно стандартному HTML-файлу, но с добавлением дополнительных тегов, используемых для получения переменных данных и вставки их в определенное место HTML-страницы. Проектировщики XSLT стремились гарантировать, что несмотря на мощь XSLT как полноценного языка преобразований, таким простым способом его смогут использовать и люди, далекие от программирования, но имеющие некоторый опыт создания HTML-страниц.

Ниже приведен пример такой таблицы стилей. Она использует упрощенный синтаксис **таблиц стилей** (или синтаксис *конечного литерального элемента как таблицы стилей*, – так неуклюже это названо в стандарте), поэтому элементы `<xsl:stylesheet>` и `<xsl:template match="/">` подразумеваются.

Пример: Таблица стилей «Заполнение бланков»

Исходный документ

Этот XML-документ, `orgchart.xml`, представляет схему структуры организации, показывая главное руководство некоторой компании в определенный день. Документ организован как рекурсивная структура, которая прямо отражает управленческую иерархию:

```
<orgchart дата="28 марта 2001">
<сотрудник>
<имя>Тим Бернерс-Ли</имя>
<должность>Исполнительный директор</должность>
<отчет>
   <сотрудник>
      <имя>Шэрон Адлер</имя>
      <должность>Технический директор</должность>
      <отчет>
         <сотрудник>
             <имя>Тим Брей</имя>
             <должность>Главный инженер</должность>
         </сотрудник>
         <сотрудник>
             <имя>Джеймс Кларк</имя>
             <должность>Начальник исследовательского отдела</должность>
         </сотрудник>
```

```
      </отчет>
   </сотрудник>
   <сотрудник>
      <имя>Генри Томпсон</имя>
      <должность>Финансовый директор</должность>
   </сотрудник>
   <сотрудник>
      <имя>Давид Меггинсон</имя>
      <должность>Начальник отдела кадров</должность>
   </сотрудник>
   <сотрудник>
      <имя>Стив Мюнх</имя>
      <должность>Начальник торгового отдела</должность>
   </сотрудник>
   <сотрудник>
      <имя>Скотт Боаг</имя>
      <должность>Начальник международного отдела</должность>
   </сотрудник>
</отчет>
</сотрудник>
</orgchart>
```

Таблица стилей

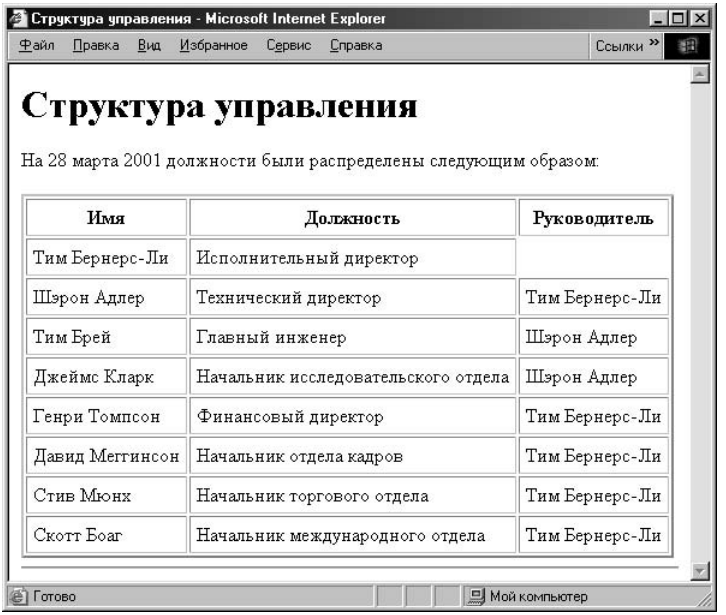
Можно придумать много способов для отображения этих данных. Например, можно воспользоваться SVG-графикой, деревьями в стиле проводника Windows, реализованными с помощью JavaScript на стороне клиента, или просто списками с отступом. Целью книги не является обучение читателей приемам HTML, поэтому эта таблица стилей (`orgchart.xsl`) будет отображать данные просто в виде довольно скучной таблицы с одной строкой для каждого должностного лица и тремя столбцами: для имени данного лица, его должности, а также имени его руководителя:

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xsl:version="1.0">
<head>
   <title>Структура управления</title>
</head>
<body>
   <h1>Структура управления</h1>
   <p>На <xsl:value-of select="/orgchart/@дата"/>
      должности были распределены следующим образом:
   </p>
   <table border="2" cellpadding="5">
   <tr>
<th>Имя</th><th>Должность</th><th>Руководитель</th>
   </tr>
   <xsl:for-each select="//сотрудник">
      <tr>
         <td><xsl:value-of select="имя"/></td>
         <td><xsl:value-of select="должность"/></td>
```

```
            <td><xsl:value-of select="ancestor::сотрудник[1]/имя"/></td>
          </tr>
        </xsl:for-each>
        </table>
        <hr/>
    </body>
    </html>
```

Идея этого образца проектирования заключается в том, что таблица стилей имеет такую же структуру, как требуемый вывод. Неизменное содержимое включено прямо в таблицу стилей в виде текста или конечных литеральных элементов, в то время как переменные данные включаются с помощью инструкций <xsl:value-of>, которые извлекают требуемые данные из исходного документа. Повторяющиеся разделы вывода, обычно строки таблицы или пункты списка, могут быть включены в инструкцию <xsl:for-each>, а условные разделы – в инструкцию <xsl:if> или <xsl:choose>.

Вывод



Структура управления

На 28 марта 2001 должности были распределены следующим образом:

Имя	Должность	Руководитель
Тим Бернерс-Ли	Исполнительный директор	
Шэрон Адлер	Технический директор	Тим Бернерс-Ли
Тим Брей	Главный инженер	Шэрон Адлер
Джеймс Кларк	Начальник исследовательского отдела	Шэрон Адлер
Генри Томпсон	Финансовый директор	Тим Бернерс-Ли
Давид Меггинсон	Начальник отдела кадров	Тим Бернерс-Ли
Стив Мюнх	Начальник торгового отдела	Тим Бернерс-Ли
Скотт Боаг	Начальник международного отдела	Тим Бернерс-Ли

Рис. 9.1. *Вывод первого примера применения образца проектирования таблицы стилей*

Таблица стилей такого типа весьма ограниченно использует мощь XSLT, зато она очень похожа на многие специализированные языки шаблонов, используемые в настоящее время. Опыт показывает, что такие таблицы стилей достаточно просты для приобретения навыков XSLT опытными авторами HTML, даже если они далеки от программирования. Это – важное соображение, так как на многих крупных веб-сайтах постоянно приходится в краткие сроки

вводить новые шаблоны страниц, а с этим гораздо проще справиться, если авторы и редакторы содержимого смогут делать это самостоятельно.

Здесь, конечно, есть ограничение: на входе должен быть XML-документ. Это отличается от большинства патентованных языков, где ввод часто берется прямо из реляционной базы данных. Наиболее изящным способом преодоления этого ограничения является обеспечение преобразования требуемых данных в древовидный формат, понятный используемому XSLT-процессору; в этом случае отпадает потребность в фактическом наличии XML-документа как промежуточного формата. Многие XSLT-процессоры принимают ввод или в форме дерева DOM, или в форме потока событий SAX, поэтому можно написать модуль интерфейса, который будет делать запросы к базе данных, а результат выдавать в древовидной форме для использования авторами таблиц стилей. Другой подход состоит в том, чтобы использовать функцию `document()` (описанную в главе 7) с URI для обращения к сервлету с параметрами для получения требуемых данных. Как можно ожидать, Oracle с их технологиями XSQL-страниц и XSQL-сервлетов, вероятно, достигли большего в области XML/XSLT/SQL-интеграции, чем другие поставщики: воспользуйтесь ссылками, приведенными в приложении B.

Навигационные таблицы стилей

Навигационные таблицы стилей являются естественным развитием простых таблиц стилей для заполнения бланков.

Подобно таблицам стилей для заполнения бланков, навигационная таблица стилей также, по существу, ориентирована на вывод. Однако в ней, вероятно, уже стоит использовать именованные шаблоны в качестве подпрограмм для выполнения типичных задач, переменные – для вычисления значений, необходимых в нескольких местах, а также ключи, параметры и сортировку.

В то время как таблица стилей для заполнения бланков напоминает HTML, сдобренный несколькими дополнительными управляющими инструкциями, навигационная таблица стилей (если не смотреть на угловые скобки) выглядит очень похоже на обычную процедурную программу с переменными и условными операторами, циклами и вызовами подпрограмм.

Навигационные таблицы стилей часто используются для генерирования отчетов из XML-документов, ориентированных на данные, структура которых регулярна и предсказуема.

Пример: Навигационная таблица стилей

Исходный документ

Предположим, что исходный документ, `книги.xml`, выглядит так:

```
<книги>
   <книга>
```

```
      <название>Прах Анжелы</название>
      <автор>Фрэнк МакКорт</автор>
      <издательство>HarperCollins</издательство>
      <isbn>0 00 649840 X</isbn>
      <цена>6.99</цена>
      <продажи>235</продажи>
   </книга>
   <книга>
      <название>Меч чести</название>
      <автор>Ивлин Во</автор>
      <издательство>Penguin Books</издательство>
      <isbn>0 14 018967 X</isbn>
      <цена>12.99</цена>
      <продажи>12</продажи>
   </книга>
</книги>
```

Таблица стилей

Следующая навигационная таблица стилей (продажи.xsl) генерирует отчет об общем количестве продаж книг каждого издателя.

Обратите внимание на глобальную переменную «$издатели». Это – набор узлов, содержащий по одному элементу <издатель> для каждого отдельного издателя, упомянутого в исходном файле. Здесь выбираются только те издатели, которые еще не встречались ни разу, другими словами, отфильтровываются дубликаты.

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

<xsl:key name="изд" match="книга" use="издатель"/>

<xsl:variable name="издатели"
   select="//издатель[not(.=preceding::издатель)]"/>

<xsl:template match="/">
<html>
<head>
   <title>Объемы продаж по издателям</title>
</head>
<body>
   <h1>Объемы продаж по издателям</h1>
   <table>
      <tr>
      <th>Издатель</th><th>Общий объем продаж</th>
      </tr>
   <xsl:for-each select="$издатели">
      <tr>
      <td><xsl:value-of select="."/></td>
      <td><xsl:call-template name="сумма-продаж"/></td>
</tr>
```

```
      </xsl:for-each>
      </table>
   </body>
   </html>
   </xsl:template>

   <!-- вычисляем общее количество продаж книг для текущего издателя -->
   <xsl:template name="сумма-продаж">
      <xsl:value-of select="sum(key('изд',string(.))/продажи)"/>
   </xsl:template>
   </xsl:stylesheet>
```

Эта таблица стилей недалеко ушла от предыдущего примера с заполнением бланков. Но поскольку в ней используются некоторые элементы верхнего уровня, например `<xsl:key>` и именованный шаблон, здесь требуется полный синтаксис с элементом `<xsl:stylesheet>`.

Вывод

```
<html>
   <head>
      <META http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Объемы продаж по издателям</title>
   </head>
   <body>
      <h1>Объемы продаж по издателям</h1>
      <table>
         <tr>
            <th>Издатель</th>
            <th>Общий объем продаж</th>
         </tr>
         <tr>
            <td>HarperCollins</td>
            <td>235</td>
         </tr>
         <tr>
            <td>Penguin Books</td>
            <td>12</td>
         </tr>
      </table>
   </body>
</html>
```

Очевидная разница между таблицей стилей для заполнения бланков и этой навигационной таблицей стилей в том, что элементы `<xsl:stylesheet>` и `<xsl:template>` теперь заданы явно, что позволяет вводить другие элементы верхнего уровня, например `<xsl:key>` и глобальную переменную `<xsl:variable>`. Еще один нюанс в том, что, используя особенности XSLT, эта таблица стилей перешла за границы документа HTML с добавлением инструкций управления и является реальной программой. Эта граница, тем не менее, довольно размытая и не требующая визы при ее пересечении, поэтому многие из

тех, кто умеет писать простые таблицы стилей для заполнения бланков, по мере приобретения опыта вполне справятся с созданием подобных навигационных таблиц стилей.

Хотя использование инструкций управления ходом программы, таких как <xsl:if>, <xsl:call-template> и <xsl:for-each>, придает этой таблице стилей процедурный характер, фактически это не нарушает основную концепцию, согласно которой XSLT является декларативным языком. Дело в том, что инструкции не обязаны выполняться в том порядке, в каком они написаны: переменные не могут быть модифицированы, поэтому результат одной инструкции не может воздействовать на другие инструкции. Например, удобно думать, что инструкция <xsl:for-each> в этом примере обрабатывает выбранные узлы в порядке появления в документе и добавляет их один за другим в конечное дерево; но XSLT-процессор с тем же успехом мог бы обрабатывать их в обратном порядке или параллельно, при условии, что узлы добавляются в нужное место в конечном дереве. Именно поэтому этот образец проектирования назван здесь *навигационным*, а не *процедурным*. Он является навигационным, поскольку в нем точно задается, где в исходном дереве нужно искать требуемые узлы, но он не является процедурным, потому что порядок обработки узлов не определяется.

Таблицы стилей, основанные на правилах

Таблица стилей, основанная на правилах, – это таблица стилей, которая состоит, главным образом, из правил, описывающих, как должны обрабатываться различные особенности исходного документа, например: «если встретится элемент <вид>, следует отобразить его курсивом».

Некоторые считают этот, основанный на правилах подход сущностью языка XSLT, основным способом его использования. Однако правильнее рассматривать это как один из способов написания таблиц стилей, часто лучший способ, но не единственный и не обязательно лучший в каждой ситуации.

В отличие от навигационных таблиц стилей, таблица стилей, основанная на правилах, не структурирована в соответствии с желательным размещением в выводе. Фактически по ней мало что можно сказать о структуре как исходного, так и конечного документа. Скорее ее структура читается как опись компонентов, которые можно встретить в исходном документе, размещенных в произвольном порядке.

По этой причине таблицы стилей, основанные на правилах, наиболее полезны при обработке исходных документов, структура которых гибка или непредсказуема или которые могут быть значительно изменены в будущем. Они очень полезны для случаев, когда во многих различных структурах документа встречается один и тот же набор элементов, тогда правила, подобные правилу «отображать даты в формате *23 марта 2000 г.*», могут многократно использоваться во многих разных контекстах.

Таблицы стилей, основанные на правилах, – естественное развитие таблиц стилей CSS и CSS2. В CSS можно определять, например, такие правила: «для данного набора элементов применять такое отображение». В XSLT правила становятся гораздо более гибкими в двух аспектах: язык образцов, определяющий, о каких элементах идет речь, гораздо богаче, а выбор действий, которые можно задать для выполнения, когда правило активизируется, значительно шире.

Простая таблица стилей, основанная на правилах, содержит отдельное правило для каждого типа элементов. Типичное правило сопоставляется со специфическим типом элемента, выводит HTML-тег, определяющий отображение этого элемента, и вызывает `<xsl:apply-templates>` для обработки дочерних узлов шаблона. При этом текстовые узлы элемента копируются в вывод, а вложенные дочерние элементы обрабатываются с помощью соответствующего каждому из них шаблонного правила.

Пример: Таблица стилей, основанная на правилах

Исходный документ

Исходный документ `scene2.xml` – сцена из трагедии Шекспира «Отелло»; акт I, сцена 2. Начало такое:

```
<?xml version="1.0" encoding="utf-8" ?>
<SCENE>
   <TITLE>СЦЕНА II.  Другая улица.</TITLE>
   <STAGEDIR>Там же. Другая улица.
         Входят Отелло, Яго и слуги с факелами.</STAGEDIR>
   <SPEECH>
      <SPEAKER>ЯГО</SPEAKER>
      <LINE>Хоть на войне я убивал людей,</LINE>
      <LINE>Убийство в мирной жизни - преступленье.</LINE>
      <LINE>Так я смотрю. Мне было б легче жить</LINE>
      <LINE>Без этой щепетильности. Раз десять</LINE>
      <LINE>Хотелось мне пырнуть его в живот.</LINE>
   </SPEECH>
   <SPEECH>
      <SPEAKER>ОТЕЛЛО</SPEAKER>
      <LINE>И лучше, что не тронул.</LINE>
   </SPEECH>
</SCENE>
```

В этом фрагменте не отражены некоторые сложности, которые нужно учесть в таблице стилей:

- Элемент верхнего уровня – это не всегда `<SCENE>`; это может быть также `<PROLOGUE>` или `<EPILOGUE>`. Элемент `<STAGEDIR>` может встречаться на любом уровне вложенности, в частности, сценические комментарии могут появляться между двумя диалогами, между двумя строками одного диалога или в середине строки.

- Несколько действующих лиц могут говорить одновременно. В этом случае один элемент <SPEECH> будет содержать несколько элементов <SPEAKER>. Вообще, элемент <SPEECH> состоит из одного или нескольких элементов <SPEAKER>, сопровождаемых любым количеством элементов <LINE> и <STAGEDIR> в любом порядке.

Таблица стилей

Таблица стилей scene.xsl состоит из ряда шаблонных правил. Она начинается с объявления глобальной переменной (используемой просто как константа) и правила для элемента документа:

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0"
>

<xsl:variable name="цвет-фона" select="'#FFFFFF'" />

<xsl:template match="SCENE|PROLOGUE|EPILOGUE">
   <HTML>
   <HEAD>
      <TITLE><xsl:value-of select="TITLE"/></TITLE>
   </HEAD>
   <BODY BGCOLOR='{$цвет-фона}'>
      <xsl:apply-templates/>
   </BODY>
   </HTML>
</xsl:template>
```

Наличие <xsl:value-of>, не характерное для образцов проектирования, основанных на правилах, только подтверждает, что ни один из образцов проектирования не исключает использование других.

Шаблонное правило для элемента <SPEECH> выводит таблицу, содержащую одну строку и два столбца: в первом столбце перечислены действующие лица, а во втором – строки их диалогов и любые сценические ремарки:

```
<xsl:template match="SPEECH">
   <TABLE><TR>
   <TD WIDTH="160" VALIGN="TOP">
<xsl:apply-templates select="SPEAKER"/>
   </TD>
   <TD VALIGN="TOP">
   <xsl:apply-templates select="STAGEDIR|LINE"/>
   </TD>
   </TR></TABLE>
</xsl:template>
```

Остальные шаблонные правила достаточно просты. Каждое из них просто выводит текст элемента, используя соответствующее HTML-представление. Единственная сложность в том, что для некоторых элементов (<STA-

GEDIR> и <SUBHEAD>, которые фактически не встречаются в этой конкретной сцене) HTML-представление может быть разным, в зависимости от контекста элемента, поэтому для этих элементов определено более одного правила.

```
<xsl:template match="TITLE">
   <H1><CENTER>
   <xsl:apply-templates/>
   </CENTER></H1><HR/>
</xsl:template>

<xsl:template match="SPEAKER">
   <B>
      <xsl:apply-templates/>
      <xsl:if test="not(position()=last())"><BR/></xsl:if>
   </B>
</xsl:template>

<xsl:template match="SCENE/STAGEDIR">
    <CENTER><H3>
    <xsl:apply-templates/>
    </H3></CENTER>
</xsl:template>

<xsl:template match="SPEECH/STAGEDIR">
   <P><I>
   <xsl:apply-templates/>
   </I></P>
</xsl:template>

<xsl:template match="LINE/STAGEDIR">
   [ <I>
   <xsl:apply-templates/>
   </I> ]
</xsl:template>

<xsl:template match="SCENE/SUBHEAD">
   <CENTER><H3>
   <xsl:apply-templates/>
   </H3></CENTER>
</xsl:template>

<xsl:template match="SPEECH/SUBHEAD">
   <P><B>
   <xsl:apply-templates/>
   </B></P>
</xsl:template>

<xsl:template match="LINE">
   <xsl:apply-templates/>
   <BR/>
</xsl:template>

</xsl:stylesheet>
```

Вывод

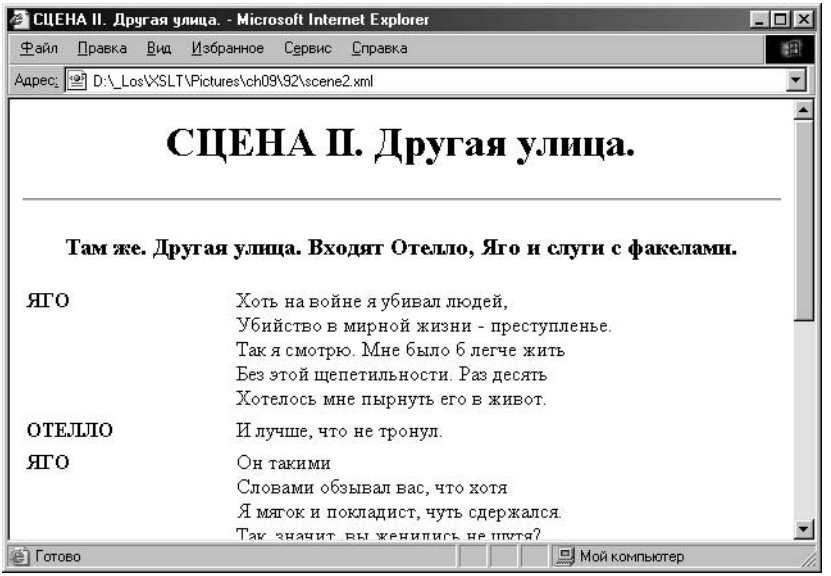


Рис. 9.2. *Полученный вывод в окне броузера*

Чаще всего таблица стилей, основанная на правилах, создает конечное дерево, которое имеет структуру, близкую к структуре исходного дерева, когда текст в основном воспроизводится в конечном документе в том же порядке, но часто с другими тегами. Чем ближе это описывает желательное для вас преобразование, тем ближе к данному примеру будет ваша таблица стилей. Однако это не означает, что обработка должна выполняться лишь последовательно. Можно обрабатывать фрагменты дерева по несколько раз, используя разные режимы, можно переупорядочить узлы дерева, а также можно использовать данные из родительских узлов, не отклоняясь при этом от образца проектирования, основанного на правилах.

Характерной особенностью таблицы стилей, основанной на правилах, является то, что она обычно содержит по одному шаблонному правилу для каждого типа объекта, встречающегося в исходном документе. Конечно, можно комбинировать образцы проектирования, особенно в тех случаях, когда исходный документ содержит смесь структур «ориентированных на данные» и «ориентированных на текст» (примером может служить форма заявления о приеме на работу). Тогда вполне оправданно использование для регулярных структур навигационных образцов проектирования, а для менее регулярных – образцов, основанных на правилах. Чем больше и сложнее ваша таблица стилей, тем более вероятно, что в ней будут содержаться примеры всех образцов проектирования.

Вычислительные таблицы стилей

Вычислительные таблицы стилей – наиболее сложные из обсуждаемых четырех образцов проектирования. Они требуются, когда в конечном дереве нужно генерировать узлы, которые не соответствуют непосредственно узлам исходного дерева. Чаще всего это бывает при отсутствии в исходном документе точной разметки. Например:

- Текстовое поле в источнике состоит из разделенных запятыми пунктов списка, которые должны быть отображены в выводе как маркированный список.
- В выводе может понадобиться генерировать элементы <раздел>, хотя в источнике разделы явно не отражены, но определены как состоящие из элемента <h1> и всех последующих одноуровневых элементов, вплоть до следующего элемента <h1>.

Другими примерами могут служить ситуации, когда требуется организовать данные в строки и столбцы, а в исходном документе они структурированы по-другому, или когда нужно произвести сложную агрегацию фрагментов данных, например, для построения сводной таблицы игр футбольной лиги из исходных данных, содержащих результаты индивидуальных матчей.

При создании вычислительных таблиц стилей постоянно возникает проблема, связанная с тем, что в XSLT нет выражений присваивания, и поэтому невозможно писать циклы привычным в других языках способом. Для решения таких задач требуется понимание некоторых концепций **функционального программирования**, которые обсуждаются в следующем разделе.

Программирование без выражений присваивания

В 1968 году известный ученый-компьютерщик Еджер Дейкстра (Edsger Dijkstra) опубликовал статью под названием «Инструкция GoTo вредна» («GoTo Statement Considered Harmful»). Его идея, что программы должны писаться без инструкций goto, разрушили общепринятые взгляды программистов. В то время они использовали ранние диалекты языков Fortran и Cobol, в которых подавляющее большинство решений в программе осуществлялось с использованием конструкции, которая прямо соответствовала инструкции условного перехода в аппаратных средствах: «if *условие* goto *метка*». Даже способ записи решений проектирования того времени – повсеместные блок-схемы, рисуемые карандашом с помощью пластмассовых лекал, – представлял управляющую логику программ таким образом.

Дейкстра утверждал, что в структурированных программах, написанных с использованием конструкций if-then-else и while-do вместо goto, гораздо меньше вероятность ошибок, и они легче читаются и поэтому их проще сопровождать. В то время эти идеи вызвали отчаянное противодействие, особенно со стороны программистов-практиков, и долгое время впоследствии противники этой идеи встречали энтузиастов структурного программирования язвительными фразами типа: «Что ж, а как же вы сделаете *это* без инструкции goto?».

Сегодня, однако, сражение выиграно, и инструкция `goto` ушла в историю. Современные языки, подобные Java, не используют инструкцию `goto`, и никто по ней не скучает.

Однако до сих пор есть также другая группа энтузиастов, которые утверждают, что вредны выражения присваивания. В отличие от Дейкстры, этим проповедникам еще нужно убедить скептиков в своей правоте, хотя всегда существовал определенный круг сторонников этого подхода.

Стиль программирования без выражений присваивания называется *функциональным программированием*. Самым первым и наиболее известным функциональным языком программирования был Lisp (иногда шутливо расшифровываемый как «избыточное множество раздражающих круглых скобок» – «Lots of Irritating Superfluous Parentheses»), а более современными примерами являются ML и Scheme. (См., например, книгу Брайана Харви и Мэтью Райта (Brian Harvey and Matthew Wright) «Simply Scheme: Introducing Computer Science», MIT Press, 1999, ISBN 0-262082-81-0.)

XSLT – язык без выражений присваивания, и хотя его синтаксис сильно отличается от этих языков, его философия основана на концепциях функционального программирования. Фактически его нельзя назвать полноценным функциональным языком программирования, потому что он не позволяет манипулировать функциями таким же образом, как данными; но во многих других отношениях он вписывается в эту категорию языков. Если приходится делать в XSLT что-то очень сложное, стоит обучиться программированию без выражений присваивания. Поначалу это, вероятно, будет непросто, потому что как для прежних программистов на языках Fortran и Cobol решением любой проблемы неизменно служила инструкция `goto`, сейчас для тех, кто имеет опыт программирования на языках типа C или Visual Basic, или даже Java, любимым универсальным инструментом являются выражения присваивания.

Что же не так в выражениях присваивания, и почему их нет в XSLT?

Основная загвоздка заключается в том, что именно выражения присваивания требуют определенного порядка выполнения программы. Без выражений присваивания порядок выполнения не играет роли, потому что результат одной инструкции в этом случае не зависит от состояния, в которое привела систему предыдущая инструкция. Так же как инструкция `goto` равносильна команде «jump» в аппаратных средствах, так и выражение присваивания равносильно команде «store», а причина сохранения выражений присваивания в современных языках программирования заключается в том, что они были предназначены для последовательного программирования на вычислительных машинах с фон-неймановской архитектурой, использующих команды перехода и сохранения. Для того чтобы избавиться от последовательного мышления, привязанного к последовательной аппаратной архитектуре, нужно найти способ описания требуемого эффекта, а не последовательности команд, которые должна выполнить машина для его достижения.

Задача функциональной программы – описать вывод как функцию ввода. XSLT – язык преобразований; он предназначен для преобразования входного документа в выходной документ. Таким образом, таблицу стилей можно расценивать как функцию, которая определяет это преобразование: таблица стилей – это функция $O=S(I)$, где I – входной документ, S – таблица стилей и O – выходной документ. Вспомните слова Джеймса Кларка на парижском симпозиуме в 1995 году, которые цитировались в главе 1:

> *Таблица стилей DSSSL очень точно описывает функцию, отображающую SGML в дерево потоковых объектов.*

По мере развития XSLT эта концепция явно остается ключевым аспектом философии языка. (И действительно, *потоковые объекты* DSSSL в конечном счете стали форматирующими объектами XSL.)

Слово **«функция»** используется здесь в его математическом смысле. В языках, подобных Fortran и Visual Basic, этим словом обозначается подпрограмма, которая выдает результат, но в математическом смысле функция – это не алгоритм или последовательность шагов, которые нужно выполнить, а выражение зависимости. Функция извлечения квадратного корня выражает зависимость между 3 и 9, а именно: `3=sqrt(9)`. Сущность функции в том, что она является установленной, постоянной, достоверной зависимостью, и ее вычисление не изменяет систему. На вопрос: «Чему будет равен квадратный корень из 9, если вычислить функцию?» – можно спокойно ответить: «Тому же, чему он равен, если этого не делать». В этом случае можно так сказать, потому что квадратный корень – **чистая** функция (**pure** function), она дает одинаковый результат, кто бы и как часто ее ни вызывал, и вызов ее однажды не изменяет результата, который она даст при следующем вызове; фактически, она не изменяет ничего.

Замечательным свойством чистых функций является то, что их можно вызывать любое количество раз, в любом порядке и получать всегда один и тот же результат. Если нужно вычислить квадратный корень всех целых чисел от нуля до тысячи, не имеет значения, начинать ли с нуля и вычислять по возрастающей или начинать с тысячи и двигаться вниз. Точно так же при вычислении на одном компьютере или одновременно на тысяче компьютеров результат будет тот же самый. Чистые функции не имеют никаких побочных эффектов.

С выражениями присваивания дело обстоит иначе. В этом случае **играет** роль, вычисляются они или нет. Результат выражения «`x = x+1;`» очень зависит от того, как часто оно выполняется. В случае нескольких выражений присваивания:

```
temp = x;
x = y;
y = temp;
```

их эффект зависит от последовательности их выполнения.

Это означает, конечно, что чистая функция не может модифицировать внешние переменные. Однако как только разрешаются присваивания, все становится зависимым от последовательности выполнения команд: каждый раз должна выполняться только одна команда и только в правильном порядке.

Может быть, по этой причине в объектно-ориентированных языках избегают ситуаций, при которых один объект модифицирует данные, содержащиеся в другом объекте? Нет, потому что, хотя там избегают прямой записи в защищенные данные, но допускают это через использование методов `get()` и `set()`. Модификация переменной, достигаемая косвенно через определенный интерфейс, приводит к той же самой зависимости от последовательности операций, как и прямая модификация с помощью выражений присваивания. Чистая функция не должна иметь никаких побочных эффектов; результат, который она выдает, должен быть ее единственным выводом.

Основная причина, по которой функциональные языки считаются идеальными для языка таблиц стилей (или языка преобразований деревьев, если так лучше), – не столько в возможности выполнять операции параллельно или в любом порядке, а скорее в возможности делать это по частям (инкрементно). Пусть нужно уйти от статических страниц: если вы отображаете карту автомобильных пробок в вашем районе, то при изменении данных для конкретного перекрестка нужно обновлять картину в реальном времени, причем изменения должны отображаться без необходимости повторного вычисления и перерисовки полностью всех данных. Это возможно, если только имеется прямая зависимость – функция – между тем, что отображается на отдельном участке карты, и конкретным элементом данных в используемой базе данных. Так, если разбить функцию таблицы стилей верхнего уровня, $O=S(I)$, на набор меньших, независимых функций, каждая из которых определяет взаимосвязь между одной частью вывода и соответствующей частью ввода, тогда появится возможность осуществлять обновление на лету.

Еще одним преимуществом такого подхода является то, что, когда из сети загружается большая XML-страница, броузер может начать отображать вывод по частям, по мере доступности соответствующих частей ввода, – теоретически, во всяком случае. В действительности, пока нет XSLT-процессоров, которые могут делать это, потому что это требует довольно сложного анализа таблицы стилей, чтобы решить, когда и в какой степени это возможно в каждом конкретном случае, – но если бы таблица стилей была обычной программой с побочными эффектами, это не было бы возможно вообще, потому что последняя загруженная часть ввода могла бы изменить всю предшествующую картину.

Именно здесь полезны шаблонные правила XSLT. Они действуют как небольшие, независимые функции, связывающие одну часть вывода с соответствующей частью ввода. Шаблонное правило не имеет никаких побочных эффектов, его вывод – чистая функция его ввода. Ввод представляет собой (если слегка идеализировать) текущую позицию входного документа с добавлением всех заданных параметров. Не имеет значения, в каком порядке выполняются шаблонные правила, если отдельные фрагменты вывода пра-

вильным способом объединяются при формировании конечного дерева. Если часть ввода изменяется, то потребуется заново вычислить только те шаблонные правила, которые относятся к этой части ввода, врезая их вывод в соответствующее место в конечном дереве. На практике, конечно, это не так просто, и пока еще никто не реализовал инкрементный процессор таблиц стилей, который работает таким образом. Однако это наверняка сделают, и то, что выражения присваивания удалены из языка, должно способствовать этому.

Тем временем, пока исследователи и разработчики программ придумывают, как реализовать инкрементный процессор таблиц стилей, пользователи стоят перед другой проблемой: научиться программировать без выражений присваивания. После такого довольно длинного экскурса в теорию информатики вернемся в следующем разделе к основной теме и рассмотрим некоторые примеры, демонстрирующие, как это нужно делать.

Однако сначала попытаемся отделить это от другой проблемы программирования, которая есть в XSLT: это – ограниченное число имеющихся типов данных. Для правил проектирования языка отсутствие выражений присваивания и отсутствие богатой системы типов – совсем разные вопросы. В XSLT 1.0 эти две проблемы часто встречаются одновременно:

- Единственное, что может произвести шаблон, – его вывод (из-за запрета на побочные эффекты).
- И этим единственным выводом, если нужно обрабатывать его далее, является строка символов (из-за ограниченного круга возможных типов данных).

Однако в XSLT 1.1 шаблон уже может создавать дерево, которое можно использовать как исходные данные для дальнейших стадий обработки. Дерево же – это почти самая гибкая структура, которую только можно получить, так что вторую проблему можно считать решенной. В XSLT 1.0 часто приходится сталкиваться с проблемой представления сложных данных в форме строк, но введение временных деревьев в XSLT 1.1 значительно снижает искажения, которые неизбежны при реализации сложных алгоритмов для требуемых преобразований.

Почему же их называют переменными?

В XSLT, как было показано, есть переменные, которые содержат значения. Можно инициализировать переменную и присвоить ей значение, но нельзя изменить значение существующей переменной, которая уже инициализирована.

По этой причине у многих возникает вопрос, почему это называют переменной, если ее нельзя изменить? Дело в том, что здесь этот термин используется в традиционном математическом смысле: переменная – это символ, используемый для обозначения различных значений в разных случаях. Когда говорят «площадь = длина × ширина», тогда **площадь**, **длина** и **ширина** – названия, или символы, используемые для обозначения значений: здесь они обозначают свойства прямоугольника. Все они – переменные, потому что

могут принимать разные значения при применении формулы, а не потому, что данный прямоугольник меняет свои размеры со временем.

Уловка

Иногда может оказаться, что без выражений присваивания программа выходит слишком громоздкой или слишком медленной. Тогда можно прибегнуть к уловке.

Большинство XSLT-процессоров позволяет пользовательским функциям расширения иметь побочные эффекты, и можно найти немало пространных описаний того, как воспользоваться этим, чтобы реализовать замену обновляемым переменным.

Продукт Saxon идет на шаг дальше и вводит элемент расширения `<saxon:assign>`, который прямо позволяет модифицировать переменную.

> *Прочитав только что несколько страниц, объясняющих, чем хорош язык, свободный от побочных эффектов, читатели могут удивиться, почему же автор книги в своем собственном продукте вводит особенность, которая противоречит этому принципу. Тому есть несколько оправданий:*
>
> *(a) Когда это было сделано, XSLT как функциональный язык программирования был гораздо менее развит, чем сейчас.*
>
> *(b) Следовало принять во внимание и то, что хотя свободное от побочных эффектов программирование – теоретически хорошая вещь, многие пользователи еще не готовы к этому.*
>
> *(c) Наконец, хотелось проверить, превышают ли выгоды применения чистых функций вводимые ими затраты (по производительности и практичности), а лучший способ сделать это – создать неограниченный в этом плане вариант языка и посмотреть, что это даст.*

Эта возможность – крайнее средство. Большинство XSLT-процессоров фактически производит обработку предсказуемым способом, поэтому не страшно прибегнуть к такой уловке. Однако при использовании процессора, производящего дополнительную оптимизацию, такие расширения могут иметь не те побочные эффекты, которые можно было ожидать. Например, из-за непредсказуемого порядка выполнения различных инструкций файл может быть закрыт раньше, чем в него запишутся изменения. Эти средства подобны `PEEK` и `POKE` в ранних диалектах языка Basic, опасному и непоследовательному переходу на более низкий уровень программирования, которым нужно пользоваться только в крайних случаях. Тем не менее, бывают ситуации, в которых это может разительно повысить быстродействие таблиц стилей, нуждающихся в этом.

Отказ от выражений присваивания

В следующих разделах рассмотрены некоторые типичные ситуации, где выражения присваивания кажутся необходимыми, и показано, как можно достичь требуемого эффекта без них.

Условная инициализация

Эта проблема имеет простое решение, так что с ней можно быстро справиться.

В традиционных языках переменной присваивают значение нуль в одних обстоятельствах и значение один – в других. Можно записать:

```
int x;
if (zeroBased) {
   x=0;
} else {
   x=1;
}
```

Как сделать подобное в XSLT без выражений присваивания?

Ответ прост – вспомните эквивалентное выражение:

```
int x = (zeroBased ? 0 : 1 );
```

аналог которого в XSLT выглядит так:

```
<xsl:variable name="x">
   <xsl:choose>
      <xsl:when test="$zeroBased">0</xsl:when>
      <xsl:otherwise>1</xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

Единственный нюанс в том, что когда при задании значения используется содержимое элемента `<xsl:variable>`, а не атрибут `select`, значением переменной всегда будет дерево. Неважно, требуется ли строковое или численное значение, потому что дерево легко можно преобразовать и в то, и в другое, но если в качестве значения нужен набор узлов исходного документа, это будет проблемой.

Предположим, что переменная `$транзакции` должна обозначать все кредиты или все дебеты, в зависимости от значения логической переменной `$нужны-Кредиты`. Желаемого результата можно достичь следующим способом:

```
<xsl:variable name="транзакции"
         select="//кредиты[$получитьКредиты] | //дебеты[not($получитьКредиты)]"/>
```

Это дает объединение двух наборов узлов, один из которых всегда пустой. Обратите внимание на оператор объединения «`|`». Если переменная `$нужныКредиты` истинна, первый предикат «`[$нужныКредиты]`» всегда является истиной, а второй предикат «`[not($нужныКредиты)]`» всегда является ложью, поэтому выражение эквивалентно «`//кредиты`»; в то время как если переменная `$нужныКредиты` ложна, ситуация реверсируется, и выражение становится эквивалентным выражению «`//дебеты`».

Не пользуйтесь итерацией, пользуйтесь рекурсией

Одно из наиболее распространенных применений переменных в традиционном программировании – отслеживание текущей позиции в цикле. Делается ли это с помощью целочисленного счетчика в цикле `for` или объектов `Iterator` или `Enumerator` для обработки списков, принцип один и тот же: это переменная, которая указывает, как далеко нужно идти и когда закончить.

В функциональной программе этого нельзя сделать, потому что нельзя модифицировать переменные. Тогда вместо цикла следует написать рекурсивную функцию.

В обычных программах распространенный способ обработки списка элементов выглядит так:

```
iterator = list.getIterator();
while (iterator.hasMoreItems()) {
   item = iterator.getNextItem();
   item.doSomething();
}
```

Выражение присваивания, мешающее использовать этот способ в таблице стилей, – «`item = iterator.getNextItem()`». Оно каждый раз присваивает различные значения переменной `item`, а кроме того, оно полагается на `iterator`, содержащий некоторую модифицируемую переменную, которая хранит положение в списке.

В функциональной программе это делается через рекурсию, а не через итерацию. Псевдокод становится таким:

```
function обработать(список) {
   if (!неПуст(список)) {
      выполнитьДействие(начало(список));
      обработать(остаток(список));
   }
}
```

Эта функция вызывается для обработки списка объектов. Она производит требуемые действия с первым объектом списка, а затем снова вызывает себя для обработки остальной части списка. (Здесь предполагается, что функция `начало()` возвращает первый элемент списка, а `остаток()` возвращает список, содержащий все элементы, кроме первого). При каждом вызове функции список становится меньше, и когда он, наконец, становится пустым, происходит выход из функции и возврат через все рекурсивные вызовы.

> *Важно иметь ограничивающее условие, например становление списка пустым. Иначе функция будет вызывать себя бесконечно – рекурсивный эквивалент бесконечного цикла.*

Таким образом, первый прием программирования без переменных – при обработке списков использовать рекурсию вместо итерации. В XSLT не обязательно обрабатывать так все типы циклов, поскольку в XSLT есть встроен-

ные средства, такие как <xsl:apply-templates> и <xsl:for-each>, которые обрабатывают все члены набора узлов, не требуя явной управляющей переменной, а также функции sum() и count() – для выполнения некоторых простых действий с наборами узлов; но когда невозможно обработать набор объектов при помощи этих конструкций, следует использовать рекурсию.

Действительно ли рекурсия чревата издержками? Достаточно хороший компилятор вполне может сгенерировать для рекурсивной процедуры точно такой же код, как и для итерационной. Например, общей методикой компиляторов функциональных языков программирования является методика *концевой* (*хвостовой*) *рекурсии*, которая знает, что когда в конце функция вызывает саму себя, можно не выделять новый стековый фрейм или новые переменные, а можно просто вернуться к началу функции.

Следующий пример использует рекурсивный шаблон для обработки списка разделенных пробельными символами чисел, которые по какой-то причине не были размечены как отдельные элементы.

Пример: Суммирование списка чисел

Исходный документ

Предположим, имеется строка, содержащая список разделенных пробельными символами чисел, например "12 34.5 18.2 -35", и нужно найти их сумму. (Не спрашивайте зачем; когда работаешь с XML-документами, созданными кем-то другим, это может понадобиться.)

К примеру, полное содержимое документа список-чисел.xml следующее:

```
<числа>12  34.5  18.2  -35</числа>
```

Таблица стилей

Здесь нужно использовать некоторые из обрабатывающих строки функций, имеющиеся в языке выражений XPath. Они описаны в главе 7. Здесь будут использоваться следующие функции:

- normalize-space(), которая удаляет начальные и концевые пробелы и заменяет внутренние пробельные символы одним пробелом.
- concat(), которая сцепляет две строки. Она использована для добавления пробела в конце строки, чтобы обработка происходила правильно, даже при наличии в строке только одного числа.
- substring-before(), которая использована для отыскания фрагмента строки перед первым пробелом.
- substring-after(), которая использована для отыскания фрагмента строки, следующего за первым пробелом.
- number(), которая преобразует строку в число. Фактически это производится автоматически, но для ясности эта функция введена явным образом.

Ниже приведен рекурсивный шаблон (файл сумма-чисел.xsl):

```
<xsl:template name="сложить-числа">
   <xsl:param name="список"/>
   <xsl:variable name="нсписок"
      select="concat(normalize-space($список), ' ')"/>
   <xsl:choose>
      <xsl:when test="$нсписок!=' '">
         <xsl:variable name="голова"
            select="substring-before($нсписок, ' ')"/>
         <xsl:variable name="хвост"
            select="substring-after($нсписок, ' ')"/>
         <xsl:variable name="сумма">
            <xsl:call-template name="сложить-числа">
               <xsl:with-param name="список" select="$хвост"/>
            </xsl:call-template>
         </xsl:variable>
         <xsl:value-of select="number($голова) + number($сумма)"/>
      </xsl:when>
      <xsl:otherwise>0</xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Обратите внимание, как близко это отражает структуру псевдокода, приведенного ранее.

- Сначала для упрощения работы со списком, разделенным пробельными символами, заданный список нормализуется. Это делается с помощью функции normalize-space(), которая удаляет начальные и концевые пробелы и заменяет все промежуточные пробелы одним пробелом; затем используется функция concat() для добавления в конце дополнительного пробела, чтобы первое число в списке всегда имело после себя пробел, даже если оно единственное в списке. (Фактически этого не нужно делать, когда шаблон вызывает себя рекурсивно, потому что список уже будет иметь правильный формат. При желании можно дописать еще немного кода, чтобы ускорить обработку, – здесь есть возможности для оптимизации.)
- Если заданный список пуст, шаблон возвращает значение нуль. Критерий в инструкции <xsl:when> возвращает значение ложь (после нормализации строка, не содержащая чисел, будет преобразована к единственному пробелу). Тогда берется ветвь <xsl:otherwise>, и происходит возврат из шаблона, завершая тем самым рекурсию.
- В противном случае в переменную $голова с помощью функции substring-before() извлекается первое число из списка, а в переменную $хвост с помощью функции substring-after() извлекается оставшаяся часть списка. Затем шаблон рекурсивно вызывает себя для обработки оставшейся части списка и добавляет значение первого числа к сумме остальных чисел. Элемент <xsl:with-param> устанавливает параметр для вызываемого шаблона, так что в следующий раз будет обрабаты-

ваться оставшаяся часть списка, полученная после удаления первого элемента. Элемент <xsl:value-of> записывает результат в текущее дерево вывода, которым может являться переменная или конечное дерево.

Для проверки этого добавьте следующее шаблонное правило и примените таблицу стилей к приведенному выше исходному документу:

```
<xsl:template match="/">
   <xsl:call-template name="сложить-числа">
      <xsl:with-param name="список" select="."/>
   </xsl:call-template>
</xsl:template>
```

Вывод

```
29.7
```

Другой подобный пример приведен в разделе <xsl:call-template> в главе 4.

Ниже приводится еще один пример, на этот раз обрабатывающий набор узлов. В XPath есть встроенные функции для подсчета количества узлов и суммирования их значений, но они не всегда достаточно гибки: иногда следует обрабатывать узлы самостоятельно.

Пример: Определение суммы продаж

Требуется найти сумму продаж книг, используя пример со списком книг со стр. 655. Для вычисления всей суммы продаж нужно умножить число продаж каждой книги на ее цену, чего нельзя сделать с помощью функции sum(), имеющейся в XSLT.

Исходный документ

В примере используется файл книги.xml, упоминавшийся ранее в этой главе. В нем отражены данные по продажам ряда книг:

```
<книги>
   <книга>
      <название>Прах Анжелы</название>
      <автор>Фрэнк МакКорт</автор>
      <издательство>HarperCollins</издательство>
      <isbn>0 00 649840 X</isbn>
      <цена>6.99</цена>
      <продажи>235</продажи>
   </книга>
   <книга>
      <название>Меч чести</название>
      <автор>Ивлин Во</автор>
      <издательство>Penguin Books</издательство>
      <isbn>0 14 018967 X</isbn>
      <цена>12.99</цена>
      <продажи>12</продажи>
```

```
    </книга>
</книги>
```

Таблица стилей

Таблица стилей называется `сумма-продаж.xsl`. Она тоже определяет рекурсивный шаблон, называемый «сложить-продажи». Он вызывается для обработки набора элементов `<книга>`, переданных в параметре «список». Рекурсия прекращается, выдавая значение нуль, когда список становится пустым.

Когда список не пустой, шаблон умножает число продаж первой книги в списке на ее цену и добавляет сумму продаж всех остальных книг, которую он получает, вызывая себя с этим укороченным на одну книгу списком в качестве параметра. Поскольку в этот раз шаблон оперирует наборами узлов, а не строками, то для работы с ними здесь используется синтаксис XPath: в частности, предикат «`[1]`» для отыскания первого узла в наборе и «`[position()!=1]`» для определения оставшейся части.

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

<xsl:template name="сложить-продажи">
  <xsl:param name="список"/>
  <xsl:choose>
    <xsl:when test="$список">
       <xsl:variable name="голова" select="$список[1]"/>
       <xsl:variable name="сумма-остальных">
          <xsl:call-template name="сложить-продажи">
            <xsl:with-param name="список" select="$список[position()!=1]"/>
          </xsl:call-template>
       </xsl:variable>
      <xsl:value-of select="$голова/продажи * $голова/цена + $сумма-остальных"/>
    </xsl:when>
    <xsl:otherwise>0</xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Корневой шаблон просто вызывает рекурсивный шаблон для обработки набора всех книг в исходном файле и отображает результат в соответствующем формате, вызывая функцию `format-number()`, которая описана в главе 7.

```
<xsl:template match="/">
   <xsl:variable name="сумма">
        <xsl:call-template name="сложить-продажи">
            <xsl:with-param name="список" select="//книга"/>
        </xsl:call-template>
   </xsl:variable>
Сумма продаж: <xsl:value-of select="format-number($сумма, '$#.00')"/>
</xsl:template>

</xsl:stylesheet>
```

Вывод

```
Сумма продаж: $1798.53
```

Рекурсия: Резюме

Теперь принцип понятен. Всякий раз, когда нужно что-то выяснить, обрабатывая список объектов, пишите рекурсивный шаблон и передавайте ему весь список в качестве параметра. Если список не пустой, нужно обработать первый объект и произвести рекурсивный вызов для обработки остальной части списка.

Как уже говорилось, в XSLT 1.0 при выполнении этой задачи была еще одна проблема, которая не имеет отношения к отсутствию выражений присваивания, а связана с ограниченным количеством типов данных. Результатом шаблона всегда является временное дерево, а в XSLT 1.0 с ним можно делать только две вещи: копировать его в конечное дерево или преобразовывать в строку. Это означает, что если необходимо произвести дальнейшую обработку результата рекурсивной функции, нужно найти способ выражения этого результата в виде строки.

В XSLT 1.1 этой проблемы нет, так как временным деревом можно манипулировать как угодно, обрабатывая его просто как любой другой исходный документ. Фактически эта возможность настолько полезна, что она было реализована и в некоторых процессорах XSLT 1.0 в виде расширений, определенных производителями, например MSXML3 имеет функцию `msxml:node-set()`, преобразующую дерево в набор узлов. Хотя и не стоит слишком полагаться на расширения от производителей, эта функция довольно безопасна: таблицы стилей не станут зависимыми от нее, поскольку имеются эквивалентные средства от нескольких производителей, а при переходе к следующей версии стандарта XSLT эта возможность будет доступной и переносимой.

В примере с маршрутом коня по шахматной доске, приведенном в главе 10, использовано это преимущество.

Не гонитесь сразу за двумя зайцами

Еще одна распространенная ситуация, когда возникает потребность в переменных, – это попытка одновременно решать две задачи. Например, требуется копировать текст в назначение вывода и в то же время хочется отмечать, сколько текста скопировано. Может показаться, что естественным способом достижения этого является сохранение промежуточного суммарного объема скопированного текста в переменной и модифицирование этой переменной благодаря побочному эффекту шаблона, который выполняет копирование. Аналогично можно захотеть обновлять счетчик по мере вывода узлов, чтобы после каждых десяти узлов можно было начинать новую строку таблицы.

Точно так же при сканировании набора чисел можно определять сразу и минимальное, и максимальное значение или при выводе списка служащих ус-

танавливать флажок для тех, у кого зарплата превышает $100000 в год. В XSLT такие задачи следует решать иначе. Каждую часть вывода, которую требуется получить, нужно продумывать отдельно, и писать для каждой функцию (или шаблонное правило), которое генерирует данную часть вывода из исходных данных. Не следует в это же время выполнять вычисления.

Таким образом, нужно написать одну функцию для генерирования вывода, а другую – для вычисления суммы. Точно так же – написать один шаблон для нахождения минимального значения и другой – для максимального.

Для этого придется писать немного больше кода, и может потребоваться больше времени для повторной обработки, но обычно это – верный подход. Проблему повторной обработки часто можно решить с помощью введения переменных для наборов узлов, используемых при обоих вычислениях: если какой-то набор узлов используется как ввод более чем для одного процесса, можно сохранить его в переменной, которая тогда будет передаваться в качеств параметра каждому из этих шаблонов.

Вариант, который иногда может оказаться более эффективным, – создание шаблона, который возвращает составной результат. В XSLT 1.0 это можно сделать единственным удобным способом: записать результат в виде строк, разделенных пробельными символами. Однако в XSLT 1.1 (или с процессором XSLT 1.0, который имеет функцию расширения `node-set()`) можно вернуть составной результат, структурированный в виде дерева. Вызванный код может тогда обращаться к индивидуальным узлам этого дерева, используя выражение XPath. Например, следующий рекурсивный шаблон, которому набор узлов передан в виде параметра, создает дерево, содержащее два элемента: `<min>` и `<max>`, соответствующие минимальному и максимальному значению узлов. Рабочая таблица стилей, основанная на этом примере, – файл `minimax.xsl`; этот же файл служит и исходным документом для этой таблицы стилей.

```
<xsl:template name="get-min-and-max">
  <xsl:param name="nodes"/>
  <xsl:param name="best-so-far">
    <min><xsl:value-of select="999999999"/></min>
    <max><xsl:value-of select="-999999999"/></max>
  </xsl:param>
  <xsl:choose>
    <xsl:when test="$nodes">
      <xsl:variable name="new-best-so-far">
      <min>
        <xsl:choose>
          <xsl:when test="$nodes[1] &lt; $best-so-far/min">
            <xsl:value-of select="$nodes[1]"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="$best-so-far/min"/>
          </xsl:otherwise>
        </xsl:choose>
```

```
        </min>
         <max>
          <xsl:choose>
            <xsl:when test="$nodes[1] &gt; $best-so-far/max">
              <xsl:value-of select="$nodes[1]"/>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of select="$best-so-far/max"/>
            </xsl:otherwise>
          </xsl:choose>
        </max>
        </xsl:variable>
        <xsl:call-template name="get-min-and-max">
           <xsl:with-param name="nodes" select="$nodes[position() &gt; 1]"/>
          <xsl:with-param name="best-so-far" select="$new-best-so-far"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy-of select="$best-so-far"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
```

При вызове этого шаблона можно позволить второму параметру, `$best-so-far`, принять значение по умолчанию:

```
<xsl:variable name="min-and-max">
  <xsl:call-template name="get-min-and-max">
    <xsl:with-param name="nodes" select="//item/цена"/>
  </xsl:call-template>
</xsl:variable>
Minimum price is: <xsl:value-of select="$min-and-max/min"/>.
Maximum price is: <xsl:value-of select="$min-and-max/max"/>.
```

Очень удобно сохранять промежуточные результаты в переменной, значением которой является дерево, а затем использовать их в качестве ввода для нескольких процессов в тех случаях, когда эти промежуточные результаты должны сортироваться. Когда требуется отсортировать большой набор узлов, не хочется проделывать это несколько раз. Решением может быть преобразование в два прохода: в первом проходе создается отсортированное дерево, а во втором проходе выполняются преобразования с этим отсортированным деревом. В XSLT многоступенчатого преобразования можно достичь двумя способами:

- Создать временное дерево, в котором узлы отсортированы в нужном порядке. Затем использовать `<xsl:for-each>` или `<xsl:apply-templates>`, чтобы по порядку обрабатывать узлы этого дерева. Это аналогично предыдущему примеру с поиском максимального и минимального значений; и для выполнения такой задачи нужен процессор XSLT 1.1 или XSLT 1.0 с функцией расширения `node-set()`.

- Использовать последовательность (или **цепочку**) таблиц стилей: первая таблица стилей создает документ, в котором узлы отсортированы в правильном порядке, а последующие таблицы стилей используют этот документ в качестве ввода. Такой цепочкой таблиц стилей удобно управлять, используя интерфейс TrAX, описанный в приложении F.

Следует отметить, что ни один из перечисленных приемов не нарушает принципа проектирования XSLT по поводу отсутствия побочных эффектов.

Группировка

Типичная задача обработки, для которой необходимость в модифицируемой переменной кажется очевидной, – разбиение данных на группы.

Предположим, что в исходном документе находятся города и соответствующие им страны в следующем формате:

```
<города>
   <город название="Город" страна="Страна"/>
… и т. д. …
</города>
```

Однако требуется создать список городов, находящихся в одной стране:

```
<страны>
   <страна название="Страна">
      <город>Город1</город>
      <город>Город2</город>
   </страна>
… и т. д. …
</страны>
```

На других языках, вероятно, все добивались бы требуемого результата, используя псевдокод такого типа:

```
отсортированныеГорода = города.сортироватьПо("страна");
предыдущаяСтрана = null;
вывести("<страна>")
для каждого город в отсортированныеГорода {
   даннаяСтрана = город.атрибут("страна");
   if (даннаяСтрана != предыдущаяСтрана) {
      вывести("</страна>\n<страна>");
   }
   вывести("<город>" + город.атрибут("город") + "</город>");
   предыдущаяСтрана = даннаяСтрана;
}
вывести("</страна>")
```

В XSLT это, конечно, не годится по двум причинам. Во-первых, требуется вывести дерево, а не текстовый файл, содержащий теги разметки: это означает, что нельзя записывать закрывающий и открывающий теги элемента как две отдельные операции. Во-вторых, по мере обработки данных для оп-

ределения изменения страны нельзя воспользоваться выражениями присваивания.

В XSLT для решения этой проблемы лучше сначала сформировать список всех стран, упомянутых в документе, а затем обрабатывать каждую страну в этом списке по очереди. Рассмотрим пример.

Пример: Разбиение данных на группы

Исходный документ

В этом примере используется файл `города.xml`. В нем отображены города и соответствующие им страны, как показано ниже:

```
<города>
   <город название="Париж" страна="Франция"/>
   <город название="Рим" страна="Италия"/>
   <город название="Ницца" страна="Франция"/>
   <город название="Мадрид" страна="Испания"/>
   <город название="Милан" страна="Италия"/>
   <город название="Фиренце" страна="Италия"/>
   <город название="Неаполь" страна="Италия"/>
   <город название="Лион" страна="Франция"/>
   <город название="Барселона" страна="Испания"/>
</города>
```

Таблица стилей

Список уникальных стран (в виде набора узлов, содержащего нужные узлы атрибутов) можно получить следующим образом. Таблица стилей находится в файле `уникальные-города.xsl`:

```
<xsl:template match="/">
<xsl:variable name="уникальные-страны"
   select="/города
           /город[not(@страна=preceding-sibling::город/@страна)]
           /@страна"
/>
```

Предикат элемента `<город>` отклоняет любой город, атрибут `страна` которого такой же, как у предшествующего одноуровневого элемента, если он есть. Другими словами, выбирается только первый встретившийся город каждой страны. Можно использовать другие предикаты:

- Если известно, что города отсортированы по странам, можно заменить предикат: `preceding-sibling::город` на `preceding-sibling::город[1]`. Это означает, что страна будет сравниваться только со страной непосредственно предшествующего данному элементу города. Этот путь может оказаться эффективнее.

- Если бы элементы <город> не были бы одноуровневыми, можно было бы использовать ось preceding, вместо оси preceding-sibling. Однако это удлинило бы поиск.

Сформировав список уникальных стран, можно затем генерировать для каждой из них элементы <город>:

```
<xsl:template match="/">
<xsl:variable name="уникальные-страны"
   select="/города
           /город[not(@страна=preceding-sibling::город/@страна)]
           /@страна"
/>
<страны>
   <xsl:for-each select="$уникальные-страны">
      <страна name="{.}">
         <xsl:for-each select="//город[@страна=current()]">
            <город><xsl:value-of select="@название"/></город>
         </xsl:for-each>
      </страна>
   </xsl:for-each>
</страны>
</xsl:template>
```

Вывод

```
<страны>
   <страна название="Франция">
      <город>Париж</город>
      <город>Ницца</город>
      <город>Лион</город>
   </страна>
   <страна название="Италия">
      <город>Рим</город>
      <город>Милан</город>
      <город>Фиренце</город>
      <город>Неаполь</город>
   </страна>
   <страна название="Испания">
      <город>Мадрид</город>
      <город>Барселона</город>
   </страна>
</страны>
```

Повышение эффективности

Основная проблема с такими алгоритмами состоит в том, что если требуется сгруппировать N узлов, то каждый из них должен сравниваться со всеми предыдущими, что в сумме дает $N \times (N - 1) / 2$ сравнений. Группировка удвоенного количества узлов потребует приблизительно в четыре раза больше времени. Результат может быть приемлем, если речь идет о десяти городах, но придется запастись очень большим терпением, если

будет задано несколько тысяч городов. Конечно, умный XSLT-процессор мог бы оптимизировать запрос и найти более эффективный способ выполнения этой задачи, но на это не стоит полагаться. Технология XSLT-процессоров еще молода, и методы оптимизации запросов еще только развиваются.

Одно из решений – сначала отсортировать узлы, а затем их сгруппировать. На стадии группировки каждый узел должен сравниваться только со своим непосредственным предшественником, как показано в примере. Это требует двухступенчатого преобразования, обсуждавшегося ранее: на первой ступени создается временное дерево, узлы которого отсортированы, а на второй это дерево обрабатывается.

Ниже приведена переработанная таблица стилей `отсортированные-города.xsl`, в которой использован этот подход. Она рассчитана на XSLT 1.1, который позволяет обрабатывать временное дерево `$отсортированные-города` как набор узлов.

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.1">

<xsl:output indent="yes"/>

<xsl:template match="/">
<xsl:variable name="отсортированные-города">
  <xsl:for-each select="//город">
    <xsl:sort select="@страна"/>
    <xsl:copy-of select="."/>
  </xsl:for-each>
</xsl:variable>
<xsl:variable name="уникальные-страны"
   select="$отсортированные-города
           /город[not(@страна=preceding-sibling::город[1]/@страна)]
           /@страна"
/>
<страны>
   <xsl:for-each select="$уникальные-страны">
      <страна name="{.}">
         <xsl:for-each select="$отсортированные-города/город[@страна=current()]"
>
            <город><xsl:value-of select="@название"/></город>
         </xsl:for-each>
      </страна>
   </xsl:for-each>
</страны>
</xsl:template>

</xsl:stylesheet>
```

Это повысит эффективность, если только XSLT-процессор достаточно умен, чтобы заметить, что он может определить значение «preceding-

sibling::город[1]», посмотрев лишь на непосредственно предшествующий элемент, вместо поиска всех предшествующих одноуровневых элементов и проверки, равна ли 1 позиция position() каждого из них на оси. Однако это довольно простая оптимизация, так что любой уважающий себя XSLT-процессор должен ею владеть.

Конечно, для такого крошечного исходного файла данных, который используется для демонстрации, разница будет незначительной, но для списка из тысячи городов она будет ощутимой.

Метод группировки Мюнча

Существует еще один, лучший способ решения этой проблемы группировки. Это метод группировки Мюнча, названный по имени Стива Мюнча (Steve Muench) из Oracle, который его изобрел. Он основан на ключах. В следующем примере показано, как этот метод работает.

Пример: Группировка методом Мюнча

Исходный документ

В этом примере, как и в предыдущем, используется файл города.xml.

Таблица стилей

Эта таблица стилей находится в файле мюнчевы-города.xsl.

Требуется сгруппировать города по странам, в которых они находятся, поэтому на первой стадии нужно определить значение ключа для группировки:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

<xsl:output indent="yes"/>

<xsl:key name="ключ-страна" match="город" use="@страна"/>
```

Это означает, что для отыскания всех элементов <город>, атрибут @страна которых имеет значение «Франция», можно вызвать функцию «key('ключ-страна', 'Франция')».

Это также означает, что «key('ключ-страна', 'Франция')[1]» будет искать первый встретившийся город Франции.

Кроме того, это означает, что следующее выражение будет истинно только для первого встретившегося города страны (в порядке появления в документе):

```
generate-id(.)=generate-id(key('ключ-страна', @страна)[1])
```

Фактически предикат «[1]» здесь избыточен, потому что функция generate-id() обрабатывает только первый узел в заданном наборе узлов.

Ниже приведен другой способ записи этого критерия:

```
count(. | key('ключ-страна', @страна)[1]) = 1
```

Вместо критерия «город[not(@страна=preceding-sibling::город/@страна)]», применявшегося в предыдущем примере, можно использовать любое из этих выражений. Таким образом, таблица стилей принимает такой вид:

```
<xsl:key name="ключ-страна" match="город" use="@страна"/>

<xsl:template match="/">
<xsl:variable name="уникальные-страны"
   select="/города
           /город[generate-id(.)=
                 generate-id(key('ключ-страна', @страна))]
           /@страна"
/>
<страны>
   <xsl:for-each select="$уникальные-страны">
      <страна name="{.}">
         <xsl:for-each select="key('ключ-страна', .)">
            <город><xsl:value-of select="@название"/></город>
         </xsl:for-each>
      </страна>
   </xsl:for-each>
</страны>
</xsl:template>
```

Вывод

Вывод такой же, как и в предыдущем примере.

Группировка методом Мюнча обычно выполняется быстрее, чем прямой подход с использованием оси preceding-sibling для проверки, является ли узел первым в его группе, потому что ключи используют индекс или хеш-таблицу, с помощью которых можно быстрее найти все узлы с одинаковым значением ключа. Еще одно преимущество в том, что группировку методом Мюнча можно использовать в тех случаях, когда выражение для группировки (общее значение, которое определяет, принадлежат ли два узла одной и той же группе) не является строковым значением узла, как это было в вышеупомянутом примере. Типичный пример – когда требуется создать группы по алфавиту, для каждой буквы алфавита. Например, если нужно перечислить города в алфавитном порядке, с заголовком для каждой буквы алфавита, можно определить ключ следующим образом:

```
<key name="alpha-group" match="город" use="substring(@название, 1, 1)"/>
```

а в остальном логика таблицы стилей будет аналогичной.

Проблемы группировки очень часто возникают при программировании таблиц стилей XSLT, и разрешить их бывает довольно сложно: требуется освоиться с программированием без выражений присваивания. На практике эти проблемы обычно решаемы, хотя в некоторых случаях требуется писать для этого рекурсивные шаблоны.

Рабочая группа консорциума W3C, ответственная за развитие языка, признает, что требуются расширения к языку в этой области, чтобы сделать группировку столь же простой, как, скажем, сортировку или нумерацию. Пока неясно, в чем будут заключаться эти расширения, хотя их направленность можно предположить на основании требований к XSLT 2.0, опубликованных по адресу *http://www.w3.org/TR/XSLT20req*. Есть вероятность, что это будет нечто подобное элементу расширения `<saxon:group>`, содержащемуся в программном продукте Saxon, или аналогичным расширениям в XSLT-процессоре Unicorn. Таблица стилей Saxon для группировки обсуждавшегося списка городов по странам выглядела бы следующим образом:

```
<страны>
<saxon:group select="//город" group-by="@страна">
   <xsl:sort select="@страна"/>
   <страна name="{@страна}">
      <saxon:item>
         <город><xsl:value-of select="@название"/></город>
      </saxon:item>
   </страна>
</saxon:group>
</страны>
```

Более подробно об этом см. в приложении C.

Резюме

В этой главе были рассмотрены четыре образца проектирования таблиц стилей XSLT:

- Для заполнения бланков
- Навигационный
- Основанный на правилах
- Вычислительный

В последней категории описан, возможно, незнакомый читателям подход ко многим проблемам, так как XSLT – чисто функциональный язык программирования без выражений присваивания или других побочных эффектов, которые накладывают ограничения на последовательность обработки. Результатом этого является необходимость написания многих сложных алгоритмов с использованием именованных рекурсивных шаблонов.

10

Действующие примеры

Основная задача этой главы – продемонстрировать, как работают вместе все средства языка XSLT для решения некоторых реальных, довольно сложных проблем обработки XML. Здесь выбраны три приложения и приведены полноценные таблицы стилей для их обработки. В этой главе код представлен по частям, а полные таблицы стилей и образцы файлов данных можно загрузить с веб-сайта издательства Wrox *http://www.wrox.com/*.

В предыдущей главе было показано, что XSLT имеет широкую область применения, а в этой главе будет рассмотрена репрезентативная выборка проблем. Здесь приведены следующие три примера:

- Первый пример – таблица стилей для отображения последовательных документов, в частности таблица стилей, используемая для отображения W3C-спецификаций, например рекомендаций XML и XSLT. Это классический пример образца проектирования, *основанного на правилах*, который описан в соответствующем разделе главы 9.
- Второй пример связан с представлением структурированных данных. Здесь выбрана сложная структура данных со множеством перекрестных ссылок, чтобы продемонстрировать, как навигационная таблица стилей обрабатывает исходное дерево. Для примера взят файл данных, содержащий генеалогическое дерево семьи Кеннеди.
- Последний пример таблицы стилей довольно нереалистичен, но забавен. В нем показано, как можно использовать XSLT для вычисления маршрута коня по шахматной доске, при условии, что конь должен посетить каждую клетку, но при этом ни в одной клетке не останавливаться дважды. XSLT не предназначен для решения таких задач, но он может справиться с ними, и этот пример убедительно доказывает, что XSLT безусловно обладает достаточной вычислительной мощью и гибкостью, чтобы решать куда более скромные алгоритмические проблемы, каждый день возникающие в форматирующих приложениях.

Форматирование спецификации XML

На этом действующем примере мы изучим таблицу стилей, используемую для форматирования самих спецификаций XML. На веб-сайте W3C можно получить спецификации стандартов XML, XSLT и XPath либо в формате XML, либо в HTML. Рассмотрим таблицу стилей для преобразования рекомендации XML из формата XML в формат HTML, показанный ниже:

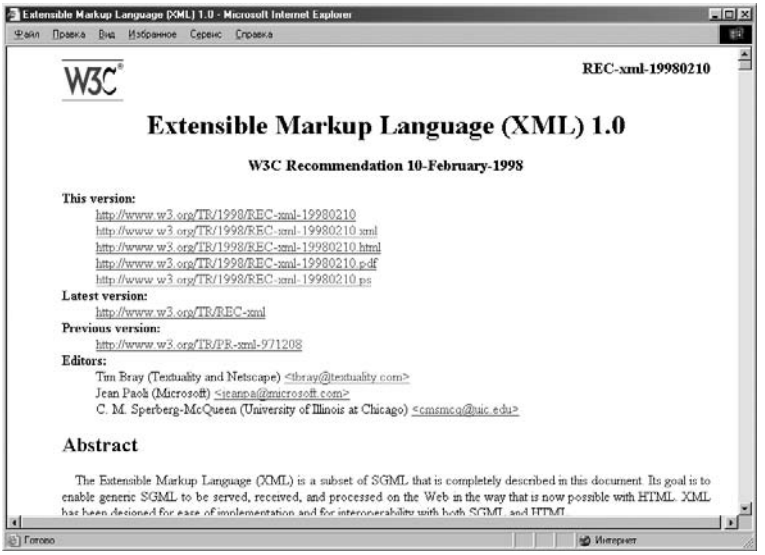


Рис. 10.1. *Пример форматирования спецификации XML. Полученный вывод в окне броузера*

Таблицы стилей, используемые для спецификаций XSLT и XPath, адаптированы из версии для спецификации XML, и адаптированные версии будут также вкратце обсуждены здесь.

Эта таблица стилей была первоначально написана Эдуардо Гутентагом (Eduardo Gutentag), а позднее модернизирована Джеймсом Кларком (James Clark). Автор благодарен Джеймсу Кларку за то, что он сделал ее всеобщим достоянием. В варианте для книги произведены некоторые изменения в компоновке таблицы и последовательности правил, а также удалены некоторые дублирующиеся правила, приводящие к появлению предупреждений XSLT-процессоров, выполняющих тщательную проверку; но не изменена логика таблицы стилей. Файл этой главы на *http://www.wrox.com/* содержит оригинал таблицы стилей Джеймса Кларка.

Данная таблица стилей – классический пример образца проектирования, *основанного на правилах*, который обсуждался в главе 9. Он делает минимум предположений о расположении различных элементов исходного XML-документа относительно друг друга, и это позволяет свободно добавлять новые правила по мере развития структуры документа.

Полезно во время изучения этой таблицы стилей иметь под рукой исходный XML-документ. Его официальная версия находится по адресу *http://www.w3.org/TR/1998/REC-xml-19980210.xml*, но с ней могут быть проблемы. Казалось бы, поскольку спецификация написана экспертами в XML, она должна представлять собой образец совершенного XML. К сожалению, это не так, и в ней использованы конструкции, с которыми не очень ладят некоторые синтаксические анализаторы. В частности:

- В ней есть объявление сущности `<!ENTITY lt "<">`, которое нарушает правило, заключающееся в том, что текст замены анализируемой сущности должен быть корректным. Это объявление заставляет IE5 в ужасе хвататься за голову и отказываться отображать документ. Собственно, это объявление не так уж важно и нужно только для совместимости с SGML.
- Более важный нюанс состоит в том, что DTD (под названием `spec.dtd`) содержит ссылки на параметрические сущности, которых на самом деле не существует или объявления которых даются после ссылки. Все эти ссылки содержатся в комментариях DTD. В оригинале спецификации XML правила для параметрических сущностей, помещенных в комментарии, были нечеткими. Позднее они были переработаны в исправлениях к документу, но пока еще существуют синтаксические анализаторы, реализующие иную интерпретацию спецификации.

Во избежание этих проблем в книге используются слегка модифицированные версии исходного документа и DTD; они включены в загружаемые файлы для этой книги. Исходный файл – `REC-XML-19980210.XML`, а файл с DTD – `spec.dtd`. Исходный файл можно просматривать в текстовом редакторе или в используемом по умолчанию в IE5 средстве просмотра XML.

XML-версии спецификаций XSLT и XPath не имеют таких проблем, и можно просматривать их прямо на веб-сайте W3C, используя броузер IE5, хотя для удобства их копии есть и на веб-сайте Wrox. В этих документах нет ссылок на таблицу стилей, поэтому IE5 будет использовать заданную по умолчанию таблицу стилей, которая отображает XML в виде дерева. В этом случае можно использовать команду View Source, чтобы отобразить исходный код XML и при желании сохранить его на своем жестком диске.

Теперь рассмотрим таблицу стилей `xmlspec.xsl`.

Пролог

Начнем с самого начала:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!-- 1999/01/11 SMI; Style Sheet for the XML and XSL
      Recommendations and Working Drafts;
      written by Eduardo Gutentag -->
<!-- v 1.28 1999/11/15 12:58:16 by James Clark -->

<!DOCTYPE xsl:stylesheet [
<!ENTITY copy   "&#169;">
```

```
<!ENTITY nbsp   " ">
]>

<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"
      doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"/>

<xsl:param name="w3">http://www.w3.org/</xsl:param>
```

Таблица стилей является XML-документом, поэтому она начинается с объявления XML; выбранная кодировка символов – iso-8859-1, так как она более универсальна для большинства редакторов, чем UTF-8. В действительности все символы являются символами ASCII, так что задание кодировки UTF-8 ничего бы не изменило.

Затем идет пара комментариев, содержащих перечень изменений и авторство.

Объявление `<!DOCTYPE>` не ссылается на внешнее DTD; это обычная практика для таблиц стилей, так как DTD ограничило бы возможность включения в шаблон любого желаемого элемента и считало бы его конечным литеральным элементом. Задача `<!DOCTYPE>` – объявить пару сущностей для специальных символов. Они позволяют записывать символ авторского права как «`&copy;`», а символ неразрывного пробела – как «` `». Если не объявить эти сущности, то нужно записывать их при каждом появлении как «`&#169;`» и «` `», соответственно.

> *Использование в этой таблице стилей ссылок на сущности, а не цифровых ссылок на эти символы, не означает, что именно так они будут отображаться в выводе HTML. То, как XSLT-процессор представляет эти символы в выводе, зависит от реализации, но независимо от выбираемого им представления, в броузере они отобразятся правильно.*

Элемент `<xsl:stylesheet>` читателям уже знаком. Элемент `<xsl:output>` определяет метод вывода как `html` и указывает открытый идентификатор версии генерируемого HTML, который будет копироваться в генерируемый выходной файл. Это хорошая практика. Это не так существенно, но, конечно, консорциум W3C хочет создавать HTML-код, который соответствует его собственным рекомендациям.

Инструкция `<xsl:param>` объявляет глобальный параметр, названный `w3`, и дает ему значение по умолчанию, которым является URL веб-сайта W3C. Задание его как глобального параметра, а не как глобальной переменной позволяет передавать различные значения во время тестирования, например, URL-сервера, используемого для тестирования.

Создание общей структуры HTML

Теперь начнем с основных шаблонных правил для документа. Порядок может быть произвольным, и здесь он отличается от порядка, использованного

авторами в оригинале. Сделано это, чтобы представить аналогичные правила вместе.

```
<xsl:template match="spec">
   <html>
      <head>
         <title>
            <xsl:value-of select="header/title"/>
         </title>
         <link rel="stylesheet" type="text/css"
               href="{$w3}StyleSheets/TR/W3C-{
                     substring-before(header/w3c-designation,'-')}"/>
         <!-- This stops Netscape 4.5 from messing up. -->
         <style type="text/css">
            <xsl:apply-templates select="." mode="css"/>
         </style>
      </head>
      <body>
         <xsl:apply-templates/>
      </body>
   </html>
</xsl:template>
```

Это шаблонное правило верхнего уровня, создающее общую структуру HTML. Здесь нет правила, соответствующего корневому узлу, поэтому здесь работает встроенный шаблон для корневого узла, и он находит правило, которое соответствует самому внешнему элементу, в данном случае — элементу `<spec>`.

Кроме создания общей структуры HTML, этот шаблон делает три вещи: он извлекает заголовок документа, выводит ссылку на таблицу стилей CSS, которую нужно использовать (в двух различных форматах, так что это работает с различными броузерами), а внутри HTML-тега `<body>` он вызывает `<xsl:apply-templates>`, чтобы обработать дочерние элементы элемента `<spec>` в исходном документе.

Вызов `<xsl:apply-templates mode="css"/>` вызывает следующее небольшое шаблонное правило, которое просто выводит стандартное содержимое элемента `<style>`. Заметьте, что с тем же успехом это можно было сделать с помощью `<xsl:call-template>` или записать соответствующий код прямо в таблице стилей. Непонятно, почему авторы выбрали именно этот способ: возможно, они хотели переопределить это правило в другом модуле таблицы стилей, импортирующем этот модуль. Вероятно также, что во время написания этой таблицы стилей некоторые возможности XSLT, в частности именованные шаблоны, еще не были реализованы в программных продуктах.

```
<xsl:template match="spec" mode="css">
   <xsl:text>code { font-family: monospace }</xsl:text>
</xsl:template>
```

Форматирование заголовка документа

Заголовок HTML-документа генерируется из элемента <title>, находящегося в пределах элемента <header> исходного XML-документа. Для понимания таких запросов следует рассмотреть структуру элемента <header> в исходном документе. В сокращенном виде структура спецификации XML начинается так:

```
<spec>
   <header>
      <title>Extensible Markup Language (XML) 1.0</title>
      <version></version>
      <w3c-designation>REC-xml-&iso6.doc.date;</w3c-designation>
      <w3c-doctype>W3C Recommendation</w3c-doctype>
      <pubdate>
         <day>&draft.day;</day>
         <month>&draft.month;</month>
         <year>&draft.year;</year>
      </pubdate>

      <publoc>
         <loc  href="url">url</loc>
         <loc  href="url">url</loc>
      </publoc>
      <latestloc>
         <loc  href="url">url</loc>
      </latestloc>
      <prevlocs>
         <loc  href="url">url</loc>
      </prevlocs>

      <authlist>
         <author>
            <name>Tim Bray</name>
            <affiliation>Textuality and Netscape</affiliation>
            <email href="mailto:tbray@textuality.com">
                        tbray@textuality.com</email>
      </author>
         остальные авторы
      </authlist>

      <abstract>
         <p>The Extensible Markup Language (XML) is a subset of SGML
            that is completely described in this document...</p>
      </abstract>

      <status>
      <p>This document has been reviewed by W3C Members and
                              other interested parties ...</p>

      <p>This document specifies a syntax… It is a product of the W3C
                  XML Activity, details of which can be found at <loc
                  href='url'>url</loc>.  A list of current W3C
```

```
                Recommendations … can be found at <loc
                href='url'>url</loc>.</p>

      <p>This specification uses the term URI, which is defined by
            <bibref ref="Berners-Lee"/>, a work in progress expected
                to update
            <bibref ref="RFC1738"/> and <bibref ref="RFC1808"/>. </p>
      </status>
   </header>
   <body>
      основной раздел документа
   </body>
   <back>
      приложения
   </back>
</spec>
```

Обратите внимание, что некоторые теги являются структурными элементами с предсказуемой вложенностью, в то время как другие, например <loc>, могут появляться в любых местах, даже внутри текста.

Несколько следующих шаблонных правил касаются обработки данного заголовка:

```
<xsl:template match="header">
   <div class="head">

      <!-- выводим логотип W3C со ссылкой-->

      <a href="http://www.w3.org/">
         <img src="{$w3}Icons/WWW/w3c_home"
               alt="W3C" height="48" width="72"/>
      </a>

      <!-- выводим заголовок и версию документа -->

      <h1><xsl:value-of select="title"/><br/>
          <xsl:value-of select="version"/>
      </h1>

      <!-- выводим тип публикации и дату -->

      <h2>
         <xsl:value-of select="w3c-doctype"/>
         <xsl:text> </xsl:text>
         <xsl:value-of select="pubdate/day"/>
         <xsl:text> </xsl:text>
         <xsl:value-of select="pubdate/month"/>
         <xsl:text> </xsl:text>
         <xsl:value-of select="pubdate/year"/>
      </h2>

      <!-- выводим URL последующих версий и имена авторов -->

      <dl>
         <xsl:apply-templates select="publoc"/>
```

```
         <xsl:apply-templates select="latestloc"/>
         <xsl:apply-templates select="prevlocs"/>
         <xsl:apply-templates select="authlist"/>
      </dl>

      <!-- выводим стандартную информацию о правообладании -->

      <xsl:call-template name="copyright"/>
      <hr title="Separator for header"/>
   </div>

   <!-- выводим разделы описания и состояния -->

   <xsl:apply-templates select="abstract"/>
   <xsl:apply-templates select="status"/>
</xsl:template>

<!-- шаблонные правила для местоположения документа -->

<xsl:template match="publoc">
   <dt>This version:</dt>
   <dd><xsl:apply-templates/></dd>
</xsl:template>

<xsl:template match="latestloc">
   <dt>Latest version:</dt>
   <dd><xsl:apply-templates/></dd>
</xsl:template>

<xsl:template match="prevlocs">
   <dt>
      <xsl:text>Previous version</xsl:text>
      <xsl:if test="count(loc)>1">s</xsl:if>
      <xsl:text>:</xsl:text>
   </dt>
   <dd><xsl:apply-templates/></dd>
</xsl:template>

<xsl:template match="publoc/loc | latestloc/loc | prevlocs/loc">
   <a href="{@href}"><xsl:apply-templates/></a>
   <br/>
</xsl:template>

<!-- шаблонные правила для авторов и информации об их членстве в различных
организациях-->

<xsl:template match="authlist">
   <dt>
      <xsl:text>Editor</xsl:text>
      <xsl:if test="count(author)>1">s</xsl:if>
      <xsl:text>:</xsl:text>
   </dt>
   <dd><xsl:apply-templates/></dd>
</xsl:template>

<xsl:template match="author">
   <xsl:apply-templates/><br/>
```

```
</xsl:template>

<xsl:template match="author/name">
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="author/affiliation">
   <xsl:text> (</xsl:text>
      <xsl:apply-templates/>
   <xsl:text>) </xsl:text>
</xsl:template>
<xsl:template match="author/email">
   <a href="{@href}">
      <xsl:text>&lt;</xsl:text>
      <xsl:apply-templates/>
      <xsl:text>&gt;</xsl:text>
   </a>
</xsl:template>

<!-- шаблоны для отображения описания и состояния -->

<xsl:template match="abstract">
   <h2><a name="abstract">Abstract</a></h2>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="status">
   <h2><a name="status">Status of this document</a></h2>
   <xsl:apply-templates/>
</xsl:template>
```

Так много нужно для заголовка. Здесь нет ничего особенно сложного, хотя следует отметить введение многочисленных указаний на предыдущие версии («previous versions») и редакторы («editors»), а также использование `<xsl:text>`, чтобы избежать копирования в вывод нежелательных пробельных символов.

Оглавление

Следующая часть таблицы стилей имеет дело с телом документа, а также с его концовкой («back»), которая имеет аналогичную структуру. Элемент `<back>` содержит приложения, библиографию и так далее.

```
<xsl:template match="body">
   <h2><a name="contents">Table of contents</a></h2>
   <xsl:call-template name="toc"/>
   <hr/>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="back">
   <hr title="Separator from footer"/>
   <xsl:apply-templates/>
</xsl:template>
```

Шаблонное правило для элемента <body> построено по стандартной схеме: сначала оно вызывает шаблон для формирования оглавления, а затем использует <xsl:apply-templates> для обработки собственных дочерних элементов.

Посмотрим, как создается оглавление. Структура элемента <body>, а также элемента <back> представляет собой последовательность элементов <div1>, соответствующих разделам верхнего уровня:

```
<div1>
   <head>First-level heading</head>
   <p>Some text</p>
   <div2>
      <head>Second-level heading</head>
      <p>Some more text</p>
      <div3>
         <head>Third-level heading</head>
         <p>Lots more text</p>
      </div3>
   </div2>
</div1>
```

Каждый элемент <div1> содержит элемент <head>, задающий его заголовок и абзацы непосредственно его текста, а кроме того в него могут входить элементы <div2>, содержащие подразделы второго уровня. Элемент <div2> также содержит элемент <head> и может содержать элементы <div3> – подразделы третьего уровня, и так далее.

В разделе <back> ненормативное приложение представлено элементом <inform-div1> вместо обычного <div1>, а в остальном структура та же самая.

> *Термин «ненормативный» является жаргонным определением, означающим «только для сведения, а не как компонент спецификации».*

Вот так выглядит именованный шаблон для формирования оглавления. Сначала он обрабатывает вложенные разделы элемента <body>, а затем снова делает то же самое для разделов в элементе <back>. За фактическое создание перекрестных ссылок отвечает другой именованный шаблон, «makeref»:

```
<xsl:template name="toc">
   <xsl:for-each select="/spec/body/div1">
      <xsl:call-template name="makeref"/><br/>
      <xsl:for-each select="div2">
         <xsl:text>    </xsl:text>
         <xsl:call-template name="makeref"/><br/>
         <xsl:for-each select="div3">
            <xsl:text>    </xsl:text>
            <xsl:text>    </xsl:text>
            <xsl:call-template name="makeref"/><br/>
         </xsl:for-each>
      </xsl:for-each>
   </xsl:for-each>
```

```
    <h3>Appendices</h3>
    <xsl:for-each select="/spec/back/div1 | /spec/back/inform-div1">
       <xsl:call-template name="makeref"/><br/>
       <xsl:for-each select="div2">
          <xsl:text>    </xsl:text>
          <xsl:call-template name="makeref"/><br/>
          <xsl:for-each select="div3">
             <xsl:text>    </xsl:text>
             <xsl:text>    </xsl:text>
             <xsl:call-template name="makeref"/><br/>
          </xsl:for-each>
       </xsl:for-each>
    </xsl:for-each>
</xsl:template>

<xsl:template name="makeref">
    <xsl:apply-templates select="." mode="number"/>
    <xsl:choose>
       <xsl:when test="@id">
          <a href="#{@id}">
             <xsl:value-of select="head"/>
          </a>
       </xsl:when>
       <xsl:otherwise>
          <a href="#section-{translate(head,' ','-')}">
             <xsl:value-of select="head"/>
          </a>
       </xsl:otherwise>
    </xsl:choose>
   <xsl:for-each select="head">
       <xsl:call-template name="inform"/>
   </xsl:for-each>
</xsl:template>

<xsl:template name="inform">
    <xsl:if test="parent::inform-div1">
       <xsl:text> (Non-Normative)</xsl:text>
    </xsl:if>
</xsl:template>
```

Шаблон «makeref» сначала генерирует порядковый номер раздела, а затем создает гиперссылку на него, используя атрибут id раздела, если он есть, а если нет – название, полученное из его заголовка (пробелы заменяются дефисами). Это не самая блестящая идея, так как любым неуникальным заголовкам, например «Example», идентификаторы id должны назначаться вручную: возможно, авторы воспользовались бы функцией generate-id(), если бы она существовала в то время. Затем, если раздел ненормативный (элемент <inform-div1>), шаблон выводит строку «(Non-Normative)».

Способ, которым это достигнуто, не особо изящен, хотя и работает. Возможно, шаблон под названием «inform» *первоначально был написан для отображения фактического заголовка раздела на месте, а затем повторно использовался для*

оглавления. Более естественным подходом было бы включение в шаблон «makeref» *критерия* <xsl:if self::inform-div1>.

Шаблон «makeref» передает задачу назначения порядкового номера шаблонному правилу, связанному с разделом, который нужно пронумеровать. Он использует специальный режим, чтобы избежать конфликтов с шаблонным правилом для вставки содержимого этого раздела. Есть две версии этого правила: одна для разделов в теле документа, а вторая – для приложений. Для разделов различных уровней отдельные правила не нужны, потому что способ форматирования номеров достаточно гибок, чтобы дать правильный результат для каждого уровня.

```
<xsl:template mode="number" match="*">
   <xsl:number    level="multiple"
                  count="inform-div1 | div1 | div2 | div3 | div4"
                  format="1.1 "/>
</xsl:template>

<xsl:template mode="number" match="back//*">
   <xsl:number    level="multiple"
                  count="inform-div1 | div1 | div2 | div3 | div4"
                  format="A.1 "/>
</xsl:template>
```

Такое использование списка вариантов в атрибуте count – распространенный способ проведения многоуровневой нумерации. В действительности он заключается в выведении порядкового номера для каждого родительского элемента, которым является или <inform-div1>, или <div1>, или <div2>, и т. д. Подобно большинству шаблонных правил в таблице стилей, основанной на правилах, проверки здесь не производится: если исходная структура построена неправильно, шаблон все равно произведет какой-то вывод, а выяснять причины проблемы должен автор документа. Это влечет за собой интересный вопрос, который следует задавать себе при проектировании собственных таблиц стилей: должна ли таблица стилей обнаруживать ошибки в исходном документе и сообщать о них?

Создание заголовков разделов

Следующий раздел таблицы стилей предназначен для форматирования заголовков разделов. Общая логика здесь заключается в формировании номера раздела и создании текстового якоря (...) как цели гиперссылки, и для всех разделов эту работу выполняет общий именованный шаблон. Если раздел имеет атрибут id, этот атрибут используется как якорь; в других случаях якорь генерируется из заголовка раздела. Эта логика должна соответствовать логике, использованной ранее при создании гиперссылок в оглавлении.

Номера генерируются путем повторного использования шаблонного правила, показанного при создании оглавления.

```
<xsl:template match="div1/head | inform-div1/head">
   <h2><xsl:call-template name="head"/></h2>
</xsl:template>

<xsl:template match="div2/head">
   <h3><xsl:call-template name="head"/></h3>
</xsl:template>
<xsl:template match="div3/head">
   <h4><xsl:call-template name="head"/></h4>
</xsl:template>

<xsl:template match="div4/head">
   <h5><xsl:call-template name="head"/></h5>
</xsl:template>

<xsl:template name="head">
   <xsl:for-each select="..">
      <xsl:call-template name="insertID"/>
      <xsl:apply-templates select="." mode="number"/>
  </xsl:for-each>
   <xsl:apply-templates/>
   <xsl:call-template name="inform"/>
</xsl:template>

<xsl:template name="insertID">
   <xsl:choose>
      <xsl:when test="@id">
         <a name="{@id}"/>
      </xsl:when>
      <xsl:otherwise>
         <a name="section-{translate(head,' ','-')}"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Обратите внимание на использование `<xsl:for-each select="..">`. Это выражение не производит итерацию (узел имеет только одного родителя), оно здесь скорее для того, чтобы изменить текущий узел с элемента `<head>` на его родительский элемент `<divX>`, потому что именно здесь, как ожидается, должен быть вызван шаблон `insertID`.

Вызов шаблона `«inform»` внутри шаблона `«head»`, как и прежде, просто выводит для соответствующих приложений фразу «(Non-normative)».

Форматирование текста

Теперь рассмотрим ряд шаблонных правил для обработки обычной текстовой разметки в теле документа. Это довольно простые правила, хотя для лучшего понимания причин именно их использования стоит посмотреть на исходный документ и понять его структуру. Некоторые элементы, упомянутые здесь, отсутствуют в спецификации XML: вероятно, они существовали в ее более ранних рабочих проектах.

```
<xsl:template match="item/p" priority="1">
   <p><xsl:apply-templates/></p>
</xsl:template>
                <!-- этот шаблон, кажется, лишний. Вероятно, он когда-то отличался
от следующего шаблона -->
<xsl:template match="p">
   <p><xsl:apply-templates/></p>
</xsl:template>
```

Это на вид нормальное правило для обработки элементов <p> – фактически неправильное! Если посмотреть DTD для исходного XML-документа, можно заметить, что оно более либерально, чем позволяет спецификация HTML, в отношении вложенности списков в пределах абзацев. В результате эта таблица стилей может создать неправильный HTML из верного исходного документа. В более поздних версиях этой таблицы стилей, использованных для последующих стандартов W3C, эта ошибка исправлена – кое-что там в действительности требует решения довольно интересной проблемы группирования. Но здесь мы не будем поддаваться искушению исправлять это.

```
<xsl:template match="term">
   <b><xsl:apply-templates/></b>
</xsl:template>

<xsl:template match="code">
   <code><xsl:apply-templates/></code>
</xsl:template>

<xsl:template match="emph">
   <i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="blist">
   <dl><xsl:apply-templates/></dl>
</xsl:template>

<xsl:template match="slist">
   <ul><xsl:apply-templates/></ul>
</xsl:template>

<xsl:template match="sitem">
   <li><xsl:apply-templates/></li>
</xsl:template>

<xsl:template match="olist">
   <ol><xsl:apply-templates/></ol>
</xsl:template>

<xsl:template match="ulist">
   <ul><xsl:apply-templates/></ul>
</xsl:template>

<xsl:template match="glist">
   <dl><xsl:apply-templates/></dl>
</xsl:template>
```

```
<xsl:template match="olist">
   <ol><xsl:apply-templates/></ol>
</xsl:template>

<xsl:template match="item">
   <li><xsl:apply-templates/></li>
</xsl:template>

<xsl:template match="label">
   <dt><b><xsl:apply-templates/></b></dt>
</xsl:template>
<xsl:template match="def">
   <dd><xsl:apply-templates/></dd>
</xsl:template>

<xsl:template match="quote">
   <xsl:text>"</xsl:text>
   <xsl:apply-templates/>
   <xsl:text>"</xsl:text>
</xsl:template>

<!-- примеры -->

<xsl:template match="eg">
   <pre>
      <xsl:if test="@role='error'">
          <xsl:attribute name="style">color: red</xsl:attribute>
      </xsl:if>
      <xsl:apply-templates/>
   </pre>
</xsl:template>

<!-- таблицы общего назначения -->

<xsl:template match="htable">
   <table border="{@border}"
            cellpadding="{@cellpadding}"
            align="{@align}">
      <xsl:apply-templates/>
   </table>
</xsl:template>
```

Можно заметить, что «`border="{@border}"`» будет генерировать вывод «`border=""`», если у элемента в исходном документе нет атрибута `border`. Лучше было бы написать:

```
<table>
   <xsl:copy-of select="@align | @border | @cellpadding"/>
   <xsl:apply-templates/>
</table>
```

Продолжение таблицы стилей:

```
<xsl:template match="htbody">
   <tbody><xsl:apply-templates/></tbody>
```

```
</xsl:template>

<xsl:template match="tr">
   <tr align="{@align}" valign="{@valign}">
      <xsl:apply-templates/>
   </tr>
</xsl:template>
<xsl:template match="td">
   <td bgcolor="{@bgcolor}"
         rowspan="{@rowspan}"   colspan="{@colspan}"
         align="{@align}"   valign="{@valign}">
      <xsl:apply-templates/>
   </td>
</xsl:template>

<!-- замечания -->

<xsl:template match="ednote">
   <blockquote>
      <p><b>Ed. Note: </b><xsl:apply-templates/></p>
   </blockquote>
</xsl:template>

<xsl:template match="edtext">
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="issue">
   <xsl:call-template name="insertID"/>
   <blockquote>
      <p><b>Issue (<xsl:text/>
         <xsl:value-of   select="substring-after(@id,'-')"/>
         <xsl:text/>): </b>
         <xsl:apply-templates/>
      </p>
   </blockquote>
</xsl:template>

<xsl:template match="note">
   <blockquote>
      <b>NOTE: </b><xsl:apply-templates/>
   </blockquote>
</xsl:template>

<xsl:template match="issue/p | note/p">
   <xsl:apply-templates/>
</xsl:template>
```

Вывод порождающих правил

Теперь перейдем к более интересной области. Рекомендация XML содержит порождающие правила синтаксиса, и они определены довольно подробно. Последовательность порождающих правил содержится внутри элемента `<scrap>`, и каждое правило является элементом `<prod>`. Ниже приведен фраг-

мент элемента <scrap>, который содержит единственное порождающее правило:

```
<scrap lang='ebnf' id='document'>
   <head>Document</head>
   <prod id='NT-document'>
      <lhs>document</lhs>
      <rhs>
         <nt def='NT-prolog'>prolog</nt>
         <nt def='NT-element'>element</nt>
         <nt def='NT-Misc'>Misc</nt>*
      </rhs>
   </prod>
</scrap>
```

Это, конечно, порождающее правило XML, которое в спецификации приводится так:

Document

[1] document ::= prolog element Misc*

В некоторых случаях порождающие правила внутри элемента <scrap> сгруппированы в элементы <prodgroup>, но это группирование игнорируется при выводе.

Ниже приведены шаблонные правила высшего уровня:

```
<xsl:template match="scrap">
   <xsl:if test="string(head)">
      <h5><xsl:value-of select="head"/></h5>
   </xsl:if>
   <table class="scrap">
      <tbody>
         <xsl:apply-templates select="prodgroup | prod"/>
      </tbody>
   </table>
</xsl:template>

<xsl:template match="prodgroup">
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="prod">
  <!-- выбор элементов, начинающих строку -->
   <xsl:apply-templates
      select="*[self::lhs
                or (self::rhs
                    and not(preceding-sibling::*[1][self::lhs]))
                  or ((self::vc or self::wfc or self::com)
                    and not(preceding-sibling::*[1][self::rhs]))]
            "/>
</xsl:template>
```

Что делает это страшное выражение select? Порождающее правило (<prod>) имеет одну левую сторону (<lhs>), одну или более правых сторон (<rhs>) и одну или более аннотаций (<vc>, <wfc> или <com>). Элемент <vc> используется для указания ограничения действительности, элемент <wfc> – для указания ограничений корректности и элемент <com> – для ссылки на комментарии. Правило с одним элементом <lhs>, двумя элементами <rhs> и тремя аннотациями <wfc> будет размещено в HTML-таблице следующим образом:

[17]	lhs1	::=	rhs1	
			rhs2	wfc1
				wfc2
				wfc3

Как говорится в комментарии, выражение select обрабатывает непосредственных потомков элемента <prod>, которые начинают новую строку: здесь это – lhs1, rhs2, wfc2 и wfc3. Если точнее, выбранные элементы включают любые элементы <lhs>, любые элементы <rhs>, которым непосредственно не предшествует элемент <lhs>, и любые элементы <vc>, <wfc> или <com>, которым непосредственно не предшествует элемент <rhs>. Таким образом, этот шаблон выбирает элементы, которые начинают новую строку, и вызывает <xsl:apply-templates> для их обработки.

Теперь рассмотрим шаблонные правила, которые соответствуют этим элементам. Сначала <lhs>:

```
<xsl:template match="lhs">
   <tr valign="baseline">
   <td><a name="{../@id}"/>
      <xsl:number from="body" level="any"
                  format="[1]   "/>
   </td>
   <td><xsl:apply-templates/></td>
   <td>
      <xsl:text>   ::=   </xsl:text>
   </td>
   <xsl:for-each select="following-sibling::*[1]">
      <td><xsl:apply-templates mode="cell" select="."/></td>
      <td><xsl:apply-templates mode="cell"
            select="following-sibling::*[1]
                    [self::vc or self::wfc or self::com]"/>
      </td>
   </xsl:for-each>
   </tr>
</xsl:template>
```

Вызов <xsl:number> здесь – это хороший пример того, как генерировать последовательность номеров по всему документу. В данном случае создается последовательный номер для каждого элемента <lhs>, то есть для каждого порождающего правила.

После этого шаблон вызывает <xsl:apply-templates/> для обработки содержимого элемента <lhs>, которым обычно будет просто название определяемого синтаксического терма, и выводит символы « ::= », отделяющие терм от его определения.

Следующий затем элемент <xsl:for-each> не производит никакой итерации, он просто перемещает текущую позицию к следующему одноуровневому элементу. Дело в том, что выражение select с предикатом «[1]» может выбрать только один узел.

Этим узлом будет первый элемент <rhs>. Для его обработки вызывается <xsl:apply-templates> в режиме «mode="cell"»; а затем, если следующим элементом является элемент <vc>, <wfc> или <com>, он также обрабатывается в режиме «mode="cell"». Эта логика должна соответствовать логике, использованной ранее при решении, какие элементы должны начинать новую строку: на этой стадии должны обрабатываться только те элементы, которые находятся в одной строке с элементом <lhs>.

Следующие два шаблонных правила используются, соответственно, для элементов <rhs>, которые начинают новую строку, и для элементов <vc>, <wfc> или <com>, которые также начинают новую строку: то есть для rhs2, wfc2 и wfc3 в данном примере. В каждом случае для правильного горизонтального выравнивания шаблон должен генерировать ряд пустых ячеек таблицы.

```
<xsl:template match="rhs">
   <tr valign="baseline">
      <td></td>
      <td></td>
      <td></td>
      <td><xsl:apply-templates mode="cell" select="."/></td>
      <td><xsl:apply-templates mode="cell"
            select="following-sibling::*[1]
                     [self::vc or self::wfc or self::com]"/>
      </td>
   </tr>
</xsl:template>

<xsl:template match="vc | wfc | com">
   <tr valign="baseline">
      <td></td>
      <td></td>
      <td></td>
      <td></td>
      <td><xsl:apply-templates mode="cell" select="."/></td>
   </tr>
</xsl:template>
```

Некоторые предпочитают избегать пустых ячеек таблицы, добавляя «<td> </td>», но это действительно необходимо, если только таблица имеет границы или цветной фон.

Правило для <rhs> следует той же логике, что и правило для <lhs>: обрабатываются любые следующие за <rhs> элементы <vc>, <wfc> или <com>, которые принадлежат той же строке.

Теперь нужно написать шаблонные правила для элементов, которые входят в ту же строку, что и предыдущий элемент, в нашем примере это rhs1 и wfc1. Они всегда будут вызываться в режиме «mode="cell"». Первые два правила очень просты, потому что окружающие элементы <td> уже сгенерированы.

```
<xsl:template match="rhs" mode="cell">
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="com" mode="cell">
   <xsl:text>/*</xsl:text>
   <xsl:apply-templates/>
   <xsl:text>*/</xsl:text>
</xsl:template>
```

В случае элементов <vc> и <wfc> для генерирования гиперссылки на абзац, описывающий ограничение действительности или ограничение корректности, необходима определенная логика. Ссылка в XML представляется атрибутом def, и он используется непосредственно для создания внутренней гиперссылки HTML. Отображаемый текст ссылки формируется путем отыскания элемента, ID которого соответствует этому атрибуту def, и отображения его текста.

Например, если элемент <vc> имеет вид <vc def="vc-roottype"/>, то он указывает на следующий элемент в исходном файле:

```
<vcnote id="vc-roottype">
    <head>Root Element Type</head>
    <p>The Name in the document type declaration must match the element type
       of the root element.</p>
</vcnote>
```

тогда будет сгенерирован следующий HTML-код:

```
<a href="#vc-roottype">Root Element Type</a>
```

Вернемся к таблице стилей:

```
<xsl:template match="vc" mode="cell">
   <xsl:text>[ VC: </xsl:text>
   <a href="#{@def}">
      <xsl:value-of select="id(@def)/head"/>
   </a>
   <xsl:text> ]</xsl:text>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="wfc" mode="cell">
   <xsl:text>[ WFC: </xsl:text>
   <a href="#{@def}">
```

```
      <xsl:value-of select="id(@def)/head"/>
    </a>
    <xsl:text> ]</xsl:text>
    <xsl:apply-templates/>
</xsl:template>
```

Так много требуется для форматирования порождающих правил! Это была наиболее сложная часть таблицы стилей, остальное должно быть просто.

Создание перекрестных ссылок

Следующий раздел таблицы стилей предназначен для форматирования перекрестных ссылок. Сначала рассмотрим шаблонные правила для создания якорей, на которые могут указывать ссылки, – все они генерируют элементы вида `<a name="xyz">...</a>`. Они используются для библиографических ссылок, определений терминов и описания ограничений действительности и корректности.

```
<xsl:template match="blist/bibl">
   <dt>
      <a name="{@id}"><xsl:value-of select="@key"/></a>
   </dt>
   <dd>
      <xsl:apply-templates/>
   </dd>
</xsl:template>
<xsl:template match="termdef">
   <a name="{@id}"/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="vcnote">
   <a name="{@id}"/>
   <p><b>Validity Constraint: <xsl:text/>
      <xsl:value-of select="head"/></b></p>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="wfcnote">
   <a name="{@id}"></a>
   <p><b>Well Formedness Constraint: <xsl:text/>
      <xsl:value-of   select="head"/></b></p>
   <xsl:apply-templates/>
</xsl:template>
```

А дальше идут шаблонные правила, которые генерируют гиперссылки. Все они генерируют элемент в виде `<a href="xyz">...</a>` для внешней ссылки или в виде `<a href="#xyz">...</a>` – для внутренней ссылки.

```
<!-- внешние ссылки -->

<xsl:template match="p/loc" priority="1">
   <a href="{@href}"><xsl:apply-templates/></a>
</xsl:template>
```

```
<xsl:template match="loc">
   <a href="{@href}"><xsl:apply-templates/></a>
</xsl:template>
```

Атрибут `priority` в первом шаблонном правиле необязателен: по умолчанию правило автоматически получило бы более высокий приоритет, чем следующее за ним правило. Здесь он тем более не нужен, потому что оба правила идентичны! Однако некоторые предпочитают в таких случаях расставлять приоритеты явно, для облегчения чтения, если нет иной причины.

```
<xsl:template match="xspecref | xtermref">
   <a href="{@href}"><xsl:apply-templates/></a>
</xsl:template>

<xsl:template match="xnt">
   <a href="{@href}"><xsl:apply-templates/></a>
</xsl:template>

<!-- внутренние перекрестные ссылки -->

<xsl:template match="titleref">
   <a href="#{@href}"><xsl:apply-templates/>   </a>
</xsl:template>

<xsl:template match="nt">
   <a href="#{@def}"><xsl:apply-templates/></a>
</xsl:template>

<xsl:template match="termref">
   <a href="#{@def}"><xsl:apply-templates/></a>
</xsl:template>
<xsl:template match="bibref">
   <a href="#{@ref}">
      <xsl:text>[</xsl:text>
      <xsl:value-of select="id(@ref)/@key"/>
      <xsl:apply-templates/>
      <xsl:text>]</xsl:text>
   </a>
</xsl:template>

<xsl:template match="specref">
   <a href="#{@ref}">
      <xsl:text>[</xsl:text>
      <b>
         <!-- Добавляем номер раздела и заголовок цели -->
         <xsl:for-each select="id(@ref)/head">
         <xsl:apply-templates select=".." mode="number"/>
         <xsl:apply-templates/>
         </xsl:for-each>
      </b>
      <xsl:apply-templates/>
      <xsl:text>]</xsl:text>
   </a>
</xsl:template>
```

Здесь нет ничего сложного. Последнее шаблонное правило чуть посложнее, чем другие: оно создает ссылку на раздел, включающую номер раздела, который оно получает, вызывая <xsl:apply-templates>, чтобы обработать элемент, на который делается ссылка, в режиме «mode="number"».

Отфильтровывание ненужного

Теперь перейдем к шаблонным правилам для элементов исходного XML-документа, которые вообще не нужно включать в вывод HTML. Пустой элемент <xsl:template> – это холостая команда «нет операции», которая означает, что для элементов, соответствующих этому шаблонному правилу, не должен производиться никакой вывод. Если бы эти шаблонные правила не были включены, эти элементы были бы обработаны встроенным шаблонным правилом, которые вывело бы текст элемента без всякой разметки.

```
<xsl:template match="w3c-designation"/>
<xsl:template match="w3c-doctype"/>
<xsl:template match="header/pubdate"/>
<xsl:template match="spec/header/title"/>
<xsl:template match="revisiondesc"/>
<xsl:template match="pubstmt"/>
<xsl:template match="sourcedesc"/>
<xsl:template match="langusage"/>
<xsl:template match="version"/>
```

Эти элементы являются, главным образом, метаданными, представляющими интерес для авторов, но не для читателей. (На самом деле хронология редактирования весьма занятна для тех, кто интересуется вещами такого рода. Она очень явно показывает, что использование таблицы стилей для отфильтровывания конфиденциальной информации – далеко не самый эффективный способ обеспечения безопасности.)

Заготовки текстов

Теперь рассмотрим шаблон, который генерирует готовые тексты:

```
<xsl:template name="copyright">
   <xsl:variable name="legal"
         select="'http://www.w3.org/Consortium/Legal/'"/>
   <p class="copyright">
      <a href="{$legal}ipr-notice.html#Copyright">Copyright</a>
         &amp;nbsp;&amp;copy;&amp;nbsp; 1999
       <a href="http://www.w3.org">W3C</a>
      (<a href="http://www.lcs.mit.edu">MIT</a>,
      <a href="http://www.inria.fr/">INRIA</a>,
      <a href="http://www.keio.ac.jp/">Keio</a>),
         All Rights Reserved. W3C
      <a href="{$legal}ipr-notice.html#Legal_Disclaimer"
         >liability</a>,
      <a href="{$legal}ipr-notice.html#W3C_Trademarks"
```

```
            >trademark</a>,
         <a href="{$legal}copyright-documents.html">document use</a>
            and
         <a href="{$legal}copyright-software.html"
            >software licensing</a> rules apply.
      </p>
   </xsl:template>
```

Список составителей

Последние четыре шаблонных правила форматируют список составителей, приводимый в конце документа. Элементы «orglist/member» соответствуют двум правилам, в результате действия которых перед именем каждого составителя, кроме первого, ставится точка с запятой. На этот раз атрибут `priority` необходим, потому что по умолчанию оба правила получили бы равные приоритеты.

```
<xsl:template match="orglist">
   <xsl:apply-templates select="*"/>
</xsl:template>

<xsl:template match="orglist/member[1]" priority="2">
   <xsl:apply-templates select="*"/>
</xsl:template>

<xsl:template match="orglist/member">
   <xsl:text>; </xsl:text>
   <xsl:apply-templates select="*"/>
</xsl:template>

<xsl:template match="orglist/member/affiliation">
   <xsl:text>, </xsl:text>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="orglist/member/role">
   <xsl:text> (</xsl:text>
   <xsl:apply-templates/>
   <xsl:text>)</xsl:text>
</xsl:template>
```

И этим заканчивается таблица стилей:

```
</xsl:stylesheet>
```

Варианты таблиц стилей для спецификаций XSLT и XPath

Представленная выше таблица стилей применяется для спецификации XML. Таблицы стилей, используемые для спецификаций XPath и XSLT, несколько отличаются от нее, потому что в этих документах встречаются дополнительные типы элементов, которых нет в спецификации XML. В каж-

дом случае исходный XML-документ имеет внутреннее подмножество DTD, которое дополняет базовое DTD некоторыми дополнительными типами элементов. Например, в документе XPath использованы специальные теги для разметки шаблонов функций, а в XSLT-документе – специальные теги для разметки проформ, используемых для итогового описания синтаксиса каждого XSL-элемента.

Поэтому таблицы стилей, используемые для XPath и XSLT, состоят из небольшого модуля, импортирующего с помощью `<xsl:import>` базовую таблицу стилей XML и определяющего дополнительные правила, необходимые для дополнительных конструкций. Все три таблицы стилей включены в файлы загрузки для этой главы.

Резюме

Примером, представленным здесь, послужила реальная таблица стилей, используемая для реального приложения, а не в образовательных целях. Возможно, она слегка нетипична, поскольку многое в ней было написано еще до завершения работы над спецификацией XSLT, в результате чего некоторые вещи в ней делаются окольными путями. Однако она, вероятно, не так уж отличается от многих других таблиц стилей, используемых в приложениях для форматирования документов.

Фраза «форматирование документов» является ключевой. Основными задачами этой таблицы стилей были применение стилей отображения HTML к различным элементам, формирование гиперссылок и форматирование таблиц. Все это – задачи, с которыми справляются образцы проектирования, основанные на правилах.

Следующий пример – это совершенно иное приложение: в нем используются высокоструктурированные данные, которые отображаются совершенно иначе, чем они представлены в исходном документе.

Генеалогическое дерево

XML часто используется для представления информации, являющейся смесью структурированных данных и текста. Вместо простого примера структурированных данных (типа книжных каталогов, которые обсуждаются во множестве примеров XML) выберем для обработки нечто более сложное – генеалогическое дерево.

Можно было бы взять также пример со счетами, заявками и заказами на товары. Методы, используемые в этом работающем примере, также применимы и ко многим практическим коммерческим задачам, но автор полагает, что небольшой экскурс в мир генеалогии читатели сочтут приятной разгрузкой после рабочего дня.

Тем не менее, следует иметь в виду один нюанс. Везде в этой книге говорится о древовидных моделях в XML, и в контексте этих информационных деревь-

ев используются такие термины, как родитель и дочерний элемент, предок и потомок. Не думайте, однако, что эту древовидную структуру можно непосредственно использовать для представления генеалогического дерева. Фактически генеалогическое дерево – совсем не дерево, потому что каждый человек имеет двух родителей, а в деревьях XML считается достаточным наличие одного родителя.

Структура генеалогического дерева на самом деле довольно сильно отличается от дерева документа, используемого для его представления. В данном разделе слова: «родитель» и «ребенок», «предок» и «потомок» – имеют их каждодневное значение!

Модель данных и ее представление в XML

Признанный стандарт для представления генеалогических данных известен как GEDCOM, и данными в этом формате обычно обмениваются различные программы, и данные в этом формате передаются по Интернету. Основными объектами, содержащимися в файле GEDCOM, являются записи, представляющие индивидуальных личностей (называемые записями INDI), и записи, представляющие пары (называемые записями FAM).

Запись INDI выглядит так:

```
0 @I1@ INDI
1 NAME John Fitzgerald/Kennedy/
1 SEX M
1 BIRT
2 DATE 29 MAY 1917
2 PLAC Brookline, MA, USA
1 DEAT
2 DATE 22 NOV 1963
2 PLAC Dallas, TX, USA
2 NOTE Assassinated by Lee Harvey Oswald.
1 NOTE Educated at Harvard University.
2 CONT Elected Congressman in 1945
2 CONT aged 29; served three terms in the House of Representatives.
2 CONT Elected Senator in 1952. Elected President in 1960, the
2 CONT youngest ever President of the United States.
1 FAMS @F1@
1 FAMC @F2@
```

Это, конечно, не XML, но можно преобразовать эту запись в XML, чтобы она выглядела следующим образом:

```
<INDI ID="I1">
   <NAME>John Fitzgerald<S>Kennedy</S></NAME>
   <SEX>M</SEX>
   <BIRT>
      <DATE>29 MAY 1917</DATE>
      <PLAC>Brookline, MA, USA</PLAC>
   </BIRT>
```

```
    <DEAT>
      <DATE>22 NOV 1963</DATE>
      <PLAC>Dallas, TX, USA</PLAC>
      <NOTE>Assassinated by Lee Harvey Oswald.<BR/></NOTE>
    </DEAT>
    <NOTE>Educated at Harvard University.<BR/>
Elected Congressman in 1945<BR/>
aged 29; served three terms in the House of Representatives.<BR/>
Elected Senator in 1952. Elected President in 1960, the<BR/>
youngest ever President of the United States.<BR/>
    </NOTE>
    <FAMS REF="F1"/>
    <FAMC REF="F2"/>
</INDI>
```

Позже (на стр. 732) будет показано, как фактически выполняется это преобразование в XML.

Каждая запись в файле GEDCOM имеет уникальный идентификатор (в данном случае `I1` – это буква I и цифра один), который используется для создания перекрестных ссылок между записями. Большая часть информации в этих записях очевидна, кроме полей `<FAMS>` и `<FAMC>`: `<FAMS>` – это ссылка на запись `<FAM>`, представляющую семью, в которой этот человек является родителем, а `<FAMC>` – ссылка на семью, в которой этот человек является ребенком.

Обратите внимание, каким образом могут быть вложены поля, чтобы показать структуру. Спецификация GEDCOM (разработанная церковью Иисуса Христа Святых последних дней – Church of Jesus Christ of Latter-day Saints) определяет строгую схему вложения тегов, что очень похоже на DTD в XML.

> *HTML-версию спецификации GEDCOM можно найти по адресу http://www.gendex.com/gedcom55/55gcint.htm.*

Однако хотя схема и строга, она все же позволяет вводить разнообразную информацию. Например, любое количество событий или характеристик, касающихся личности, с полными или частичными датами, текстовыми замечаниями на любом уровне и ссылками на источники информации. Примерами событий могут быть рождение, смерть, усыновление или выход в отставку; примерами характеристик – род занятий, вероисповедание или здоровье; но это не окончательный список.

Запись `<INDI>` содержит сведения о событиях и характеристиках отдельных людей, в то время как запись `<FAM>` определяет сведения о парах и их семьях («пара» здесь представляет двух людей, которые или состояли в браке, или имели детей, или и то и другое; кроме того, в этих записях отражены и те случаи, когда партнер неизвестен или незафиксирован). Запись `<FAM>`, подобно записи `<INDI>`, имеет произвольный уникальный идентификатор. После преобразования в XML она может выглядеть следующим образом:

```
<FAM ID="F1">
    <HUSB REF="I1"/>
    <WIFE REF="I2"/>
```

```
    <CHIL REF="I5"/>
    <CHIL REF="I6"/>
    <CHIL REF="I7"/>
    <MARR>
       <DATE>12 SEP 1953</DATE>
       <PLAC>Newport, RI, USA</PLAC>
    </MARR>
</FAM>
```

Элемент <FAM> содержит ссылки на все личности, составляющие семью. Формально эти ссылки избыточны – в том смысле, что они могли бы быть выявлены по ссылкам в противоположном направлении; но избыточность – это не всегда плохо. Элемент содержит также информацию о событиях, касающихся обоих партнеров, наиболее типичным из таких событий является их вступление в брак.

Лучше понять это поможет схема, показывающая, как представлены взаимосвязи в семье.

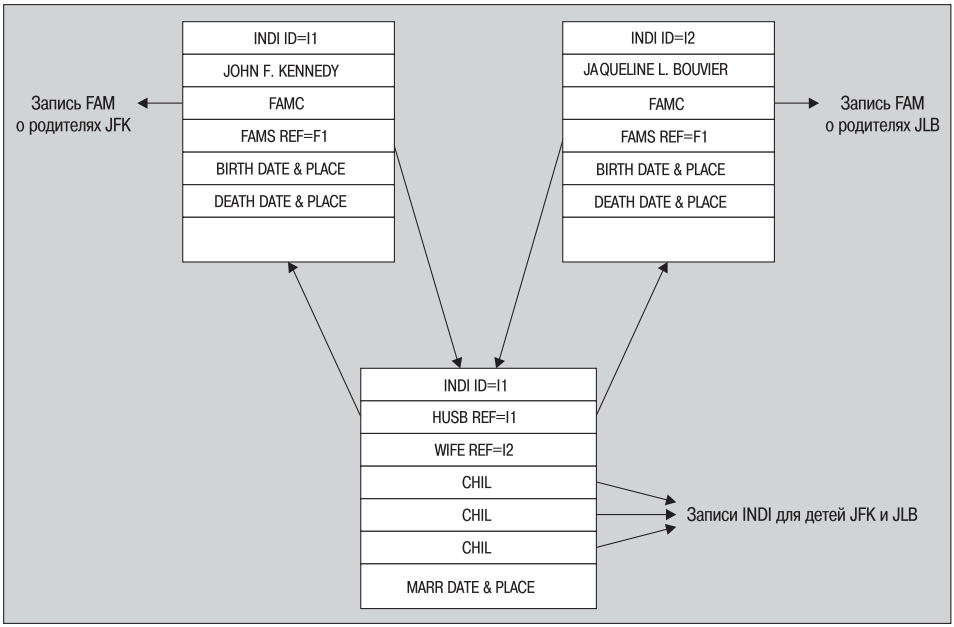


Рис. 10.2. *Представление взаимосвязей между членами семьи*

Не будем обсуждать здесь, насколько хорош или плох такой способ представления генеалогической информации. Многие критиковали эту модель данных – по техническим соображениям или по соображениям политкорректности, но подобно клавиатуре QWERTY, модель GEDCOM сохраняется, несмотря на ее недостатки, просто потому, что очень многие ее используют.

В модели GEDCOM нет формального способа связать один файл с другим. XML, конечно, дает замечательные возможности определить, как одно гене-

алогическое дерево связывается с чьим-то еще. Но это связывание осуществить не так просто, как может показаться (как и все в генеалогии), из-за проблем сохранения целостности версий между двумя наборами данных, которые изменяются независимо. По этой причине не будем здесь вдаваться в эту область, а рассмотрим случай, когда все генеалогическое дерево находится в одном XML-документе.

Отображение данных генеалогического дерева

Пусть нам нужно написать таблицу стилей, которая отображает данные файла GEDCOM в формате HTML. Предположим сейчас, что преобразование к синтаксису XML уже произведено (как это делается, обсудим позже на стр. 732). Мы хотим получить результат, подобный представленному на рис. 10.3 снимку экрана.

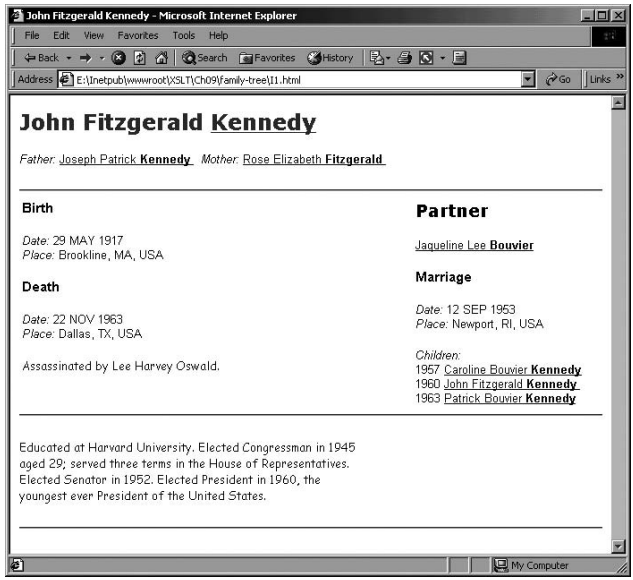


***Рис. 10.3.** Отображение данных генеалогического дерева в окне броузера*

Здесь приведены все сведения об одной личности и ссылки на родственные личности, благодаря которым можно просматривать генеалогическое дерево. Конечно, можно было отобразить эти данные более претенциозным способом, и автор оставляет эту попытку читателям: можно начать с небольшого набора данных о Кеннеди, включенного в файлы загрузки для этой книги, а затем продолжить с любым другим набором данных GEDCOM, возможно, с данными собственного генеалогического дерева.

Поскольку для каждой личности из файла требуется создать отдельную HTML-страницу, нужно подумать, как создать несколько HTML-страниц из одного исходного XML-документа. Для этого существует, по крайней мере, три способа:

- Процесс массового опубликования, при котором исходный XML-документ преобразуется в набор HTML-страниц, а затем публикуется на вебсервере в виде статических страниц. Преимущество этого способа в том, что преобразование приходится выполнять только один раз. Это минимизирует зависимость от средств, доступных у Интернет-провайдера, и работает с любым броузером.
- Генерирование HTML-страниц на сервере по запросу при помощи Java-сервлетов или ASP-страниц. Это тоже будет работать с любым броузером, но на этот раз придется найти Интернет-провайдер, разрешающего выполнять сервлеты или ASP-страницы.
- Загрузка клиентом всего XML-файла и генерирование представления там. Преимущество этого способа в том, что данные загружаются только однажды, и пользователь может затем просматривать их не спеша без дальнейшего взаимодействия с сервером. Однако для этого нужен броузер, поддерживающий XML, что сейчас означает Internet Explorer 5 или выше.

  Броузер Netscape 6 / Mozilla поддерживает XML и CSS, но поддержка XSLT (с использованием процессора Transformiix) пока еще находится в стадии разработки. Следите за новостями на http://www.mozilla.org/projects/xslt/.

  Другой недостаток – слабая защищенность: нет никакого способа фильтрации данных, например для удаления сведений о живых людях, и нет также никакого способа защитить весь файл XML от копирования пользователями (пользователь может, например, использовать опцию View Source или может найти данные в кэше броузера).

Единственное реальное различие между этими тремя способами, которое касается таблицы стилей, заключатся в том, что по-разному генерируются гиперссылки.

С этими различиями можно справиться, написав универсальный модуль таблицы стилей, содержащий общий для этих трех случаев код. Затем этот модуль можно импортировать в таблицы стилей, которые обрабатывают различия для каждого случая.

Таблица стилей

Сначала напишем таблицу стилей, `person.xsl`, которая генерирует HTML-страницу, отображающую сведения, относящиеся к отдельной личности. Эта таблица стилей должна принимать `ID` заданной личности как глобальный параметр. Если значение не задано, выберем первую запись `<INDI>` в файле. Ниже приведена структура верхнего уровня:

```
<xsl:transform
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0" >

<xsl:param name="id" select="/*/INDI[1]/@ID"/>

<!-- определяем ключи для поиска записей по идентификаторам -->
```

```
<xsl:key name="indi" match="INDI" use="@ID"/>
<xsl:key name="fam" match="FAM" use="@ID"/>
<xsl:template match="/">
   <xsl:variable name="person" select="key('indi', $id)"/>
   <xsl:apply-templates select="$person"/>
</xsl:template>

. . .

</xsl:transform>
```

Для непосредственного доступа к элементам <INDI> и <FAM>, когда известны их атрибуты ID, определены два ключа. Можно было бы полагаться на функцию id(), но это возможно лишь в случае, когда атрибут ID определен в DTD как являющийся атрибутом типа ID. Это делает таблицу стилей зависимой от наличия DTD для документа, поэтому более надежным все-таки кажется использование функции key().

Затем начальный шаблон (который соответствует корневому узлу) вызывает <xsl:apply-templates> для обработки элемента <INDI>, представляющего выбранную личность.

Шаблонное правило для элемента <INDI> создает каркас HTML-страницы:

```
<xsl:template match="INDI">
   <html>
      <head>
         <xsl:call-template name="css-style"/>
         <xsl:variable name="name">
            <xsl:apply-templates select="NAME"/>
         </xsl:variable>
         <title><xsl:value-of select="$name"/></title>
      </head>

      <!-- выбираем цвет фона в зависимости от пола -->

      <xsl:variable name="color">
         <xsl:choose>
            <xsl:when test="SEX='M'">cyan</xsl:when>
            <xsl:otherwise>pink</xsl:otherwise>
         </xsl:choose>
      </xsl:variable>

      <body bgcolor="{$color}">

         <!-- Выводим имя и информацию о родителях -->

         <h1><xsl:apply-templates select="NAME"/></h1>
         <xsl:call-template name="show-parents"/>
         <hr/>

         <table>
         <tr>
            <!-- Выводим события и характеристики -->

            <td width="50%" valign="top">
```

```
            <xsl:call-template name="show-events"/>
         </td>
         <td width="20%"/>

         <!-- Выводим имена родителей, детей и информацию о браках -->

         <td width="30%" valign="top">
            <xsl:call-template name="show-partners"/>
         </td>
      </tr>
      </table>

      <hr/>

      <!-- Выводим примечания -->

      <xsl:for-each select="NOTE">
         <p class="text"><xsl:apply-templates/></p>
         <xsl:if test="position()=last()"><hr/></xsl:if>
      </xsl:for-each>

   </body>
</html>
</xsl:template>
```

Это шаблонное правило работает в процессе формирования вывода страницы. Некоторые наблюдения:

- Название в заголовке HTML генерируется путем создания переменной, а затем – копирования значения переменной в элемент `<title>`. Это делается намеренно, чтобы использовать стандартное шаблонное правило для формирования имени человека, а инструкция `<xsl:value-of>` затем удаляет теги, подобные `<b>`, которые появляются в сформированном имени, потому что они мешают отображению названия в некоторых броузерах.
- Цвет фона страницы зависит от значения атрибута `SEX` данной личности. Это может выглядеть несколько по-детски, и читатель вправе изменить это, но для книги это полезно, так как демонстрирует еще одну методику XSLT.
- Основная задача формирования содержимого страницы разделяется между несколькими отдельными именованными шаблонами, просто для обеспечения модульности.
- Здесь мы не будем пытаться отображать все данные, которые позволяет включать или на которые позволяет ссылаться GEDCOM в записях `<INDI>`: цитаты источников, объекты мультимедиа – фотографии и т. п. Любые такие данные будут просто пропущены.

Для указания размеров шрифта и тому подобного используется внутренняя таблица стилей CSS, а задача генерирования этой таблицы стилей передается шаблону под названием `css-style`. В результате будет сгенерирован следующий вывод:

```
<xsl:template name="css-style">
   <style type="text/css">
```

```
      H1 {
         font-family: Verdana, Helvetica, sans-serif;
         font-size: 18pt;
         font-weight: bold;
         color: "#FF0080"
      }

      H2 {
         font-family: Verdana, Helvetica, sans-serif;
         font-size: 14pt;
         font-weight: bold;
         color: black;
      }

      H3 {
         font-family: Lucida Sans, Helvetica, sans-serif;
         font-size: 11pt;
         font-weight: bold;
         color: black;
      }

      SPAN.label {
         font-family: Lucida Sans, Helvetica, sans-serif;
         font-size: 10pt;
         font-weight: normal;
         font-style: italic;
         color: black;
      }

      P,LI,TD {
         font-family: Lucida Sans, Helvetica, sans-serif;
         font-size: 10pt;
         font-weight: normal;
         color: black;
      }

      P.text {
         font-family: Comic Sans MS, Helvetica, sans-serif;
         font-size: 10pt;
         font-weight: normal;
         color: black;
      }

      </style>
   </xsl:template>
```

Вполне можно было бы, конечно, добавить эти атрибуты к различным элементам HTML по отдельности или включать их, используя наборы атрибутов XSLT, но данный способ кажется более ясным и показывает, как XSLT и CSS могут дополнять друг друга. Фактически даже лучше было бы использовать внешнюю таблицу стилей CSS, так как в этом случае пользователь, просматривающий многие из этих HTML-страниц, получил бы преимущество от кэширования.

Следующий шаблон отображает родителей текущей личности в виде гиперссылок:

```
<xsl:template name="show-parents">

   <xsl:variable name="parents" select="key('fam', FAMC/@REF)"/>
   <xsl:variable name="father" select="key('indi', $parents/HUSB/@REF)"/>
   <xsl:variable name="mother" select="key('indi', $parents/WIFE/@REF)"/>

   <p>
   <xsl:if test="$father">
      <span class="label">Father: </span>
      <xsl:apply-templates select="$father/NAME" mode="link"/> 
   </xsl:if>
   <xsl:if test="$mother">
      <span class="label">Mother: </span>
      <xsl:apply-templates select="$mother/NAME" mode="link"/> 
   </xsl:if>
   </p>
</xsl:template>
```

Шаблон начинается с определения местонахождения записи <FAM>, которая упомянута в поле <FAMC> текущей записи <INDI>. Он делает это с помощью ключа «fam», определенного ранее. Затем, используя на этот раз ключ «indi», он выбирает записи <INDI> отца и матери, которые указаны в полях <HUSB> и <WIFE> записи <FAM>.

Если каких-то данных нет, например, если нет поля <FAMC>, или если в записи <FAM> нет полей <HUSB> или <WIFE> (никакая родословная не безгранична), то переменные «$father» или «$mother» просто будут идентифицировать пустой набор узлов. Последовательность инструкций <xsl:if> гарантирует, что в этом случае соответствующая метка исключается из вывода.

Фактические гиперссылки формируются с помощью <xsl:apply-templates> в режиме «mode="link"». Это можно многократно использовать для всех других ссылок на странице, и позже будет показано, как это работает. Ссылка на символ « » выводит неразрывный пробел. Фактически это сделать проще, чем вывести обычный пробел, для которого потребовался бы элемент <xsl:text>. Если не нравится вводить числовые ссылки на символы, можно определить в объявлении <!DOCTYPE> сущность «nbsp» и затем использовать « » вместо « ».

Следующий именованный шаблон используется для отображения перечня событий для личности, таких как рождение и смерть. Позже он будет также использован для отображения общих событий пары (вступление в брак, развод).

```
<xsl:template name="show-events">
   <xsl:for-each select="*">
      <xsl:sort select="substring(DATE, string-length(DATE) - 3)"/>

      <xsl:variable name="event-name">
         <xsl:apply-templates select="." mode="expand"/>
```

```
        </xsl:variable>

        <xsl:if test="string($event-name)">
           <h3><xsl:value-of select="$event-name"/></h3>
           <p>
           <xsl:if test="DATE">
              <span class="label">Date: </span>
              <xsl:value-of select="DATE"/><br/>
           </xsl:if>
           <xsl:if test="PLAC">
              <span class="label">Place: </span>
              <xsl:value-of select="PLAC"/><br/>
           </xsl:if>
           </p>
           <xsl:for-each select="NOTE">
              <p class="text"><xsl:apply-templates/></p>
           </xsl:for-each>
        </xsl:if>
     </xsl:for-each>
  </xsl:template>
```

События здесь представлены в хронологическом порядке. Хотя логика `<xsl:sort>`, нацеленная на извлечение последних четырех символов поля `<DATE>`, чересчур прагматична, но в большинстве случаев позволяет извлечь год события. Простые даты в GEDCOM приводятся в формате «21 APR 1862». Прагматический подход работает даже в тех случаях, где дата имеет форму «BETWEEN 1865 AND 1868». Правда для даты в форме «55 BC» (55 год до н. э.) он не срабатывает, но, к счастью, данные Кеннеди не идут так глубоко в историю. Неплохо было бы при сортировке принимать во внимание месяц и день (довольно часто два события происходят в одном и том же году, например смерть и захоронение). Лучшим способом для этого, вероятно, является использование функции расширения, которая преобразовывает дату GEDCOM в формат ISO, то есть к виду ггггммдд.

Не так очевидно, какие элементы в исходных данных касаются событий. Стандарт GEDCOM определяет множество возможных событий, и вряд ли стоит использовать выражение объединения в форме «select="BIRT | DEAT | BAPM | BURI"», которое перечислит их все. Кроме того, необходима некоторая трансляция сокращенных названий тегов (типа BURI) в более понятные названия или описания событий, например «Burial» («Захоронение»). Можно убить сразу двух зайцев, применив к элементу другой шаблон, в режиме «expand»: если элемент признается как событие, этот шаблон выдаст понятное название этого события, а в противном случае выдаст пустую строку. Тогда обрабатывать нужно будет только те элементы, для которых значение возвращаемой переменной $event-name – не равно пустой строке. Это один из немногих случаев, когда преобразование типов должно быть явным. Если просто написать <xsl:if test="$event-name"/>, то дерево было бы преобразовано в логический тип данных, а в этом случае результат всегда истинен. Здесь нужно проверять строковое значение, поэтому используется функция string(), чтобы конвертировать его явным образом.

Итак, нужен длинный список шаблонов, чтобы развернуть теги различных типов событий. Здесь не будут приводиться все они, а только используемые наиболее часто:

```
<xsl:template match="BIRT" mode="expand">Birth</xsl:template>
<xsl:template match="DEAT" mode="expand">Death</xsl:template>
<xsl:template match="BURI" mode="expand">Burial</xsl:template>
<xsl:template match="BAPM" mode="expand">Baptism</xsl:template>
<xsl:template match="MARR" mode="expand">Marriage</xsl:template>
<xsl:template match="EVEN" mode="expand">
   <xsl:value-of select="TYPE"/>
</xsl:template>
<xsl:template match="*" mode="expand"/>
```

Шаблонное правило для элементов <EVEN> отбирает общие события GEDCOM, имеющие дочерние элементы <TYPE>, в которых даны описания этих событий. Это, как правило, такие исключительные случаи, как, например, крупный выигрыш в национальной лотерее, – для которых в стандарте GEDCOM не определено специальных тегов. А заключительное правило гарантирует, что несобытийные элементы выдадут пустую строку и, таким образом, не будут рассматриваться как события.

Осталось вывести последнюю часть HTML – правую панель, где собрана информация относительно супруга(ов) данной личности и ее браках. При наличии у кого-то нескольких супругов и браков его элемент <INDI> будет иметь несколько полей <FAMS>. В этом случае используются заголовки «Partner 1», «Partner 2»; если же супруг только один, номер опускается.

Итак, прежде всего отыскиваются все записи <FAM>, в которых данная личность представлена как супруг; найденный набор узлов становится значением переменной $partnerships.

Для того чтобы найти имя мужа женщины, нужно следовать по ссылке <HUSB>, а для поиска имени чьей-то жены нужно следовать по ссылке <WIFE>. Поскольку пол может быть не указан, надежнее всего искать обоих партнеров и выводить список людей, отличающихся от первоначальной личности. Это также применимо к бракам с нестандартной сексуальной ориентацией, когда оба супруга одного пола. При обработке генеалогических данных следует проявлять гибкость не только потому, что такие вещи случаются, одобряете вы их или нет, но и потому (что более важно!), что в источниках исторических данных вполне возможны ошибки.

Шаблон выглядит следующим образом:

```
<xsl:template name="show-partners">
   <xsl:variable name="subject" select="."/>
   <xsl:variable name="partnerships"
                    select="key('fam', FAMS/@REF)"/>
   <xsl:for-each select="$partnerships">
      <xsl:sort select="substring(MARR/DATE,
                           string-length(MARR/DATE) - 3)"/>
```

```
      <xsl:variable name="partner"
         select="key('indi',
                   (HUSB/@REF | WIFE/@REF)[.!=$subject/@ID])"/>
      <xsl:variable name="partner-seq">
         <xsl:choose>
            <xsl:when test="count($subject/FAMS)=1"></xsl:when>
            <xsl:otherwise>
               <xsl:value-of select="position()"/>
            </xsl:otherwise>
         </xsl:choose>
      </xsl:variable>

      <xsl:if test="$partner">
         <h2>Partner <xsl:value-of select="$partner-seq"/></h2>
         <p>
         <xsl:apply-templates select="$partner/NAME" mode="link"/>
         </p>
      </xsl:if>

      <xsl:call-template name="show-events"/>

      <xsl:variable name="children" select="key('indi', CHIL/@REF)"/>
      <xsl:if test="$children">
         <p><span class="label">Children:</span><br/>
         <xsl:for-each select="$children">
            <xsl:sort select="substring(BIRT/DATE,
                           string-length(BIRT/DATE) - 3)"/>
            <xsl:value-of select="substring(BIRT/DATE,
                           string-length(BIRT/DATE) - 3)"/>
            <xsl:text> </xsl:text>
            <xsl:apply-templates select="NAME" mode="link"/><br/>
         </xsl:for-each>
         </p>
      </xsl:if>
   </xsl:for-each>
</xsl:template>
```

Как и раньше, супруги перечисляются в хронологическом порядке в соответствии с годом вступления в брак. Если он неизвестен, то в этом случае выбор действий невелик. Для каждого брака перечисляются имена супругов в виде гиперссылок, затем события (обычно только вступление в брак) и, наконец, имена детей, опять же в виде гиперссылок. Дети отыскиваются по полям <CHIL> записи <FAM> и перечисляются по порядку годов рождения, если они известны.

Следующая группа шаблонных правил используется для создания гиперссылок в HTML:

```
<xsl:template match="NAME" mode="link">
   <a>
      <xsl:attribute name="href">
         <xsl:call-template name="make-href"/>
      </xsl:attribute>
```

```
        <xsl:apply-templates/>
      </a>
   </xsl:template>
   <xsl:template match="S">
      <xsl:text> </xsl:text>
      <b><u><xsl:apply-templates/></u></b>
      <xsl:text> </xsl:text>
   </xsl:template>

   <xsl:template name="make-href">
      <xsl:value-of select="concat(../@ID, '.html')"/>
   </xsl:template>
```

Шаблон «make-href» – единственное место, где определяется формат ссылки: в данном случае она состоит из относительного URL другого HTML-файла, имя которого основано на атрибуте ID исследуемой личности, например I27.html.

Таблица стилей заканчивается тривиальным правилом для элементов <BR>, использующихся в качестве разделителей строк текста:

```
   <xsl:template match="BR"><BR/></xsl:template>

   </xsl:transform>
```

Собрание всего воедино

Итак, получена таблица стилей, которая может генерировать HTML-страницу для отдельно выбранной личности. Это пока еще не действующий вебсайт!

Как было сказано ранее, можно действовать тремя способами. Можно выполнить пакетное преобразование всего файла данных в набор связанных статических HTML-страниц, хранящихся на веб-сервере; можно генерировать на сервере каждую страницу по запросу или генерировать страницы динамически в клиенте. Ниже подробно описаны все три способа.

Опубликование статического HTML

Для формирования HTML-файлов для каждой личности, упомянутой в файле данных, нужен некоторый сценарий, который обрабатывает по очереди данные по каждой персоне и для каждого создает отдельный выходной файл. Здесь можно воспользоваться особенностью XSLT 1.1 – возможностью формировать из одного исходного файла несколько выходных файлов. Многие продукты, поддерживающие XSLT 1.0, также обладают подобной возможностью, но, к сожалению, каждый из них использует свой синтаксис.

Кроме того, потребуется новый шаблон для обработки корневого элемента, а так как он должен будет заменить шаблон, определенный в person.xsl, следует использовать <xsl:import>, чтобы новый шаблон имел более высокое преимущество импортирования.

Ниже приведена полная таблица стилей, publish.xsl, для выполнения всех преобразований. Помимо создания HTML-страницы для отдельной личнос-

ти, она также создает указатель, в котором перечислены все упомянутые личности, отсортированные сначала по фамилиям, а затем по именам.

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.1" >

<xsl:import href="person.xsl"/>

<xsl:param name="dir" select="'.'"/>

<xsl:template match="/">
   <xsl:for-each select="*/INDI">
      <xsl:document href="{$dir}/{@ID}.html">
         <xsl:apply-templates select="."/>
      </xsl:document>
   </xsl:for-each>
   <xsl:document href="{$dir}/index.html">
      <xsl:call-template name="make-index"/>
   </xsl:document>
</xsl:template>

<xsl:template name="make-index">
   <html>
   <head>
      <title>Index of names</title>
   </head>
   <body>
   <h1>Index of names</h1>
   <p>
      <xsl:for-each select="/*/INDI/NAME">
         <xsl:sort select="S"/>
         <xsl:sort select="text()"/>
         <a>
            <xsl:attribute name="href">
               <xsl:call-template name="make-href"/>
            </xsl:attribute>
            <xsl:value-of select="S"/>,
            <xsl:for-each select="text()">
               <xsl:value-of select="concat(' ', ., ' ')"/>
            </xsl:for-each>
         </a>
         <br/>
      </xsl:for-each>
   </p>
   </body>
   </html>
</xsl:template>

</xsl:transform>
```

Эту таблицу стилей можно использовать с любым XSLT-процессором, поддерживающим инструкцию XSLT-1.1 `<xsl:document>`, или можно изменить ее, чтобы использовать с любым процессором XSLT-1.0, который имеет рас-

ширение для формирования множественных выходных файлов. На момент написания книги единственным процессором, который мог работать с этой таблицей стилей в ее приведенном здесь варианте, является Saxon. Таким образом, сначала нужно установить процессор Saxon, доступный по адресу *http://users.iclway.co.uk/mhkay/saxon/*. При работе на платформе Windows проще всего установить Instant Saxon, самая свежая версия которого может быть найдена там же.

Кроме того, требуется загрузить файлы с примерами с веб-сайта Wrox. Создайте новый каталог, скопируйте в него таблицы стилей и XML-файл данных, сделайте его текущим каталогом и выполните команду:

```
saxon kennedy.xml publish.xsl
```

Предполагается, что `saxon.exe` указан в PATH; если нет – измените команду, например на `c:\saxondir\saxon`, если процессор установлен в каталоге `c:\saxondir`.

Если требуется записывать создаваемые HTML-файлы в другой каталог, можно указать его в командной строке, например, так:

```
saxon kennedy.xml publish.xsl dir=d:\jfk
```

Новый каталог должен заполниться HTML-файлами. Двойной щелчок по файлу `index.html` выведет на экран индекс имен. Нажатие на любое из них выведет в ярком цвете то, что показано на рис. 10.3. Затем можно просмотреть данные, переходя по ссылкам.

Создание HTML–страниц с помощью сервлета

Альтернативой преобразованию всей массы XML-данных в статические HTML-страницы является создание каждой отдельной HTML-страницы по запросу. Это требует выполнения таблицы стилей на сервере, что, в принципе, можно делать с помощью ASP-страниц, Java-сервлетов или даже обычных CGI-программ. Однако поскольку многие из доступных XSLT-процессоров написаны на языке Java, удобнее использовать сервлеты.

Тем, кто не знаком с программированием сервлетов, вероятно лучше пропустить этот раздел, потому что потребуется слишком много места, если начинать с самых азов. Кому интересно, могут почитать книгу «Professional Java Server Programming», ISBN: 1-861002-77-7, также опубликованную издательством Wrox Press.

Те, кто работает с процессорами Saxon или Xalan, могут писать свой сервлет с расчетом на использование TrAX API, описанного в приложении F; по-видимому, в других Java-процессорах, в частности от Oracle, также будет реализовано что-нибудь соответствующее. Таким образом, можно написать сервлет, который работает с любым процессором. Фактически большинство процессоров поставляется с некоторым интерфейсом для сервлетов, хотя часто лучше все же настроить его, чтобы он отвечал требованиям конкретного приложения. Поскольку многое зависит от среды, в которой выполняется

работа, трудно дать универсальный действующий пример, поэтому здесь дается просто схема проектирования.

Специфическая особенность этого приложения в том, что тут выполняется множество запросов для получения данных из одного исходного документа, при этом используется одна и та же таблица стилей, но с разными параметрами. Лучше всего поэтому сохранять и исходный документ, и таблицу стилей в памяти на сервере, чтобы не требовалось проводить синтаксический разбор всего XML-документа для отображения данных по каждой отдельной личности.

Предполагается, что входящие запросы от броузера будут иметь такой вид:

```
http://www.myserver.com/examples/servlet/GedServlet?tree=kennedy&id=I1
```

Параметры, включенные в URL, – это во-первых, название набора данных, который следует использовать (желательно, чтобы сервер мог обрабатывать несколько наборов данных одновременно), и, во-вторых, идентификатор личности, сведения о которой нужно отобразить.

> ***Когда вышеупомянутый URL включается в XML-документ, символ «&» должен быть представлен в форме «&». Большинство HTML-броузеров примет оба варианта: «&» и «&», но последний строго отвечает спецификации HTML, и именно его генерирует рассмотренная таблица стилей.***

Таким образом, прежде всего, нужно генерировать гиперссылки в таком формате. Это можно сделать, написав новый модуль таблицы стилей, который импортирует `person.xsl` и переопределяет шаблон, формирующий гиперссылки. Назовем его `ged-servlet.xsl`.

Модуль таблицы стилей `ged-servlet.xsl` выглядит следующим образом. В нем введен дополнительный параметр – название требуемого дерева, так как этот же сервлет должен обрабатывать запросы к данным из различных генеалогических деревьев. Кроме того, он переопределяет шаблон «make-href» на шаблон, генерирующий гиперссылки в нужном формате:

```
<xsl:transform
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0" >

<xsl:import href="person.xsl"/>
<xsl:param name="tree"/>

<xsl:template name="make-href">
   <xsl:value-of select="concat('/examples/servlet/GedServlet?tree=',
                        $tree, '&amp;id=', ../@ID)"/>
</xsl:template>

</xsl:transform>
```

Если сервлет находится не в указанном здесь месте, нужно учесть это в таблице стилей и использовать другой URL.

Таблицу стилей и интерфейс сервлета также можно расширить, чтобы формировать указатель имен, как и в предыдущем примере, но поскольку это простая задача, ее можно предложить читателям для самостоятельной работы.

Сложнее написать сам сервлет. Код, приведенный ниже, использует интерфейс TrAX, описанный в приложении F, и должен работать с любым XSLT-процессором, который поддерживает этот API. На момент написания книги Xalan реализует как пакет `javax.xml.transform`, так и пакет `javax.xml.parsers`, в то время как Saxon реализует только `javax.xml.transform`. Для использования этого сервлета с процессором Saxon потребуется отдельный синтаксический анализатор, поддерживающий классы `javax.xml.parsers`: например Crimson с *http://xml.apache.org/*.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Hashtable;

import org.w3c.dom.Document;

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;
import org.xml.sax.SAXException;

public class GedServlet extends HttpServlet {

    /**
    * Отвечаем на HTTP-запрос
    */

    public void service(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
    {
        res.setContentType("text/html");

        try {
            String clear = req.getParameter("clear");
            if (clear!=null && clear.equals("yes")) {
                resetData();
            }
            String family = req.getParameter("tree");
            Document doc = getSourceDocument(family);
            Source source = new DOMSource(doc);

            Result result = new StreamResult(res.getOutputStream());

            Templates style = getStyleSheet();
            Transformer transformer = style.newTransformer();
            transformer.setParameter("id", req.getParameter("id"));
            transformer.setParameter("tree", family);
            transformer.transform(source, result);

        } catch (TransformerException err) {
```

```
            res.getOutputStream().
                  println("Error applying stylesheet: " + err.getMessage());
        }
    }

    /**
    * Переносим подготовленную таблицу стилей в память; при необходимости
подготавливаем ее
    */

    private synchronized Templates getStyleSheet()
    throws TransformerConfigurationException {
        if (stylesheet == null) {
            File sheet =
                 new File(getServletContext().getRealPath("/ged-servlet.xsl"));
            stylesheet = TransformerFactory.newInstance().
                             newTemplates(new StreamSource(sheet));
        }
        return stylesheet;
    }

    /**
    * Загружаем исходный документ
    */
    private synchronized Document getSourceDocument(String tree)
    throws ParserConfigurationException, SAXException {
        Document doc = (Document)trees.get(tree);
        if (doc==null) {
            File source =
                new File(getServletContext().getRealPath("/" + tree + ".xml"));
            doc = DocumentBuilderFactory.newInstance().
                                        newDocumentBuilder().parse(source);
        }
        return doc;
    }

    /**
    * Сбрасываем данные, хранимые в памяти
    */

    private synchronized void resetData() {
        trees = new Hashtable();
        stylesheet = null;
    }

    private Hashtable trees = new Hashtable();
    private Templates stylesheet = null;
}
```

XML-файл, содержащий данные генеалогического дерева, — это файл *tree*.xml**, где под** *tree* **подразумевается конкретное генеалогическое дерево, в данном случае это** kennedy.xml**. Файл должен быть в домашнем каталоге веб-приложения, содержащего сервлет, который определен в параметрах кон-**

фигурации вашего веб-сервера. Два модуля таблиц стилей: `person.xsl` и `ged-servlet.xsl`, – также должны находиться в этом каталоге.

Сервлет сохраняет в памяти копию скомпилированной таблицы стилей (объект TrAX `Templates`) – эта копия создается тогда, когда таблица стилей требуется в первый раз. Кроме того, он сохраняет в памяти объект DOM `Document`, представляющий каждое генеалогическое дерево. Различные документы индексируются с использованием хэш-таблицы. Имейте в виду, что все обращения к этим переменным должны выполняться в синхронизированных методах, чтобы избежать конфликтов при параллельном доступе, так как в сервлете несколько потоков сразу могут выполнять один и тот же код.

Затем сервлет создает новый экземпляр таблицы стилей – объект TrAX `Transformer`, – который не является разделяемым с каким-либо другим потоком и используется только однажды. Перед тем как использовать этот объект для обработки исходного документа, вызывая его метод `renderDocument()`, сервлет передает ему параметры (извлеченные из URL) и заданное назначение вывода.

Формирование HTML с помощью ASP-страниц

При работе в среде Microsoft альтернативой написанию Java-сервлетов для выполнения преобразований на сервере может служить управление процессом с помощью ASP-страниц. В этом случае, по-видимому, будет применяться синтаксический анализатор и XSLT-процессор Microsoft MSXML3. Хотя MSXML3 известен благодаря выполнению преобразований на стороне клиента, он также эффективен и как серверное средство. Он использует интерфейсы COM, поэтому может быть вызван из ASP-страниц точно так же, как любой другой объект COM. API для управления преобразованием с использованием MSXML3 описан в приложении А.

В генерируемой HTML-странице потребуется изменить гиперссылки, чтобы на этот раз для формирования вывода сведений по другой личности они обращались бы к ASP-странице. Такого изменения в таблице стилей снова можно достичь написанием другой таблицы стилей, которая импортирует первую, изменяя в ней только шаблон, формирующий гиперссылку. Это таблица стилей `ged-asp.xsl`:

```
<xsl:transform
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0"
>

<xsl:import href="person.xsl"/>
<xsl:param name="tree"/>

<xsl:template name="make-href">
<xsl:value-of select="concat('GedAsp.asp?tree=', $tree, '&amp;id=', ../@ID)"/>
</xsl:template>

</xsl:transform>
```

Для того чтобы код ASP оставался простым, мы будем каждый раз перезагружать исходный документ и таблицу стилей. В настоящем приложении имеет смысл хранить их в виде объектов уровня приложения или сеанса. Ниже приведена полноценная ASP-страница (она тоже выполнена коротко, без всякой обработки ошибок):

```
<%@ LANGUAGE = JScript %>
<%

  // Считываем параметры запроса из URL
 var family = Request.QueryString("tree") + "";
 var indi = Request.QueryString("id") + "";
  // Устанавливаем местоположение исходного файла и таблицы стилей
  var sourceFile = Server.MapPath("kennedy.xml");
  var styleFile = Server.MapPath("ged-asp.xsl");

  // Загружаем XML
  var source = new ActiveXObject("MSXML2.DOMDocument");
  source.async = false;
  source.load(sourceFile);

  // Загружаем XSLT
  var style = new ActiveXObject("MSXML2.FreeThreadedDOMDocument");
  style.async = false;
  style.load(styleFile);

  var transformer = new ActiveXObject("MSXML2.XSLTemplate");
  transformer.stylesheet = style.documentElement;

  var xslproc = transformer.createProcessor();
  xslproc.input = source;

  // Устанавливаем параметры преобразования
  xslproc.addParameter("id", indi, "");
    xslproc.addParameter("tree", family, "");

  // Запускаем преобразование
  xslproc.transform();
  Response.Write(xslproc.output);

%>
```

Формирование HTML в броузере

Наконец, рассмотрим еще один способ отображения генеалогического дерева: а именно загрузку всего XML-файла в броузер целиком и использование затем сценариев на стороне клиента для вызова обработки таблицы стилей при каждом щелчке пользователя по гиперссылке. Этот специфический пример запускается только в Internet Explorer 5 и только при наличии установленного синтаксического XML-анализатора MSXML3 от Microsoft (следует надеяться, что этот пример будет выполняться и в последующих версиях). Информация по установке этого программного продукта включена в приложение А.

Если XML-файл большой (генеалогические деревья, созданные генеалогами-профессионалами часто достигают размеров в несколько мегабайт), тогда при таком подходе пользователю придется довольно долго ждать открытия первой страницы данных. Но зато после первоначальной загрузки можно работать с файлом автономно: устраняется необходимость обращаться к серверу для перехода по ссылкам от одной личности к другой. Это дает пользователю мгновенный отклик на навигационные запросы, снижает нагрузку на сервер и число обращений к серверу. Другим преимуществом, связанным с частым наличием доступа только к ограниченному пространству в Сети, предоставляемому коммерческим Интернет-провайдером, является то, что в данном случае на сервере не нужно устанавливать никаких специальных программ. Однако пока поддержка XSLT броузерами не станет универсальной, сохраняется недостаток этого способа – его доступность только пользователям, которые потрудились установить синтаксический анализатор MSXML3.

На этот раз преобразование управляется JavaScript-кодом на HTML-странице `famtree.html`. Страница выглядит следующим образом. Элементы `<script>` содержат клиентскую часть JavaScript-кода.

```
<html>
<head>
   <title>Family Tree</title>
   <style type="text/css">
      ... как и раньше ...
   </style>
   <script>
      var source = null;
      var style = null;
      var transformer = null;

      function init() {
         source =
            new ActiveXObject("MSXML2.DOMDocument");
         source.async = false;
         source.load('kennedy.xml');

         style =
            new ActiveXObject("MSXML2.FreeThreadedDOMDocument");
         style.async = false;
         style.load('ms-person.xsl');

         transformer = new ActiveXObject("MSXML2.XSLTemplate");
         transformer.stylesheet = style.documentElement;
         refresh("I1");
      }

      function refresh(indi) {
         var xslproc = transformer.createProcessor();
         xslproc.input = source;
         xslproc.addParameter("id", indi, "");
         xslproc.transform();
         displayarea.innerHTML = xslproc.output;
      }
```

```
    </script>
    <script for="window" event="onload">
       init();
    </script>
</head>
<body>
    <div id="displayarea"></div>
</body>
</html>
```

Определения стилей CSS лишь перемещены из таблицы стилей XSLT на HTML-страницу, но никак не изменены.

При загрузке этой страницы вызывается функция `init()`. Она создает два объекта DOM: один для исходного XML, а другой для таблицы стилей, – и загружает их, используя относительные URL `kennedy.xml` и `ms-person.xsl`. Затем она компилирует таблицу стилей в объект со сбивающим с толку названием `XSLTemplate` – этот объект соответствует объекту TrAX `Templates`. Наконец, она вызывает функцию `refresh()` для отображения личности с идентификатором `I1`.

> *Здесь следует оговориться. Нет гарантии, что файл GEDCOM обязательно содержит личность с таким идентификатором. Более аккуратно созданные приложения либо отобразят первую личность в файле, либо указатель всех упоминаемых личностей.*

Функция `refresh()`создает исполняемый экземпляр таблицы стилей с помощью вызова метода `createProcessor()` объекта `XSLTemplate`. Затем она устанавливает значение глобального параметра `id` таблицы стилей и применяет таблицу стилей к исходному документу, вызывая метод `transform()`. HTML, создаваемый при обработке таблицы стилей, записывается в содержимое элемента `<div id="displayarea">` в теле HTML-страницы.

Можно снова использовать ту же самую таблицу стилей с новыми изменениями в формате гиперссылок. На этот раз нужно, чтобы гиперссылка на другую личность, скажем с `I2`, представлялась следующим образом:

```
<a href="Javascript:refresh('I2')">Jaqueline Lee Bouvier</a>
```

Когда пользователь щелкает по этой гиперссылке, выполняется функция `refresh()`, что вызывает новое применение откомпилированной таблицы стилей к тому же исходному документу, но с другим значением параметра `id`. В результате содержимое страницы меняется, и отображаются сведения о другой личности.

Таблица стилей `ms-person.xsl` снова пишется путем импортитрования стандартной таблицы стилей `person.xsl`, обсуждавшейся ранее, и внесения в нее желаемых изменений. На этот раз их два: изменение формы гиперсылок и исключение создания CSS-стиля, поскольку необходимые определения уже имеются на HTML-странице. Вот эта таблица стилей:

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
 version="1.0"
>

<xsl:import href="person.xsl"/>

<!-- Изменяем способ генерирования гиперссылок -->
<xsl:template name="make-href">
    <xsl:variable name="apos">'</xsl:variable>
      <xsl:value-of
         select="concat('Javascript:refresh(', $apos, ../@ID, $apos, ')')"/>
</xsl:template>

<!-- Не генерируем таблицу стилей CSS -->

<xsl:template name="css-style"/>

</xsl:transform>
```

В полученной таблице есть небольшая неувязка: она полностью генерирует HTML-страницу с элементами <html>, <head> и <body>, а затем вставляет ее в качестве содержимого элемента в существующую HTML-страницу. К счастью, Internet Explorer легко допускает такое вольное обращение со спецификацией HTML, хотя он игнорирует цвет фона, указанный в сформированной (внутренней) HTML-странице. Можно считать это эстетическим излишеством оригинала. Если же есть желание установить при преобразовании цвет фона (или другие особенности, например заголовок страницы), потребуется некоторый дополнительный код на JavaScript, копирующий соответствующую информацию из вывода преобразования в нужное место на HTML-странице.

Если шаблон «match="INDI"» в оригинальной таблице стилей person.xsl разбит на несколько именованных шаблонов, можно будет заменить один из них, чтобы избежать создания ненужного каркаса HTML-страницы. Но в написанном варианте таблица стилей работает, так что можно устоять перед искушением доводить ее до совершенства.

Преобразование файлов GEDCOM в XML

В начале этого примера говорилось, что формат GEDCOM, широко используемый генеалогическими пакетами программ, не является форматом XML, но его легко преобразовать в XML. Имея эти действующие примеры, вполне можно применить их для отображения собственного генеалогического дерева. Сейчас есть широкий выбор пакетов для работы с генеалогическими деревьями, и любой приличный пакет может осуществлять экспорт GEDCOM. Список доступных пакетов можно найти на *http://www.cyndislist.com/software.htm*. Неплохим началом послужит бесплатный пакет Personal Ancestral File, или PAF.

Очевидный способ преобразования GEDCOM в XML состоит в написании программы, которая на входе получает файл GEDCOM и создает на выходе файл XML. Однако есть более грамотный способ: почему бы не написать синтаксический анализатор GEDCOM, подобный совместимому с SAX синтак-

сическому анализатору XML, чтобы любая программа, способная обрабатывать ввод SAX, могла читать GEDCOM непосредственно, только переключившись на другой синтаксический анализатор? В частности, многие XSLT-процессоры могут работать с совместимым с SAX синтаксическим анализатором, так что это дает возможность передать GEDCOM прямо в таблицу стилей.

Аналогично многие XSLT-процессоры могут посылать конечное дерево в указанный пользователем `DocumentHandler` в форме потока событий SAX, поэтому если написать совместимый с SAX2 обработчик `ContentHandler`, то XSLT-процессор также сможет выводить файлы GEDCOM. Таким образом, оказывается, что можно написать таблицу стилей для преобразования одного файла GEDCOM в другой, без необходимости создания XML-файла как промежуточной фазы. В качестве примера в комплект файлов на веб-сайте включена таблица стилей `nonliving.xsl`, которая удаляет из набора данных сведения обо всех живых людях, что является благоразумной любезностью по отношению к родственникам в случае опубликования собственного генеалогического дерева в сети, к тому же, в некоторых странах этого требует закон.

На веб-сайте Wrox вместе с примерами файлов из этой главы выложен и синтаксический анализатор SAX2 для GEDCOM; он называется `GedcomParser`. GEDCOM использует архаичный набор символов под названием ANSEL, поэтому наряду с `GedcomParser` выложен и другой класс – `AnselInputStreamReader` – для преобразования символов ANSEL в Unicode.

Точно так же на стороне вывода есть еще обработчик SAX2 `ContentHandler` под названием `GedcomOutputter`, который, в свою очередь, преобразовывает Unicode в ANSEL, используя `AnselOutputStreamWriter`.

Эта структура показана ниже на схеме:

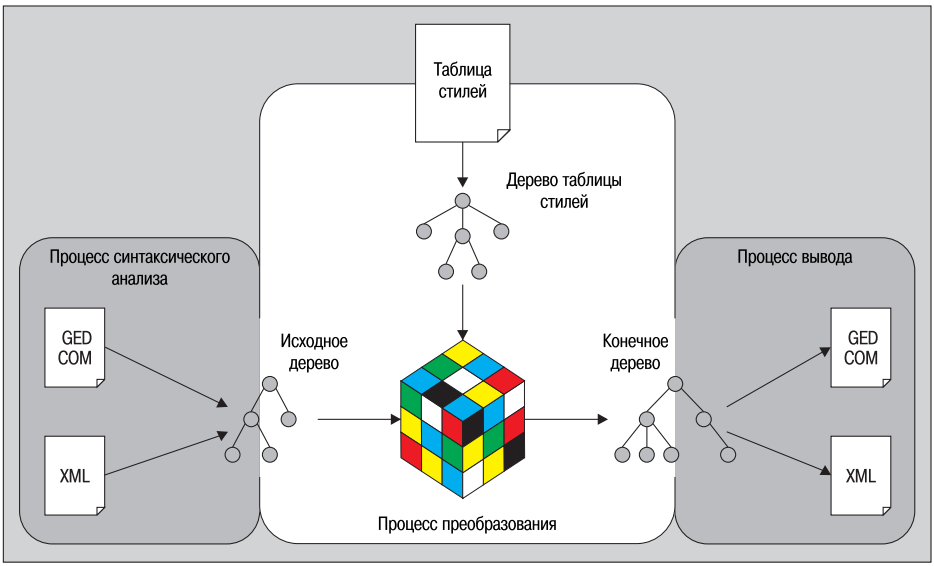


***Рис. 10.4.** Схема преобразования файлов GEDCOM*

Если потребуется увидеть XML-код, всегда можно передать GEDCOM в таблицу стилей, которая производит тождественное преобразование. Самым простым примером служит:

```
<xsl:transform
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0" >

<xsl:template match="/">
   <xsl:copy-of select="."/>
</xsl:template>

</xsl:transform>
```

Фактически для создания файла данных `kennedy.xml`, использованного в качестве примера ввода для этих таблиц стилей, была взята несколько более сложная таблица стилей, чем эта, которая применяется к уже обработанному выводу синтаксического анализатора GEDCOM.

Она называется `normalize.xsl` и переводит различные структуры, встречающиеся в GEDCOM, в более привычное XML-представление. В частности:

- Имена в GEDCOM представлены в форме «`Michael /Kay/`», но в XML их проще обрабатывать, если фамилии выделены следующим образом: «`Michael <S>Kay</S>`».
- Для указаний о продолжении строки GEDCOM использует теги `CONT` и `CONC`: `CONT` представляет новую строку, а `CONC` служит признаком сцепления с предыдущей строкой. Таблица стилей `normalize.xsl` удаляет теги `CONC`, а перед строкой с `CONT` вставляет пустой тег `<BR/>`, как в HTML.

Таким образом, если, например, потребуется конвертировать собственный файл GEDCOM `mytree.ged` в XML, можно сделать это с помощью Saxon, выполнив следующую команду

```
saxon -x GedcomParser mytree.ged normalize.xsl >mytree.xml
```

Резюме

Автор надеется, что этот небольшой экскурс в удивительный мир генеалогических моделей данных показал некоторые возможности XSLT как инструмента для оперирования сложными структурированными данными и составления отчетов по ним.

Здесь охвачено много аспектов:

- Как управлять перемещением по сложно связанным данным в пределах XML-документа.
- Несколько различных способов формирования интерактивного представления большого набора данных XML:
  - Формирование сразу множества статических HTML-страниц во время публикования.

- Формирование HTML-страниц динамически, используя Java-сервлеты или ASP-страницы Microsoft.
- Формирование HTML по частям прямо в броузере.

- Использование XSLT для преобразования структурированных данных, которые первоначально не были в формате XML.

Следующий работающий пример направлен в еще более далекую от XML область и использует XSLT для решения шахматной проблемы.

Таблица стилей для маршрута коня

Этот пример таблицы стилей довольно надуманный, но он демонстрирует некоторые дополнительные возможности использования XSLT, которые могут пригодиться при сложных манипуляциях с данными.

Назначение данной таблицы стилей – разработать маршрут коня по шахматной доске при условии, что он обязательно посещает каждую клетку и только по одному разу, как показано на рисунке. Конь может перемещаться в любую клетку, находящуюся в противоположном углу прямоугольника размером 3×2.

Рис. 10.5. *Маршрут коня по шахматной доске, начиная с позиции e5*

Единственными исходными данными для таблицы стилей будет указание стартовой клетки. В современной шахматной нотации столбцы обозначаются буквами a–h, начиная слева, а ряды – цифрами 1–8, начиная снизу. Стартовая клетка будет задаваться в виде параметра для таблицы стилей. Таблице стилей не нужны никакие данные из исходного документа. Для соответствия требованиям языка исходный документ должен существовать, но спецификация не обязывает таблицу стилей читать его.

Будем создавать таблицу стилей поэтапно; полную таблицу стилей можно найти на веб-сайте Wrox – это файл `tour.xsl`.

Версия, описанная здесь, использует новую возможность XSLT 1.1 для обращения к переменной, значением которой является дерево, как к набору узлов. По этой причине в таблице стилей встретятся конструкции типа `«$board/square»`, которые процессор XSLT 1.0 отклонит как ошибки. В дан-

ном варианте таблица стилей будет работать с процессором Saxon. Если имеющийся процессор XSLT 1.0 поддерживает функцию расширения `node-set()`, можно видоизменить для него таблицу стилей, переписав эти выражения: например, заменив «`$board/square`» на «`xx:node-set($board)/square`». Среди программ для загрузки есть версия таблицы стилей, адаптированная таким образом для MSXML3 Microsoft – `ms-tour.xsl`. Кроме того, там выложена версия таблицы стилей, использующая только стандартные средства XSLT 1.0. В ней все действующие структуры данных представлены в виде строк, а не как деревья. Не буду объяснять, как работает эта таблица стилей, – это можно исследовать самостоятельно – файл `tour10.xsl`.

> *На создание такой таблицы стилей автора вдохновил пример Орена Бен-Кики (Oren Ben-Kiki), который опубликовал таблицу стилей для решения задачи с восемью королевами. Концепции обеих задач очень похожи, хотя в деталях они различаются.*

Алгоритм

Стратегия обхода конем всей шахматной доски основана на наблюдении, что если клетка еще не посещена конем, то нужно, чтобы имелись по крайней мере две клетки, куда из нее может перейти конь, поскольку должен существовать способ попасть на клетку и уйти с нее. Это означает, что если есть возможность хода на клетку, у которой остается только один выход, лучше сразу сделать ход на эту клетку, иначе больше такая возможность не представится.

Это предполагает подход, когда перед каждым ходом анализируются все клетки, на которые можно перейти, и выбирается та из них, у которой в этом случае останется наименьшее количество возможных выходов. Оказывается, эта стратегия удачна, и коню всегда удается обойти всю доску. Автор не знает, как это доказать, и будет счастлив узнать доказательство от кого-то из читателей, но во всяком случае это работает.

Проектирование таблицы стилей лучше начинать со структуры данных. Здесь основная структура данных, которая нужна, – непосредственно шахматная доска. Необходимо знать, какие клетки конь посетил, а для того, чтобы можно было в конце распечатать шахматную доску с результатом, нужно знать последовательность, в которой посещались клетки. В XSLT 1.0 нет большого выбора типов данных, поэтому единственная практическая возможность – строковый тип; а в XSLT 1.1 можно использовать временное дерево, с которым гораздо проще работать. В данном случае будет использоваться дерево, содержащее 64 элемента `<square>`, каждый из них представляет собой одну клетку на доске. Значением каждого элемента должно быть целое число между «`1`» и «`64`», если клетка была посещена (при этом число указывает порядковый номер посещения), или элемент должен быть пуст, если конь не посещал соответствующую клетку.

В обычной программе эта структура данных, возможно, была бы сохранена в глобальной переменной и модифицировалась бы после каждого хода коня.

В XSLT это невозможно, поскольку переменные нельзя модифицировать. Вместо этого при каждом вызове шаблона он передает текущее состояние шахматной доски как параметр, а когда конь делает ход, создается новая копия шахматной доски, которая отличается от предыдущей только значением одной клетки.

Фактически не имеет значения, каким образом пронумерованы клетки, но для согласованности будем нумеровать их, как показано на рис. 10.6:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Рис. 10.6. *Нумерация клеток шахматной доски*

Так, если ряды пронумеровать от 0 до 7 и столбцы тоже от 0 до 7, то номер клетки можно получить из выражения `«ряд × 8 + столбец»`.

Выбрав структуру данных, можно обсудить структуру всей программы. Поделим ее на три стадии:

- Подготовка исходной структуры данных (пустая доска с конем, помещенным на какую-нибудь клетку)
- Вычисление маршрута
- Отображение конечного состояния доски

Вычисление маршрута включает 63 хода, каждый из которых представляется следующим образом:

- Отыскание всех непосещенных клеток, на которые конь может перейти из текущей позиции
- Вычисление для каждой из них количества выходов (то есть количества непосещенных клеток, на которые можно с них перейти)
- Выбор клетки с наименьшим количеством выходов и перемещение коня на нее

Теперь можно приступить к программированию. Сложность здесь в том, как читатель, вероятно, уже догадался, что все циклы нужно писать с использованием рекурсии. Поначалу к этому нужно привыкнуть, но это быстро становится привычкой.

Корневой шаблон

Начнем со структуры элементов верхнего уровня:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.1">

<xsl:param name="start" select="'a1'"/>

<!-- start-column - это целое в диапазоне 0-7 -->

<xsl:variable name="start-column"
   select="number(translate(substring($start, 1, 1),
         'abcdefgh', '01234567'))"/>

<!-- start-row - это целое в диапазоне 0-7, где ноль соответствует верхнему ряду -->

<xsl:variable name="start-row"
   select="8 - number(substring($start, 2, 1))"/>

. . .

</xsl:transform>
```

Здесь только лишь объявлен глобальный параметр `start`, определяющий стартовую клетку, из которого получены две глобальные переменные: номер ряда и номер столбца.

Некоторые наблюдения:

- Параметр `start` имеет значение по умолчанию `a1`. Поскольку это – строковое значение, оно должно быть в кавычках; эти кавычки – дополнительные к кавычкам, окружающим атрибут XML. Если написать «`select="a1"`», значением по умолчанию было бы строковое значение дочернего элемента `<a1>` корневого узла.
- Самый простой способ преобразования буквенного идентификатора столбца (a–h) в цифровой (0–7) состоит в использовании функции `translate()`, которая описана в главе 7.
- Номер ряда вычитается из 8, чтобы ряд с наименьшим номером был наверху и чтобы номера рядов начинались с нуля. Нумерация от нуля облегчает преобразования между номерами рядов и столбцов и номером каждой клетки на доске в диапазоне 0–63.
- Еще не проверено, что указанная стартовая клетка допустима. Это будет сделано в корневом шаблоне.

Теперь можно перейти к корневому шаблону. Он вызывается, когда найден корневой узел исходного документа, но фактически он ничего не делает с исходным документом.

Корневой шаблон определяет следующие стадии обработки:

- Проверить допустимость указанного параметра
- Установить пустую шахматную доску и разместить на ней коня на указанной стартовой клетке
- Вычислить маршрут коня
- Вывести маршрут в формате HTML

Все эти задачи будут решать другие шаблоны, так что сам корневой шаблон довольно прост:

```
<xsl:template match="/">

   <!-- Проверка допустимости входного параметра -->
   <xsl:if test="not(string-length($start)=2) or
     not(translate(substring($start,1,1), 'abcdefgh', 'aaaaaaaa')='a') or
     not(translate(substring($start,2,1), '12345678', '11111111')='1')">
      <xsl:message terminate="yes"
         >Неверный параметр: попробуйте (скажем) 'a1' or 'f6'</xsl:message>
   </xsl:if>

   <!-- Настройка пустой доски -->
   <xsl:variable name="empty-board">
      <xsl:call-template name="make-board"/>
   </xsl:variable>

   <!-- Помещаем коня в указанную стартовую позицию на доске -->
   <xsl:variable name="initial-board">
      <xsl:call-template name="place-knight">
         <xsl:with-param name="move" select="1"/>
         <xsl:with-param name="board" select="$empty-board"/>
         <xsl:with-param name="square"
                  select="$start-row * 8 + $start-column"/>
      </xsl:call-template>
   </xsl:variable>

   <!-- Вычисление маршрута коня -->
   <xsl:variable name="final-board">
      <xsl:call-template name="make-moves">
         <xsl:with-param name="move" select="2"/>
         <xsl:with-param name="board" select="$initial-board"/>
         <xsl:with-param name="square"
                  select="$start-row * 8 + $start-column"/>
      </xsl:call-template>
   </xsl:variable>

   <!-- создание вывода в формате HTML -->
   <xsl:call-template name="print-board">
      <xsl:with-param name="board" select="$final-board"/>
   </xsl:call-template>

</xsl:template>
```

Обратите внимание, как размещено большинство вызовов <xsl:call-template> внутри <xsl:variable>. Это единственный способ получения результата от вызванного шаблона. <xsl:variable> создает новое временное дерево, и любой вывод, произведенный вызванным шаблоном, направляется в это дерево. Конечным значением переменной является набор узлов, содержащий единственный узел – корневой узел временного дерева.

Стоит проанализировать код для проверки допустимости параметра start. Он рассматривает три критерия: что длина указанной строки равна двум

символам, что первый символ является одной из букв a–h и что второй символ является одной из цифр 1–8. Самый простой путь для выполнения этих проверок – использование функции translate() показанным способом. Если любой из критериев не выполняется, таблица стилей выводит сообщение с помощью <xsl:message> и прекращает работу.

Несколько переменных (empty-board, initial-board и final-board) представляют шахматную доску, содержащую все или часть ходов коня. Все эти переменные – деревья, и все они имеют одинаковую структуру: корневой узел, содержащий 64 узла элементов <square>. Вспомните, что деревья XPath не обязаны быть корректными XML-документами, поэтому нет никакой потребности во внешнем элементе, который содержит все элементы <square>.

Если клетка была посещена, ей дается порядковый номер, указывающий последовательность посещения (1 для стартовой клетки, 2 для следующей посещенной клетки, и так далее). Если клетка не была посещена, элемент пуст.

Клетки шахматной доски представляются целыми числами в диапазоне 0–63, которые вычисляются с помощью выражения $row × 8 + $column.

Установка шахматной доски

Сейчас пора обсудить, как работают вызываемые шаблоны. Начнем с шаблона make-board, задача которого – инициализировать пустую шахматную доску. Можно было бы написать для этого таблицу стилей с 64 пустыми элементами <square/>, но это задача не для ленивых, поэтому здесь выбран другой путь:

```
<xsl:template name="make-board">
   <xsl:param name="size" select="64"/>
   <xsl:if test="$size!=0">
      <square/>
      <xsl:call-template name="make-board">
         <xsl:with-param name="size" select="$size - 1"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

Когда этот шаблон вызывается из корневого шаблона, параметр $size принимает свое значение по умолчанию – 64. Шаблон создает одну непосещенную клетку (обозначаемую <square/>), а затем вызывает себя, чтобы создать оставшиеся 63. После завершения 64 рекурсивных вызовов параметр $size достигает нуля; на этой стадии будут записаны 64 непосещенных клетки, и пустая шахматная доска готова.

На следующей стадии нужно поместить коня в его исходное положение, используя шаблон «place-knight». Вот он:

```
<xsl:template name="place-knight">
   <xsl:param name="move"/>
   <xsl:param name="board"/>
   <xsl:param name="square"/>
```

```
    <xsl:copy-of select="$board/square[position()-1 &lt; $square]"/>
    <square><xsl:value-of select="$move"/></square>
    <xsl:copy-of select="$board/square[position()-1 &gt; $square]"/>
</xsl:template>
```

Этот шаблон принимает три параметра: номер данного хода, текущее состояние шахматной доски и клетку, на которую должен быть помещен конь. При вызове его из корневого шаблона номер хода всегда первый, а доска всегда пустая, но позже этот шаблон будет использован уже с другими параметрами.

Этот шаблон копирует всю шахматную доску до и после клетки, в которую должен быть помещен конь. Сама эта клетка заменяется новым элементом `<square>`, содержимое которого – порядковый номер хода. Например, пятый ход будет записан как `<square>5</square>`.

Нельзя, конечно, изменить шахматную доску. Все переменные в XSLT неизменяемые. Вместо этого нужно создать новую доску как измененную копию оригинала. Результат работы шаблона (значение, записанное в его текущее назначение вывода) – дерево, представляющее новое состояние шахматной доски после размещения коня.

Отображение конечного состояния шахматной доски

Можно отвлечься на время от шаблона, который вычисляет ходы коня, и описать относительно простую задачу вывода конечного состояния шахматной доски в формате HTML. Фактически для этого нужны три именованных шаблона. Первый создает каркас HTML и вызывает «print-rows» для вывода рядов:

```
<xsl:template name="print-board">
   <xsl:param name="board"/>
   <html>
      <head>
         <title>Маршрут коня</title>
   </head>
   <body>
      <div align="center">
      <h1>Маршрут коня начинается с <xsl:value-of select="$start"/></h1>
      <table border="1" cellpadding="4">
         <xsl:call-template name="print-rows">
            <xsl:with-param name="board" select="$board"/>
         </xsl:call-template>
      </table>
      </div>
   </body>
   </html>
</xsl:template>
```

Второй шаблон печатает строки. Фактически он печатает только первую строку (вызывая шаблон «print-columns»), а затем рекурсивно вызывает себя,

чтобы печатать остающиеся строки. Он останавливается, когда готовы все восемь строк:

```
<xsl:template name="print-rows">
   <xsl:param name="board"/>
   <xsl:param name="row" select="0"/>
   <xsl:if test="$row &lt; 8">
      <tr>
         <xsl:call-template name="print-columns">
            <xsl:with-param name="board" select="$board"/>
            <xsl:with-param name="row" select="$row"/>

         </xsl:call-template>
      </tr>

      <xsl:call-template name="print-rows">
          <xsl:with-param name="board" select="$board"/>
          <xsl:with-param name="row" select="$row + 1"/>
       </xsl:call-template>
   </xsl:if>
</xsl:template>
```

Шаблон «print-columns» очень похож. Он печатает первый столбец, а затем вызывает сам себя для печати остальных.

```
<xsl:template name="print-columns">
   <xsl:param name="board"/>
   <xsl:param name="row"/>
   <xsl:param name="column" select="0"/>
   <xsl:if test="$column &lt; 8">
      <xsl:variable name="color">
         <xsl:choose>
            <xsl:when test="($row + $column) mod 2">xffff44</xsl:when>
            <xsl:otherwise>white</xsl:otherwise>
         </xsl:choose>
      </xsl:variable>
      <td align="center" bgcolor="{$color}">
         <xsl:value-of select="$board/square[$row*8 + $column + 1]"/>
      </td>

      <xsl:call-template name="print-columns">
         <xsl:with-param name="board" select="$board"/>
         <xsl:with-param name="row" select="$row"/>
         <xsl:with-param name="column" select="$column + 1"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

Шаблон содержит некоторую логику, чтобы добиться традиционной окраски клеток шахматной доски. Он использует оператор «mod» для проверки, кратна ли 2 сумма номера ряда и номера столбца, и обработывает два возможных варианта по инструкции <xsl:choose>.

Фактическое содержимое каждой клетки – порядковый номер хода, который извлекается из соответствующего узла <square> во временном дереве, представляющем шахматную доску.

Определение маршрута

Это было все, что касается ввода и вывода таблицы стилей, а теперь основное: алгоритм для определения маршрута коня.

Основной используемый алгоритм заключается в том, что перед каждым ходом рассматриваются все клетки, куда можно пойти, и выбирается та из них, у которой остается наименьшее количество выходов. Например, если конь находится на c2, тогда можно было бы сделать ход на a1, e1, a3, e3, b4 или d4, считая, что все они еще не посещены. Из них угловая клетка a1 имеет только один выход – на b3, и если не сделать сейчас ход на эту угловую клетку, то позже подходы к ней будут заблокированы. Оказалось, что эта стратегия – делать ход на клетку с наименьшим количеством выходов – всегда приводит к успеху и конь полностью обходит все клетки на доске, хотя если вдруг произойдет неудача, алгоритм достаточно гибок, чтобы вернуться на несколько ходов назад и попробовать другой маршрут.

Корневой шаблон вызывает шаблон «make-moves». Этот шаблон, начиная с любой заданной стартовой позиции, вычисляет все ходы, необходимые для полного обхода конем шахматной доски. Конечно, он делает это с помощью рекурсии: но в отличие от предыдущих шаблонов, которые прямо вызывали себя, данный шаблон делает это косвенно, через другой шаблон, под названием «try-possible-moves».

> ***Первое, что делает шаблон «make-moves», – вызывает шаблон*** «list-possible-moves» ***для создания списка возможных в текущей ситуации ходов. Полученный результат – список ходов – представляется в виде структуры, очень похожей на структуру данных самой шахматной доски. Список представляется как временное дерево, а каждый ход представлен элементом*** <move>***, значением которого является номер клетки, на которую идет конь.***

Установив список возможных ходов, шаблон вызывает «try-possible-moves» для выбора одного из них и делает этот ход.

Ниже представлен шаблон. Его параметры – номер данного хода (начинающийся с хода 2, потому что исходное положение коня считается ходом 1), состояние доски перед этим ходом и номер клетки, на которой конь находится в данный момент.

```
<xsl:template name="make-moves">
   <xsl:param name="move"/>
   <xsl:param name="board"/>
   <xsl:param name="square"/>
   <!-- определение возможных ходов коня -->
   <xsl:variable name="possible-move-list">
       <xsl:call-template name="list-possible-moves">
          <xsl:with-param name="board" select="$board"/>
```

```
            <xsl:with-param name="square" select="$square"/>
         </xsl:call-template>
      </xsl:variable>
      <!-- проверка этих ходов по очереди до тех пор, пока не будет найден подходящий -->
      <xsl:call-template name="try-possible-moves">
         <xsl:with-param name="board" select="$board"/>
         <xsl:with-param name="square" select="$square"/>
         <xsl:with-param name="move" select="$move"/>
         <xsl:with-param name="possible-moves" select="$possible-move-list/move"/>
      </xsl:call-template>

   </xsl:template>
```

Следующий обсуждаемый шаблон – «list-possible-moves». Он берет в качестве исходных данных текущее состояние шахматной доски и позиции коня и создает список клеток, на которые может перейти конь. Конь, находящийся в центре доски, может перейти на восемь различных клеток. Это клетки, удаленные от текущего ряда либо на два столбца и один ряд, либо на два ряда и один столбец, как показано на рис. 10.7.

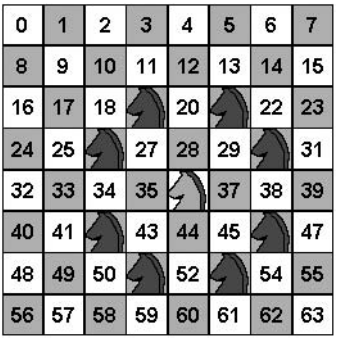


***Рис. 10.7.** Возможные ходы коня*

Однако следует рассматривать и случаи, когда некоторые из таких клеток недоступны, потому что они находятся за пределами доски, а кроме того, нужно исключать клетки, которые уже были посещены. Использованная здесь логика проста: шаблон по очереди анализирует каждую из восьми возможностей:

```
<xsl:template name="list-possible-moves">
   <xsl:param name="board"/>
   <xsl:param name="square"/>
   <xsl:variable name="row" select="$square div 8"/>
   <xsl:variable name="column" select="$square mod 8"/>
    <xsl:if test="$row &gt; 1 and $column &gt; 0
                  and $board/square[($square - 17) + 1]=''">
       <move><xsl:value-of select="$square - 17"/></move>
    </xsl:if>
    <xsl:if test="$row &gt; 1 and $column &lt; 7
                  and $board/square[($square - 15) + 1]=''">
```

```
            <move><xsl:value-of select="$square - 15"/></move>
        </xsl:if>
        <xsl:if test="$row &gt; 0 and $column &gt; 1
                      and $board/square[($square - 10) + 1]=''">
            <move><xsl:value-of select="$square - 10"/></move>
        </xsl:if>
        <xsl:if test="$row &gt; 0 and $column &lt; 6
                      and $board/square[($square - 6) + 1]=''">
            <move><xsl:value-of select="$square - 6"/></move>
        </xsl:if>
        <xsl:if test="$row &lt; 6 and $column &gt; 0
                      and $board/square[($square + 15) + 1]=''">
            <move><xsl:value-of select="$square + 15"/></move>
        </xsl:if>
        <xsl:if test="$row &lt; 6 and $column &lt; 7
                      and $board/square[($square + 17) + 1]=''">
            <move><xsl:value-of select="$square + 17"/></move>
        </xsl:if>
        <xsl:if test="$row &lt; 7 and $column &gt; 1
                      and $board/square[($square + 6) + 1]=''">
            <move><xsl:value-of select="$square + 6"/></move>
        </xsl:if>
        <xsl:if test="$row &lt; 7 and $column &lt; 6
                      and $board/square[($square + 10) + 1]=''">
            <move><xsl:value-of select="$square + 10"/></move>
        </xsl:if>

    </xsl:template>
```

Таким образом, определив возможные ходы, нужно выбрать один из них и сделать его. Это – задача шаблона `try-possible-moves`.

Этот шаблон довольно сложен, и стоит рассмотреть его внимательнее.

Сначала он проверяет, не является ли список возможных ходов пустым. Если да, то шаблон применяет маршрут `<xsl:otherwise>`, находящийся в конце тела шаблона, который выдает специальное значение «##», указывающее, что данная попытка обойти шахматную доску потерпела неудачу. Хотя уже говорилось, что алгоритм поиска клетки с наименьшим количеством выходов, фактически, всегда приводит к успеху, но следует подстраховаться.

В обычном случае есть один или более возможных ходов, и вызывается шаблон `find-best-move`, определяющий лучший из них (то есть ход на клетку с наименьшим количеством выходов). На всякий случай, если придется вернуться на несколько ходов назад, здесь также формируется список других возможных ходов, который содержит все ходы, кроме выбранного, чтобы при необходимости можно было им воспользоваться.

Далее делается выбранный ход, для чего вызывается шаблон `place-knight`, который обсуждался ранее. Он переводит коня на выбранную клетку и генерирует новое состояние шахматной доски.

А теперь можно повторять процесс, не забывая, что в XSLT обычно для повторения процесса производится рекурсия. Если на доске еще остались непосещенные клетки (идентифицируемые по пустым элементам `<square/>`), то вызывается шаблон «make-moves», описанный выше, чтобы вычислить остальную часть маршрута. Если непосещенных клеток не осталось, выводится значение конечного состояния шахматной доски. Это выходное значение передается через все 64 уровня рекурсии переменной «$final-board» в корневом шаблоне, а затем передается шаблону «print-board» для создания окончательного вида доски.

```
<xsl:template name="try-possible-moves">
   <xsl:param name="move"/>
   <xsl:param name="board"/>
   <xsl:param name="square"/>
   <xsl:param name="possible-moves"/>

   <xsl:choose>
   <xsl:when test="$possible-moves">

      <!-- если возможен по крайней мере один ход, ищем наилучший -->
      <xsl:variable name="best-move">
         <xsl:call-template name="find-best-move">
            <xsl:with-param name="board" select="$board"/>
            <xsl:with-param name="possible-moves" select="$possible-moves"/>
         </xsl:call-template>
      </xsl:variable>

      <!-- ищем список возможных ходов, исключая лучший -->
      <xsl:variable name="other-possible-moves"
                    select="$possible-moves[. != $best-move]"/>

      <!-- обновляем состояние доски, чтобы выполнить ход, выбранный как лучший -->
      <xsl:variable name="next-board">
         <xsl:call-template name="place-knight">
            <xsl:with-param name="move" select="$move"/>
            <xsl:with-param name="board" select="$board"/>
            <xsl:with-param name="square" select="$best-move"/>
          </xsl:call-template>
      </xsl:variable>

      <!-- выполняем последующие ходы до тех пор, пока не закончится доска -->
      <xsl:variable name="final-board">
         <xsl:choose>
         <xsl:when test="$next-board/square = ''">
                        <!-- if there is an empty square -->
            <xsl:call-template name="make-moves">
               <xsl:with-param name="move" select="$move + 1"/>
               <xsl:with-param name="board" select="$next-board"/>
               <xsl:with-param name="square" select="$best-move"/>
            </xsl:call-template>
         </xsl:when>
         <xsl:otherwise>
             <xsl:copy-of select="$next-board"/>
         </xsl:otherwise>
```

```
            </xsl:choose>
        </xsl:variable>

        <!-- если конечное состояние доски имеет значение '##', значит мы споткнулись
          и нужно выбрать следующий лучший из возможных ходов. Выполняется это при
              помощи рекурсивного вызова. На практике этого никогда не произойдет
              и по этому пути мы никогда не пойдем. -->
        <xsl:choose>
        <xsl:when test="$final-board='##'">
           <xsl:call-template name="try-possible-moves">
              <xsl:with-param name="board" select="$board"/>
              <xsl:with-param name="square" select="$square"/>
              <xsl:with-param name="move" select="$move"/>
              <xsl:with-param name="possible-moves" select="$other-possible-moves"/>
           </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
           <xsl:copy-of select="$final-board"/>
        </xsl:otherwise>
     </xsl:choose>
     </xsl:when>

     <xsl:otherwise>

     <!-- если возможных ходов не осталось, возвращаем особое значение '##'
          для конечного состояния доски, что говорит о том, что мы споткнулись -->
        <xsl:value-of select="'##'"/>
     </xsl:otherwise>
     </xsl:choose>
  </xsl:template>
```

Осталось еще обсудить шаблон «find-best-move», который из набора возможных ходов выбирает лучший – ход на клетку с наименьшим количеством выходов.

Как всегда, логика здесь рекурсивная. Все время учитывается лучший ход из уже проанализированных и количество выходов, имеющихся у выбранной на данный момент клетки лучшего хода. Если первый ход в списке (*пробный ход*) лучше, чем лучший ход из уже проверенных, то он заменяет предыдущий лучший, и снова вызывается шаблон для обработки оставшихся возможных ходов. В результате после обработки всего списка выявляется самый лучший ход.

Чтобы найти количество выходов с данной клетки, создается пробная шахматная доска и с помощью вызова шаблона «place-knight», описанного ранее, делается исследуемый ход. Затем для этой же доски вызывается шаблон «list-possible-moves», также описанный ранее, и определяется, какие ходы станут возможными после данного пробного хода. Здесь не важны детали, а только количество ходов, которое можно выяснить просто по длине списка.

Теперь можно вычислить две переменные: лучший ход на данный момент и наименьшее количество выходов, чтобы выяснить на основании этих данных, лучше ли пробный ход, чем предыдущий лучший. Если этот ход оказы-

вается лучшим на данный момент, он записывается в вывод. Наконец, шаблон «find-best-move» рекурсивно вызывает себя для обработки оставшихся в списке ходов. По завершении работы он выдает значение, являющееся лучшим ходом, то есть клеткой, на которую должен перейти конь.

```
<xsl:template name="find-best-move">
   <xsl:param name="board"/>
   <xsl:param name="possible-moves"/>
   <xsl:param name="fewest-exits" select="9"/>
   <xsl:param name="best-so-far" select="'XX'"/>

   <xsl:variable name="trial-move" select="number($possible-moves[1])"/>

   <xsl:variable name="other-possible-moves"
                 select="$possible-moves[position() &gt; 1]"/>

   <!-- пытаемся выполнить первый ход -->

   <xsl:variable name="trial-board">
      <xsl:call-template name="place-knight">
         <xsl:with-param name="board" select="$board"/>
            <xsl:with-param name="move" select="99"/>
            <xsl:with-param name="square" select="$trial-move"/>
         </xsl:call-template>
   </xsl:variable>

   <!-- проверяем, сколько ходов можно будет сделать в следующий раз -->

   <xsl:variable name="trial-move-exit-list">
      <xsl:call-template name="list-possible-moves">
         <xsl:with-param name="board" select="$trial-board"/>
         <xsl:with-param name="square" select="$trial-move"/>
      </xsl:call-template>
   </xsl:variable>

   <xsl:variable name="number-of-exits" select="count($trial-move-exit-list/move)"/>
   <!-- определяем, является ли этот пробный ход лучшим на данный момент -->
   <xsl:variable name="minimum-exits">
      <xsl:choose>
      <xsl:when test="$number-of-exits &lt; $fewest-exits">
         <xsl:value-of select="$number-of-exits"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="$fewest-exits"/>
      </xsl:otherwise>
      </xsl:choose>
   </xsl:variable>

<!-- определяем лучший ход (тот, который имеет наименьшее количество выходов)
     на данный момент -->
   <xsl:variable name="new-best-so-far">
      <xsl:choose>
      <xsl:when test="$number-of-exits &lt; $fewest-exits">
         <xsl:value-of select="$trial-move"/>
      </xsl:when>
      <xsl:otherwise>
```

```
            <xsl:value-of select="$best-so-far"/>
         </xsl:otherwise>
         </xsl:choose>
      </xsl:variable>

      <!-- если существуют другие возможные ходы, рассматриваем и их при помощи
           рекурсивного вызова. В противном случае возвращаем лучший найденный ход. -->

      <xsl:choose>
      <xsl:when test="$other-possible-moves">
         <xsl:call-template name="find-best-move">
            <xsl:with-param name="board" select="$board"/>
            <xsl:with-param name="possible-moves" select="$other-possible-moves"/>
            <xsl:with-param name="fewest-exits" select="$minimum-exits"/>
            <xsl:with-param name="best-so-far" select="$new-best-so-far"/>
         </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="$new-best-so-far"/>
      </xsl:otherwise>
      </xsl:choose>

   </xsl:template>
```

Вот и вся таблица стилей.

Выполнение таблицы стилей

Для запуска таблицы стилей загрузите ее с веб-сайта Wrox и примените к произвольному исходному документу (например, к самой себе). При работе с Saxon нужна такая команда:

```
saxon tour.xsl tour.xsl start=b6 >tour.html
```

Затем отобразите `tour.html` в окне броузера. Как именно передать глобальный параметр, зависит от реализации XSLT-процессора: если параметр не задан, маршрут коня начнется с клетки a1.

В написанном виде эта таблица стилей работает с процессором XSLT 1.1. Ее можно адаптировать для работы с любым процессором XSLT 1.0, который позволяет преобразовывать деревья в наборы узлов. Версия таблицы стилей, которая работает с MSXML3, используя функцию `msxsl:node-set()`, есть в файлах для загрузки под названием `ms-tour.xsl`. Она включает инструкцию обработки `<?xml-stylesheet?>`, ссылающуюся на себя, так что можно выполнить пример с поиском маршрута коня (с параметрами по умолчанию), просто дважды щелкнув по значку этого файла в проводнике Windows.

Наблюдения

Определение маршрута коня – не очень типичная таблица стилей, но она демонстрирует вычислительную мощь языка XSLT и в особенности значительную роль, которую играет рекурсия в любой таблице стилей, предназначенной для выполнения любых нетривиальных вычислений или обработки нет-

ривиальных структур данных. И хотя читателям не придется использовать XSLT для решения шахматных задач, им наверняка понадобится производить сложные вычисления, чтобы определить лучший вариант размещения набора изображений на странице или выяснить нужное количество столбцов для отображения списка телефонных номеров, или решить на основании пользовательских предпочтений, которая из сегодняшних новостей должна быть особо заметно выделена.

Итак, если читатели хотят знать, почему выбран этот пример, у автора есть два ответа: во-первых, это было интересно писать, а во-вторых, автор надеется, что это убедит всех в том, что на языке XSLT можно писать алгоритмы любой сложности.

Резюме

В этой главе представлены три полноценные таблицы стилей, все они по сложности подобны многим таблицам стилей, которые читателям понадобится создавать для реальных приложений. Для книги выбраны эти три примера, которые очень различны по характеру, отражая три образца проектирования, обсуждавшиеся в предыдущей главе, а именно:

- Таблица стилей, основанная на правилах, для преобразования документа, содержащего семантическую разметку, в HTML. В этой таблице стилей основная логика была направлена на создание правильного стиля отображения в HTML каждого XML-элемента и на формирование оглавлений, нумерации разделов и внутренних гиперссылок, с некоторой интересной логикой размещения данных в таблице.
- Навигационная таблица стилей для представления выбранной информации из иерархической структуры данных. Эта таблица стилей прежде всего касается прослеживания связей в структуре данных XML, и для достижения этого она использует мощные возможности выражений XPath. На примере этой таблицы стилей удалось исследовать некоторые аспекты применения XSLT: когда и где выполнять преобразование XML в HTML и как обрабатывать данные в форматах, не являющихся XML.
- Вычислительная таблица стилей для расчета результатов умеренно сложного алгоритма. Эта таблица стилей продемонстрировала, что даже довольно сложные алгоритмы можно программировать на языке XSLT, если овладеть методикой рекурсии. Такие алгоритмы намного проще реализовать в XSLT 1.1, чем в XSLT 1.0, благодаря возможности использовать временные деревья для сохранения промежуточных данных.

Для работы второй таблицы стилей пришлось использовать некоторые нестандартные интерфейсы, преодоставляемые различными реализациями XSLT. В приложениях к книге будет более детально рассмотрен ряд наиболее широко используемых процессоров XSLT.

A

Microsoft MSXML3

В этом приложении приводится информация по использованию продукта MSXML3 от Microsoft, который включает в себя процессор XSLT 1.0, интегрированный с синтаксическим анализатором XML и другими инструментами.

Продукт MSXML3 – это самая последняя разработка фирмы Microsoft в области XML- и XSLT-технологий.[1] Он включает в себя анализатор XML, поддерживающий интерфейс DOM, который является расширенной реализацией спецификации W3C DOM Recommendation, а также процессор XSLT, который соответствует спецификации W3C XSLT 1.0 Recommendation, и процессор XPath 1.0, который может использоваться совместно с таблицами стилей XSLT или непосредственно в документе DOM. MSXML3 также включает версию процессора схем XML, реализующего собственную версию схем XML, разработанную Microsoft (значительно отличающуюся от почти завершенной спецификации W3C для схем XML).

Это приложение **не** описывает первоначальную версию MSXML от 1998 года, которая поставлялась вместе с IE4, а позже с IE5 и IE5.5. Эти версии броузера использовали реализацию XSL, основанную на предварительных разработках спецификации, в том виде, в котором они тогда находились. Эта реализация включала много расширений от Microsoft. В этом диалекте XSLT 1998 года можно узнать XSLT 1.0 и XSLT 1.1, но различий в деталях так много, что его лучше всего рассматривать как отдельный язык. В этой книге я ссылаюсь на него как на **WD-xsl** (рабочий проект XSL, Working Draft XSL), потому что в таблицах стилей, использующих этот диалект, пространство имен объявляется следующим образом: «`xmlns:xsl="http://www.w3.org/`

[1] На сегодняшний день Microsoft анонсировала выход MSXML 4.0; за более подробной информацией обращайтесь на сайт *http://msdn.microsoft.com/xml/*. – *Примеч. перев.*

TR/WD-xsl"», вместо стандартного объявления «xmlns:xsl="http://www.w3.org/1999/XSL/Transform"».

В документации Microsoft ссылка на XSL обычно означает диалект WD-xsl. Когда же имеется в виду XSLT, то его так и называют.

На момент написания книги MSXML3 являлся серийным релизом и полностью поддерживался, однако он по-прежнему не поставлялся в качестве стандартного компонента других продуктов, таких как Internet Explorer, Windows 2000 или SQL Server. Все эти продукты выходили с более старыми версиями MSXML, которые поддерживали только диалект WD-xsl. Единственный способ получить MSXML3 – это загрузить его с Интернет-сайта Microsoft по адресу *http://msdn.microsoft.com/xml/*.

Данное приложение начинается с обзора содержимого таблицы стилей MSXML3: это не займет много времени, поскольку имеется очень близкое соответствие с XSLT 1.0. Затем будут продемонстрированы различные способы установки и работы этого продукта внутри броузера Internet Explorer, что и составит основной материал этой главы. После этого будет вкратце показано, как работать с MSXML на сервере, как правило, с ASP-страниц. И завершается приложение кратким обзором основных классов и методов, применяемых в MSXML API.

Версии MSXML

К моменту написания книги Microsoft выпустила несколько версий MSXML. Первоначальная бета-версия 1.0 была вскоре заменена версией 2.0, которая распространяется вместе с последними релизами Internet Explorer 5 и 5.5, а также с Windows 2000. Для этой версии существует полный пакет SDK, поставляемый в составе пакетов Internet Explorer и Windows SDK. В комплект программного обеспечения MSXML входят синтаксический анализатор XML и процессор XSLT. Несмотря на то что предметом данной книги является XSLT, данный продукт часто упоминается просто как «анализатор MSXML» (MSXML parser). Как уже упоминалось, в версии 2.0 реализовано нечто похожее на рабочую версию спецификации XSL от декабря 1998 года, дополненную собственными расширениями Microsoft, предназначенными для упрощения распространенных операций и обработки действий, не описываемых в спецификации W3C, таких как загрузка и сохранение XML-документов.

MSXML версии 3.0 был выпущен в марте 2000 года, за чем последовала серия релизов, в которых повышалась функциональность продукта, вплоть до серийного релиза в октябре 2000 года. Ожидается, что эта версия будет включена в стандартную поставку Internet Explorer 6. Как было отмечено выше, теперь это продукт с полноценной поддержкой Microsoft, но получить его можно лишь путем загрузки из Интернета.

MSXML3 поддерживает как синтаксис WD-xsl, так и синтаксис XSLT. Однако их нельзя смешивать, нужно использовать либо тот, либо другой ди-

алект. Эти диалекты можно различить по URI пространства имен, использованному в элементе <xsl:stylesheet>:

- Таблицы стилей WD-xsl используют для пространства имен такой URI:

```
http://www.w3.org/TR/WD-xsl
```

- Таблицы стилей XSLT используют для пространства имен следующий URI:

```
http://www.w3.org/1999/XSL/Transform
```

Если вы объявили пространство имен WD-xsl, то вы уже не можете использовать синтаксис XSLT, и наоборот. Многие присущие XSLT черты отсутствуют в WD-xsl, как то: переменные, именованные шаблоны, шаблоны значения атрибутов, ключи, элемент <xsl:import>, а также некоторые оси XPath. Кроме того, существует большая разница в синтаксисе и обозначениях.

Возможно, некоторые практические соображения, такие как наличие у всех ваших пользователей установленного IE5, вынудят вас писать таблицы стилей с использованием WD-xsl. Если вы так поступите, то вам следует быть готовыми к тому, что многие конструкции, описываемые в данной книге, у вас работать не будут. Помните также, что вы используете устаревшую технологию. Документация по WD-xsl уже не поставляется в комплекте с MSXML3; вы можете найти ее на Интернет-сайте Microsoft, но это не так-то просто, поскольку теперь Microsoft ставит на XSLT.

Таблицы стилей MSXML3

MSXML3 предлагает очень хороший уровень соответствия XSLT 1.0. Но в настоящее время в нем не поддерживаются все новые элементы синтаксиса XSLT 1.1, описанные в этой книге. Однако MSXML3 предоставляет собственные решения, например:

- Нельзя непосредственно использовать временное дерево (или фрагмент конечного дерева в терминах XSLT 1.0) в качестве набора узлов. Можно использовать переменную, которая ссылается на временное дерево в любом контексте, где можно использовать строки, однако там, где контекст подразумевает набор узлов, это не разрешено. К примеру, нельзя написать «$дерево/элемент», <xsl:apply-templates select="$дерево"> или <xsl:for-each select="$дерево">. В этих случаях можно сделать следующее: использовать функцию расширения msxsl:node-set() для явного преобразования дерева в набор узлов.
- Нельзя сформировать несколько выходных файлов при помощи элемента <xsl:document>. Возможно, вследствие того что MSXML3 разрабатывался для использования внутри броузера, он не поддерживает механизм создания нескольких выходных файлов. Это препятствие можно обойти, если произвести несколько преобразований над одним и тем же исходным файлом, используя параметр для определения той части выходных дан-

ных, которую нужно получить в каждом преобразовании. Или же можно без особого труда написать функцию расширения на языке JScript, которая будет вызываться из таблицы стилей. Эта функция может принимать в качестве параметра фрагмент конечного дерева и сериализовывать его в файле как XML.

- Нельзя использовать элемент <xsl:script> для объявления внешних функций. В то же время, MSXML3 поддерживает элемент <msxsl:script> с почти идентичной функциональностью. Процессор от Microsoft может вызывать внешние функции, написанные на любом из типичных языков сценариев Microsoft, таких как JScript или VBScript, но не может непосредственно вызывать методы Java. Однако можно вызвать любой COM-объект, если написать соответствующую функцию-оболочку на JScript.

Реализация XSLT от Microsoft очень близка к стандарту XSLT 1.0, поэтому нет нужды говорить что-то еще на эту тему в данном приложении. Однако в этом разделе мы рассмотрим некоторые ограничения и расширения, а также те области, в которых Microsoft использовала ту свободу выбора, которую стандарт предоставляет разработчикам.

Расширяемость

В MSXML3 отсутствует поддержка элементов расширения.

Функции расширения можно реализовать, написав соответствующие модули на VBScript или JScript и разместив их внутри элемента верхнего уровня <msxsl:script>. Этот элемент похож на элемент <xsl:script>, описанный в рабочем проекте XSLT 1.1.

Ниже приведен пример таблицы стилей, использующей такое расширение:

Пример: Использование VBScript в таблице стилей MSXML3

В этом примере приводится таблица стилей, которая преобразует дюймы в миллиметры.

Исходный документ

Исходный файл называется дюймы.xml. Для того чтобы осуществить преобразование, нужно выполнить двойной щелчок на значке файла в Проводнике Windows.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="в-миллиметры.xsl"?>
<размер>
Размер картинки <дюймы>5</дюймы> на
<дюймы>12</дюймы>
</размер>
```

Таблица стилей

Таблица стилей содержится в файле `в-миллиметры.xsl`.

Она содержит простую функцию, написанную на VBScript, внутри элемента `<msxsl:script>`. Эта функция вызывается из шаблонного правила для элемента `<дюймы>` как функция расширения.

```
<xsl:stylesheet
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       version="1.0"
       xmlns:extra="urn:extra-functions"
>
<msxsl:script xmlns:msxsl="urn:schemas-microsoft-com:xslt"
        language="VBScript"
        implements-prefix="extra"
>
Function ToMillimetres(inches)
   ToMillimetres = inches * 25.4
End Function
</msxsl:script>
<xsl:output method="html"/>
<xsl:template match="/" >
<html><body><p>
   <xsl:apply-templates/>
</p></body></html>
</xsl:template>
<xsl:template match="дюймы">
   <xsl:text> </xsl:text>
   <xsl:value-of select="format-number(extra:ToMillimetres(number(.)), '0.00')"/>
   <xsl:text>мм </xsl:text>
</xsl:template>
</xsl:stylesheet>
```

Результат

В броузере отобразится следующий текст:

```
Размер картинки 127.00мм на 304.80мм.
```

Такие сценарии могут вызвать объекты COM, указанные обычным способом в системном реестре. Однако при запуске таблицы стилей в броузере пользовательские настройки безопасности могут помешать созданию объекта на стороне клиента.

Не следует путать сценарий, вызываемый из таблицы стилей, со сценарием, который является частью HTML-страницы, вызывающей таблицу стилей, а также со сценарием, который составляет часть HTML-страницы, сгенерированной при помощи таблицы стилей. Если вы используете все три вида сценариев, то вам нужно уметь четко разделять их. Для функций, которые должны вызываться из выражений XPath, имеющихся в таблице стилей, следует использовать только элемент `<msxsl:script>`. Если же таблица стилей

содержит конечный литеральный элемент <script>, то внутри него должен располагаться код, который будет частью формируемой HTML-страницы, – этот код не будет исполняться во время преобразования документа.

Функция msxsl:node-set()

MSXML3 соответствует стандарту XSLT 1.0, и потому придерживается ограничений, накладываемых в XSLT 1.0 на использование переменных, чье значение представляет собой дерево (фрагменты конечного дерева). Эти ограничения применяются к переменным, значения которых определяются конструкцией, подобной следующей:

```
<xsl:variable name="дерево">
   <тег>значение</тег>
</xsl:variable>
```

В частности, переменная, значением которой является дерево, не может использоваться:

- в выражениях пути, таких как «$дерево/тег»
- в выражении выборки <xsl:for-each>
- в выражении выборки <xsl:apply-templates>

Чтобы преодолеть такие ограничения, MSXML3, подобно многим другим процессорам XSLT 1.0, предоставляет функцию расширения, которая преобразует дерево в набор узлов. Конечный набор узлов содержит всего один узел, а именно корень дерева. Поэтому можно написать:

- <xsl:value-of select="count(msxsl:node-set($дерево)/тег)"/>
- <xsl:for-each select="msxsl:node-set($дерево)">
- <xsl:apply-templates select="msxsl:node-set($дерево)"/>

URI пространства имен функции node-set() (так же, как и элемента <msxsl:script>) – xmlns:msxsl="urn:schemas-microsoft-com:xslt". Конечно, можно было бы использовать любой другой префикс пространства имен, лишь бы ему соответствовал корректный URI.

Системные свойства

Функция system-property() в MSXML3 в настоящее время возвращает следующие значения:

Название системного свойства	Значение
xsl:version	1
xsl:vendor	Microsoft
xsl:vendor-uri	http://www.microsoft.com
msxsl:version	3

Ограничения

Фирма Microsoft заявляет о полном соответствии своего продукта стандартам XSLT 1.0 и XPath 1.0. На данный момент не производится независимого тестирования XSLT-процессоров на соответствие стандарту, поэтому остается лишь принять заявления поставщиков о соответствии на веру. А на деле это означает, что любое найденное несоответствие является ошибкой и, по всей видимости, должно быть исправлено.

Существует несколько нечетких мест, где соответствие стандарту не настолько строгое, как того бы хотелось. Среди них:

- Обработка пробельных узлов. Обычно данные подаются на вход XSLT-процессора Microsoft в форме DOM, а по умолчанию в процессе анализа текста для формирования DOM в MSXML3 удаляются все текстовые узлы, состоящие из пробельных символов. Как следствие, элемент таблицы стилей `<xsl:preserve-space>` не оказывает желаемого воздействия, поскольку к тому моменту, когда XSLT-процессор начинает просмотр данных, там уже нет пробельных узлов, которые можно было бы сохранить. Чтобы добиться согласованного со стандартом поведения, нужно перед загрузкой документа установить свойство `preserveWhitespace` объекта `DOMDocument` в значение `True`. То же самое относится к таблице стилей: если вы хотите использовать элемент `<xsl:text>` для управления выводом пробельных символов, в частности, когда выходной формат чувствителен к пробелам, например формат значений, разделенных запятыми, то нужно загружать таблицу стилей со свойством `preserveWhitespace`, установленным в `True`.
- Нормализация текстовых узлов. В спецификациях XSLT и XPath обозначено, что смежные текстовые узлы всегда сливаются в один узел. MSXML3 использует в качестве внутренней структуры данных модель DOM, которая не накладывает такого правила. Хотя MSXML3 хорошо справляется с созданием корректного представления исходного дерева DOM в XPath, данный случай является примером неполноты такого представления. Вот два общих случая, когда соседние текстовые узлы не сливаются: во-первых, когда один из узлов представляет собой содержимое секции `CDATA` исходного XML-документа, а во-вторых, когда один из них является замененным текстом ссылки на сущность (отличной от встроенных ссылок на сущности, таких как `«&lt;»`). Поэтому становится опасным использование таких конструкций, как `<xsl:value-of select="text()"/>`, поскольку MSXML3 вернет только первый из текстовых узлов, то есть тот текст, который расположен перед началом сущности или границей секции `CDATA`. Безопаснее выводить значение элемента при помощи конструкции `<xsl:value-of select="."/>`.
- Инструкция `<xsl:message>` не оказывает воздействия, если не указать `«terminate="yes"»`.

Установка MSXML3

Текущая версия продукта MSXML3 доступна на странице загрузки на Интернет-сайте Microsoft. Чтобы загрузить ее, зайдите по следующему адресу:

http://msdn.microsoft.com/downloads/webtechnology/xml/msxml.asp

После того как файл `msxml3.exe` загружен на ваш компьютер, просто запустите его для установки продукта.

Вы, скорее всего, захотите также установить пакет разработки программного обеспечения, SDK, хотя бы для того, чтобы получить документацию. Этот пакет доступен на той же веб-странице. Он распространяется в виде исполняемого файла `xmlsdk.exe` и устанавливается также путем загрузки и запуска файла. Документация представлена в формате откомпилированного файла помощи HTML – `xmlsdk.chm`, который по умолчанию устанавливается в каталог `c:\Program Files\Microsoft XML Parser SDK\Docs` и открывается двойным щелчком в Проводнике Windows.

По умолчанию MSXML3 работает в **параллельном** режиме со старой версией MSXML, которая уже установлена на вашей машине вместе с IE5 или IE5.5. Это означает, что новый процессор будет использоваться только в тех случаях, когда он явно вызывается приложениями, и в частности, не будет использоваться по умолчанию в Internet Explorer. Для работы с XSLT вы, возможно, захотите установить MSXML3 в режиме **замещения**: в этом случае все вызовы стандартного анализатора из приложений, подобных Internet Explorer, будут переадресовываться к MSXML3. В этом случае нужно запустить небольшую утилиту `xmlinst.exe`, которую можно отдельно загрузить с сайта Microsoft MSDN.

> ***Не следует устанавливать MSXML3 в режиме замещения на промышленный сервер, где используются такие продукты, как SQL Server или Biztalk Server. Эти продукты рассчитаны на использование в качестве процессора по умолчанию MSXML2.***

Также на этом Интернет-сайте находится страница, на которой много полезной информации о стратегии и инициативах Microsoft в области XML, ее адрес *http://msdn.microsoft.com/xml/default.asp*. Там же вы найдете примеры, демонстрационные продукты и статьи по XML и смежным технологиям.

Страница загрузки также содержит ссылки на некоторые инструменты, которые могут показаться вам полезными:

- Инструменты Internet Explorer, предназначенные для проверки действительности XML и просмотра результата преобразования XSLT. Эти средства встроены в Internet Explorer и позволяют выполнять по щелчку правой кнопки мыши два действия: проверку действительности исходного XML-файла и просмотр результата XSLT-преобразования. Средство просмотра еще будет упомянуто позже в этом приложении.
- Расширение Microsoft XSL ISAPI. Это расширение для веб-сервера облегчает XML/XSL-преобразования на стороне сервера. Оно может автомати-

чески применять таблицу стилей на сервере, позволяя выбирать альтернативные таблицы стилей в зависимости от типа броузера. Оно управляет кэшированием таблицы стилей для увеличения производительности сервера, а также позволяет указать кодировку формируемого документа и предоставляет возможность настройки сообщений об ошибках. Однако, к сожалению, это расширение не различает, какой из процессоров установлен в броузере – MSXML2 или MSXML3.

- Таблица стилей XSL для схем XML (XML Schemas). Эта таблица стилей (использующая WD-xsl) может применяться при создании документации для XML-схем, основанной на том же синтаксисе. Схемы XML не обсуждаются подробно в этой книге. В спецификации XSLT они еще не используются, так как формирование стандарта для них еще не завершено (в настоящий момент это кандидат в рекомендации). Тем не менее, Microsoft реализовала свою собственную версию XML-схем, и MSXML использует ее как альтернативу DTD.
- Конвертер из XSL в XSLT. Эта таблица стилей преобразует таблицы стилей, написанные на WD-xsl к синтаксису, соответствующему XSLT, тем самым делая их пригодными для использования с MSXML3 и другими процессорами. Этот конвертер не сделает стопроцентного преобразования, но устранит все основные различия.

> ***Заметьте, что вам следует установить Internet Explorer (IE5 или IE5.5) для того, чтобы MSXML3 мог полноценно функционировать. Это нужно сделать даже в том случае, когда вы собираетесь использовать MSXML3 только на стороне сервера.***

Идентификаторы ProgID и ClassID в MSXML

Вместо того чтобы присвоить одинаковые имена файлам, а также значения идентификаторам ClassID и ProgID для всех трех версий MSXML, фирма Microsoft использовала разные значения для того, чтобы эти процессоры можно было устанавливать и использовать в параллельном режиме:

Версия анализатора	Имя DLL	ProgID и ClassID

MSXML 2.0	`msxml.dll`	**ProgID:**
		`Microsoft.XMLDOM` **или** `MSXML.DOMDocument`
		
		ClassID:
		`{2933bf90-7b36-11d2-b20e-00c04f983e60}`

MSXML 2.6	`msxml2.dll`	**ProgID:**
		`MSXML2.DOMDocument`
		
		ClassID:
		`{f6d90f11-9c73-11d3-b32e-00c04f990bb4}`

Версия анализатора	Имя DLL	ProgID и ClassID
MSXML 3.0	msxml3.dll	ProgID: MSXML2.DOMDocument.3.0 ClassID: {f5078f32-c551-11d3-89b9-0000f81fe221}

Подобная терминология может вас запутать. Приведенные объекты используют название "DOMDocument", поскольку созданный объект будет хранить представление в памяти XML-документа, соответствующее версии Microsoft для объектной модели документа (W3C Document Object Model), DOM. Объект Document имеет метод load, который позволяет создать документ в памяти из исходного XML-документа. Для этого нужно, конечно, произвести синтаксический анализ данных XML. Анализатор XML является частью функциональности объекта Document, а отдельного объекта Parser нет. В присваивании имен Microsoft придерживается следующего принципа: объект получает имя в соответствии с тем, какую информацию он содержит (существительное), а не тем, какую функцию он выполняет (глагол).

В приведенной выше таблице указаны идентификаторы ProgID и ClassID для запуска анализатора в однопоточных апартаментах (normal apartment-threaded mode). Можно также запускать анализатор в многопоточных апартаментах (free-threaded mode), используя альтернативные ProgID и ClassID во время создания экземпляра компонента (заметьте, что идентификаторы ClassID для разных режимов отличаются только в восьмом символе). Возможно, вы захотите поступить именно так, если запускаете MSXML3 на сервере, поскольку часто используемые таблицы стилей могут кэшироваться в объекте Application.

Версия анализатора	ProgID и ClassID для многопоточного режима
MSXML 2.0	ProgID: Microsoft.FreeThreadedXMLDOM или MSXML.FreeThreadedDOMDocument ClassID: {2933bf91-7b36-11d2-b20e-00c04f983e60}
MSXML 2.6	ProgID: MSXML2.FreeThreadedDOMDocument ClassID: {f6d90f12-9c73-11d3-b32e-00c04f990bb4}
MSXML 3.0	ProgID: MSXML2.FreeThreadedDOMDocument.3.0 ClassID: {f5078f33-c551-11d3-89b9-0000f81fe221}

В файле помощи из набора MSXML3 SDK в подразделе «GUID and ProgID Information» (который находится в разделе «XML Developer's Guide/XML DOM User Guide») можно найти полный список идентификаторов ProgID и ClassID для других объектов, реализованных в MSXML3, таких как XMLHTTP или XSLTemplate, а также для элемента управления привязками для данных XMLDSO.

Поскольку наличие многих версий MSXML создает такую неразбериху, то в файл загрузки для этого приложения включена HTML-страница msxml-version.html. Когда эта страница будет отображена в Internet Explorer, она покажет, какая версия используется по умолчанию на вашей машине. Эта страница использует код, написанный Ярно Эловирта (Jarno Elovirta) на JScript. Можно воспользоваться усовершенствованным вариантом, который отображает информацию обо всех версиях анализатора, установленных на вашем компьютере, по адресу *http://www.vbxml.com/parsers/sniffer/default.asp*.

Запуск MSXML в режиме замещения

Я уже упоминал об утилите xmlinst.exe. Эта программа может применяться для того, чтобы изменить настройки системного реестра, касающиеся использования различных версий анализатора MSXML. Запуск этой программы без параметров (просто загрузите ее в любой каталог и дважды щелкните на ней мышью) приведет к таким изменениям в реестре, что используемый по умолчанию ProgID «MSXML.DOMDocument» будет запускать последнюю версию анализатора, а не версию 2.0. Это может оказаться полезным, если в коде ваших приложений используется это значение ProgID. В подразделе «Running MSXML3 in Replace Mode» (в разделе «XML Developer's Guide/XML DOM User Guide») приведен список аргументов командной строки, которые могут быть использованы вместе с xmlinst.exe для других изменений в реестре.

Если вы не запустите xmlinst.exe, то Internet Explorer будет по-прежнему использовать старую версию анализатора MSXML, если только вы не используете сценарии, в которых явно указано применять новую версию. В результате (как описано на стр. 763), если дважды щелкнуть на XML-документе, инструкция обработки <?xml-stylesheet?> которого ссылается на таблицу стилей XSLT, то старый анализатор не воспримет таблицу и не отобразит на выходе никакой полезной информации. Это распространенная ситуация, поэтому, если вы используете XSLT на клиентской машине, то я советую установить MSXML3 в качестве анализатора по умолчанию при помощи xmlinst.exe. Это не помешает осуществлять преобразования при помощи WD-xsl, так как MSXML3 поддерживает оба диалекта.

Конечно, если вы применяете таблицы стилей на своем веб-сайте, то вам нужно быть уверенными, что у всех посетителей вашего сайта установлен MSXML3. Как этого добиться выходит за рамки данной книги, но вы можете найти информацию о распространении данного продукта на Интернет-сайте Microsoft.

С другой стороны, устанавливать MSXML3 в качестве анализатора по умолчанию на сервере не совсем верное решение, особенно, если на нем функционируют SQL Server или BizTalk Server. Дело в том, что для работы этих продуктов требуется MSXML2. Лучше всего использовать MSXML3 на сервере в параллельном режиме (что означает сосуществование нескольких версий процессора) и при необходимости вызывать его явно при помощи ProgID «MSXML2.DOMDocument.3.0» или «MSXML2.FreeThreadedDOMDocument.3.0».

Использование таблицы стилей по умолчанию

Если дважды щелкнуть на значке XML-документа, с которым не связана таблица стилей, то в Internet Explorer этот документ будет отображен непосредственно в виде развертываемого дерева. Таким образом можно легко проверить документ и увидеть его структуру (рис. А.1).

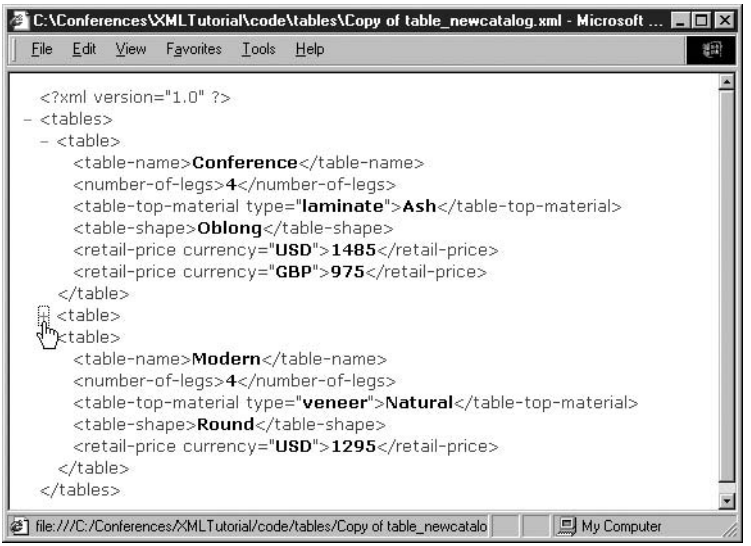


Рис. А.1. *Отображение в Internet Explorer XML-документа, с которым не связана таблица стилей*

Так происходит вследствие применения к документу таблицы стилей по умолчанию, поэтому то, что вы видите, есть результат преобразования, задаваемого этой таблицей стилей. Эта таблица стилей используется только в том случае, когда не задано никакой другой. В следующем разделе будет показано, как задать желаемую таблицу стилей для преобразования.

Если вы загрузите документ, который не является корректным, то в броузере вы увидите сообщение об ошибке (рис. А.2).

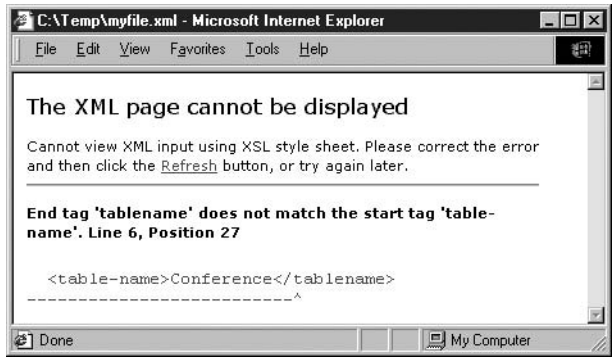


***Рис. А.2.** Сообщение об ошибке для некорректного XML-документа*

Чтобы посмотреть содержимое таблицы стилей, применяемой по умолчанию, введите в адресной строке следующий URL: `res://msxml.dll/DEFAULTSS.XSL` (тот же результат получается для URL `res://msxml3.dll/DEFAULTSS.XSL`, соответствующего анализатору MSXML3).

Эта таблица стилей написана на диалекте WD-xsl. Определить это можно по URI пространства имен «`http://www.w3.org/TR/WD-xsl`», а также по устаревшим элементам XSL, таким как `<xsl:entity-ref>` и `<xsl:node-name>`.

Присвоение документу таблицы стилей XSLT

Самый простой способ отображения результата XSLT-преобразования – это указать нужную таблицу стилей в инструкции обработки XML. Вот простейший синтаксис такого указания:

```
<?xml-stylesheet type="text/xsl" href="url_таблицы_стилей"?>
```

Например, если у вас есть подходящая таблица стилей с именем `каталог-столов.xsl`, расположенная в том же каталоге, что и XML-документ, то можно написать:

```
<?xml-stylesheet type="text/xsl" href="каталог-столов.xsl"?>
```

Эта инструкция должна идти после XML-объявления `<?xml version="1.0"?>`, а также любого объявления `<!DOCTYPE>`, но перед открывающим тегом элемента документа. Например, таким образом:

```
<?xml version="1.0" ?>
<!DOCTYPE tables SYSTEM "tables.dtd">
<?xml-stylesheet type="text/xsl" href="каталог-столов.xsl"?>
<столы>
. . .
</столы>
```

Результат может выглядеть приблизительно следующим образом:

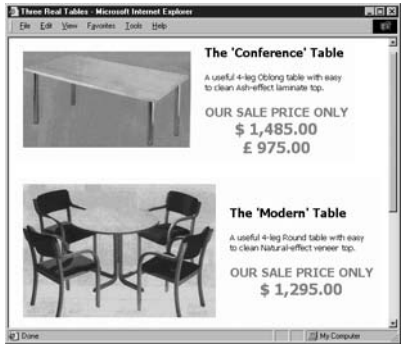


***Рис. А.3.** Отображение в Internet Explorer XML-документа с присвоенной таблицей стилей*

MSXML3 учитывает только первую инструкцию обработки `<?xml-stylesheet?>` в XML-документе, в которой присутствует `«type="text/xsl"»`, что указывает на таблицу стилей XSLT или WD-xsl. Можно, однако, отображать XML-документы с помощью каскадных таблиц стилей (Cascading Stylesheets, CSS) и инструкций `<?xml-stylesheet?>`, указывающих на эти таблицы стилей (где присутствует `«type="text/css"»`), которых в документе может быть любое количество. Internet Explorer выберет таблицу стилей XSL, если таковая имеется, в противном случае он соберет воедино таблицы стилей CSS, точно так же, как это делается при наличии нескольких элементов `<LINK>`, указывающих на таблицы стилей CSS в HTML-документе.

Полный список атрибутов, которые могут применяться в инструкции обработки `<?xml-stylesheet?>`, приведен в главе 3, но для таблиц стилей XSLT Internet Explorer проигнорирует все, кроме атрибутов `type` и `href`. Атрибут `type` должен иметь значение `«text/xsl»`, даже в том случае, если это не тип MIME, рекомендованный консорциумом W3C.

Отладка таблиц стилей

Если таблица стилей не является корректным XML-документом, то Internet Explorer отобразит сообщение об ошибке, как показано в примере на стр. 763. Такие сообщения об ошибках, как правило, указывают на строку, в которой содержится ошибка, поэтому при определенных познаниях в XML-терминологии не составляет труда исправить ошибку.

Если таблица стилей является корректным XML-документом, но не соответствует правилам XSLT, то будет также получено сообщение об ошибке, однако на этот раз оно не будет таким информативным. К примеру, если было неправильно набрано имя инструкции XSLT, то сообщение об ошибке может выглядеть как на рис. А.4.

В этом случае номер строки с ошибкой не указывается, поэтому для того, чтобы найти ошибку, придется просмотреть всю таблицу стилей. Откровен-

Рис. А.4. *Сообщение об ошибке в таблице стилей, вызванной нарушением правил XSLT*

но говоря, по мере увеличения размера таблиц стилей это начинает причинять все больше неудобств. На этой стадии может показаться, что разумнее использовать другой процессор, например Saxon или Xalan, только потому, что отладка с ними легче. На самом деле каждый процессор иногда выдает не совсем внятные сообщения об ошибках, и в таких случаях второе мнение будет наилучшим способом выявления ошибки.

Кроме того, могут быть ситуации, когда вы убеждены, что MSXML3 формирует неправильные выходные данные. Тогда преобразование с помощью другого процессора, как правило, покажет, что вопреки ожиданиям сгенерированные данные верны. Однако существует очень маленькая вероятность того, что преобразование при помощи двух процессоров приводит к разным результатам, что подтвердит ваши подозрения.

Еще одним полезным инструментом может послужить интерфейс `msxsl.exe` от Microsoft, который позволяет осуществлять преобразования из командной строки. Эту программу можно скачать с сайта MSDN; размер файла составляет меньше 60 Кбайт, поэтому стоит попробовать это сделать. Преобразование можно выполнить при помощи команды, подобной следующей:

```
MSXML source.xml style.xsl -o out.xml param=value
```

К сожалению, и здесь диагностика достаточно примитивна, и приведенное ниже сообщение является типичным примером (такое сообщение было вызвано ошибкой в таблице стилей, состоящей из 400 строчек):

```
Error occurred while compiling stylesheet 'test75.xsl'.
Code:   0x80004005
Expected token 'eof' found 'NUMBER'.
1.2-->.2<--
```

В конечном счете вы добьетесь, чтобы ваша таблица стилей стала синтаксически верной. Но это еще не все, после первой попытки вы можете и не получить требуемого результата. Основной причиной этого является то, что язык XSLT является весьма терпимым: когда вы пишете «`select="заголовок"`», а имеете в виду «`select="3аголовок"`» или «`select="'заголовок'"`», то в выходные данные ничего не попадет, так как элемента `<заголовок>` в исходном документе нет.

В первый момент это может побудить вас выбрать View Source в контекстном меню броузера, однако это не принесет собой пользы, так как будут отображены исходные XML-данные. То, что вам нужно – это увидеть результат применения таблицы стилей в формате HTML.

Вот тут-то и пригодится утилита Microsoft для просмотра результата XSLT-преобразования. Я уже упоминал о ней раньше; это один из инструментов, которые можно загрузить с сайта MSDN. Его установка достаточно необычна (вы должны выполнить щелчок правой кнопкой мыши на значке файла с расширением `.inf` и выбрать `install`), тем не менее, если буквально следовать инструкции, то все получится. Установив эту утилиту, вы сможете по щелчку правой кнопки мыши в Internet Explorer увидеть в меню пункт View XSL Output (Просмотр результата XSL-преобразования). При выборе этого пункта выдается листинг HTML-кода, который сформирован при помощи таблицы стилей и который пытается отобразить броузер.

Если и просмотр HTML-кода не дал результата, то полезно будет вставить диагностику в виде комментариев. Например, можно добавить в начало каждого шаблонного правила следующую инструкцию:

```
<xsl:comment>Входим в шаблонное правило для элемента xyz</xsl:comment>
```

Эти комментарии не отображаются в окне броузера, но их можно видеть при помощи средства просмотра результата XSL-преобразования. (С другими XSLT-процессорами можно использовать `<xsl:message>`, но в MSXML3 этот элемент не оказывает воздействия, если только не установлена жесткая опция `«terminate="yes"»`).

Если обнаружить ошибку по-прежнему не удается, то я бы порекомендовал установить некоторое дополнительное программное обеспечение. MSXML3 – замечательный продукт для преобразований с помощью таблиц стилей на стороне клиента, однако это – не самая лучшая среда разработки. Одним из преимуществ в области XSLT является наличие большого количества свободно распространяемых программ, остается лишь пользоваться ими. Xalan, например, имеет интерактивный отладчик, а в Saxon есть полезная опция для отслеживания выполняемого преобразования.

Управление обработкой XSLT при помощи сценария на стороне клиента

Использование инструкции обработки `<?xml-stylesheet?>` – это лишь один из способов применения таблицы стилей к XML-документу. Альтернативным способом является использование сценария, находящегося в HTML-странице, который явно загружает XML-документ и таблицу стилей и затем вызывает преобразование. При таком способе начинают играть роль вышеупомянутые идентификаторы `ProgID` и `ClassID`.

Создание экземпляра документа MSXML

Для создания экземпляра документа DOM, который бы использовал анализатор MSXML3, можно использовать несколько подходов.

В VBScript:

```
Dim doc
Set doc = CreateObject("MSXML2.DOMDocument.3.0")
```

В JScript:

```
var doc = new ActiveXObject('MSXML2.DOMDocument.3.0');
```

В Visual Basic нужно сначала добавить ссылку на DLL компонента при помощи диалога Project | References. В приведенном ниже снимке экрана для диалогового окна можно увидеть три версии анализатора:

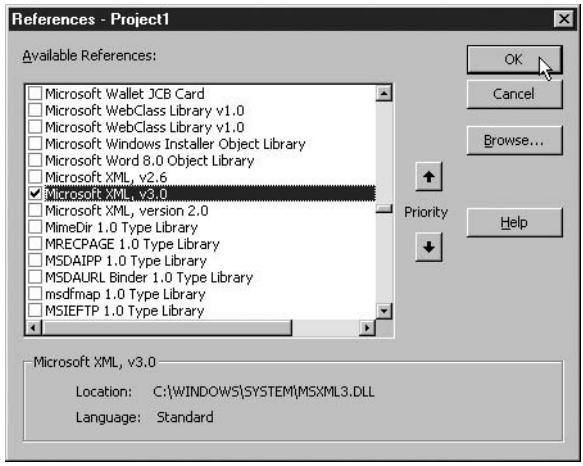


***Рис. А.5.** Диалоговое окно References в Visual Basic*

После этого можно использовать метод `CreateObject`:

```
Dim doc
Set doc = CreateObject("MSXML2.DOMDocument.3.0")
```

Однако более распространенным методом является использование ключевого слова `New`, что позволяет видеть в среде разработки всплывающую подсказку по синтаксису и список членов.

```
Dim doc As New DOMDocument3.0
```

Загрузка XML- и XSLT-документов

После создания объекта `Document` можно использовать характерные для Microsoft методы загрузки содержимого. Для загрузки XML-файла воспользуйтесь методом `load`:

```
'в VBScript или Visual Basic
doc.Load "c:\temp\myfile.xml"

//в JScript
doc.load('c:\temp\myfile.xml');
```

Если XML-данные хранятся в строке, а не в файле (как в случае чтения XML-данных из реляционной базы данных при помощи SQL или ADO), то можно воспользоваться методом `loadXML`:

```
'в VBScript или Visual Basic:
Dim strXML
strXML = "<?xml version='1.0'?><test><testitem/></test>"
doc.LoadXML strXML

//в JScript
var strXML = '<?xml version="1.0"?><test><testitem/></test>';
doc.loadXML(strXML);
```

После этого можно устанавливать любые нужные вам свойства MSXML3-анализатора. Например, полезно установить свойство `async` в значение `False`, чтобы преобразование началось только после окончания синтаксического анализа документа. Однако для повышения производительности, можно загружать исходный документ и таблицу стилей параллельно, оставив для этого свойства значение по умолчанию, то есть `True`. Это также позволяет информировать пользователя о ходе загрузки документа. Пример того, как использовать асинхронную загрузку, приведен далее на стр. 771.

Можно также включить в анализаторе проверку действительности. По умолчанию при загрузке и синтаксическом анализе анализатор проверяет только корректность XML-документа. При установке свойства `validateOnParse` в значение `True` можно заставить анализатор проверять действительность документа в соответствии с любой указанной схемой или DTD.

```
'в VBScript или Visual Basic
doc.ValidateOnParse = True

//в JScript
doc.validateOnParse = true;
```

Проверка на наличие ошибок загрузки, корректности и действительности документа

После попытки анализатора загрузить документ можно проверить, не было ли ошибок, изучив свойства объекта `parseError`, который создается объектом `DOMDocument`. Объект `parseError` имеет семь свойств, которые соответствуют последней произошедшей ошибке:

Свойство	Описание
`errorCode`	Стандартный код произошедшей ошибки.
`filepos`	Позиция символа, где была обнаружена ошибка, в файле.

Свойство	Описание
line	Номер строки, где была обнаружена ошибка.
linepos	Позиция символа, где была обнаружена ошибка, в этой строке.
reason	Текст с описанием ошибки.
srcText	Текст строки исходного кода, где была обнаружена ошибка.
url	URL или путь загружаемого файла.

Проверив значение свойства errorCode, можно понять, произошла ли ошибка. Отличное от нуля значение говорит о том, что анализатор обнаружил ошибку. Следующие примеры кода показывают, как извлечь и отобразить подробности об ошибке.

В VBScript или Visual Basic:

```
'проверяем, не было ли ошибки при загрузке
If doc.parseError.errorCode <> 0 Then 'создаем сообщение об ошибке
  Dim strError
  strError = "Неправильный XML-файл!" & vbCrlf _
           & "URL файла: " & doc.parseError.url & vbCrlf _
           & "Номер строки: " & doc.parseError.line & vbCrlf _
           & "Символ: " & doc.parseError.linepos & vbCrlf _
           & "Позиция в файле: " & doc.parseError.filepos & vbCrlf _
           & "Исходный текст: " & doc.parseError.srcText & vbCrlf _
           & "Код ошибки: " & doc.parseError.errorCode & vbCrlf _
           & "Описание: " & doc.parseError.reason
  MsgBox strError 'отображаем сообщение об ошибке
Else
  'загрузка прошла без ошибок - продолжаем обработку
  ...
End If
```

В JScript:

```
...
// проверяем, не было ли ошибки при загрузке
if (doc.parseError.errorCode != 0) { // создаем сообщение об ошибке
   var strError = new String;
   strError = 'IНеправильный XML-файл!\n'
            + 'URL файла: ' + doc.parseError.url + '\n '
            + 'Номер строки: ' + doc.parseError.line + '\n '
            + 'Символ: ' + doc.parseError.linepos + '\n '
            + 'Позиция в файле: ' + doc.parseError.filepos + '\n '
            + 'Исходный текст: ' + doc.parseError.srcText + '\n '
            + 'Код ошибки: ' + doc.parseError.errorCode + '\n '
            + 'Описание: ' + doc.parseError.reason;
   alert(strError);  // отображаем сообщение об ошибке
   }
else {
   // загрузка прошла без ошибок - продолжаем обработку
   ...
}
```

В обоих случаях получается одинаковый результат; ниже приведено сообщение об ошибке, возникающей при попытке загрузить документ, не являющийся корректным:

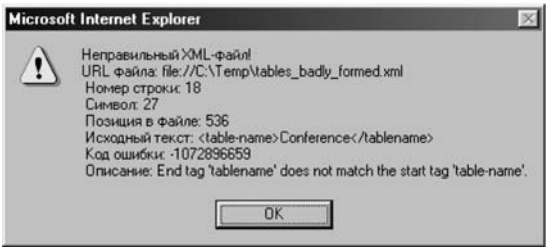


Рис. A.6. *Сообщение об ошибке при загрузке XML-документа, не являющегося корректным*

Если свойство `validateOnParse` установлено в значение `True` и загружаемый XML-документ не соответствует DTD или схеме XML, то сообщение об ошибке показывает подробности о месте ошибки в файле (как правило, не стоит сообщать о расположении ошибки, найденной во внешней сущности). Ниже показано сообщение об ошибке, выдаваемое с помощью функции `MsgBox` (VBScript):

Рис. A.7. *Сообщение об ошибке в случае, когда документ не соответствует DTD или XML-схеме*

Однако не у всех свойств обязательно есть значения. Если, например, ошибка вызвана тем, что загружаемый XML-документ не существует, то нельзя ожидать, что мы можем узнать позицию ошибки в исходном тексте:

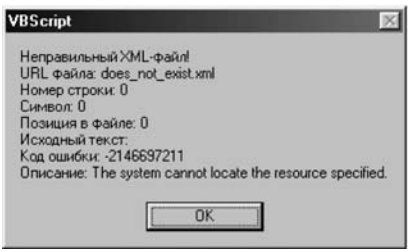


Рис. A.8. *Сообщение об ошибке при загрузке несуществующего документа*

Асинхронная загрузка документов

Асинхронная загрузка позволяет быстрее начать передачу данных пользователю, когда загружается несколько документов. Это особенно удобно, когда во время загрузки можно отображать какую-нибудь полезную информацию, а также при параллельной загрузке нескольких документов.

Чтобы начать процесс загрузки, установите свойство `async` в значение `true` и укажите функцию, которая будет вызываться по окончании загрузки:

В JScript:

```
var doc;
function loadDocument() {
    doc = new ActiveXObject('MSXML2.DOMDocument.3.0');
    doc.onDataAvailable = processDocument;
    doc.async = true;
    doc.load("source.xml");
}
```

Когда все данные будут загружены, будет вызвана функция `processDocument()`. Эта функция может, например, запустить процесс преобразования:

```
function processDocument() {
    if (doc.parseError.errorCode != 0) {
        showError();
    } else {
        transformDocument();
    }
}
```

Как выводить сообщения об ошибках, вы уже видели, в следующем разделе будет показано, как осуществлять преобразование.

В том случае, когда асинхронно загружаются исходный документ и таблица стилей, следует особенно аккуратно писать сценарий, поскольку неизвестно, какой из документов будет загружен первым. Одним из простых решений является использование метода `onDataAvailable` для обоих документов и проверка свойства `readyState` для каждого документа внутри этого метода. Если для обоих документов значение равно `4` (завершено), то можно начать процесс преобразования.

Если нужно достичь минимального времени выполнения, то можно запустить преобразование еще до того, как весь документ станет доступным. В таком случае нужно будет периодически вызывать метод `transform()`, пока он не вернет значение `true`; следить за ходом преобразования можно при помощи свойства `readyState` объекта `IXSLProcessor`.

XSLT-преобразование XML-документов при помощи сценария

Используя MSXML3-анализатор, можно загружать любой корректный XML-документ. В том числе таблицы стилей XSLT и XML-схемы, которые также являются XML-документами. Если создать два экземпляра объекта DOMDocument и загрузить исходный XML-документ в один из них, а таблицу стилей в другой, то затем можно выполнить преобразование при помощи методов transformNode() или transformNodeToObject(), описанных ниже.

Такой способ представляет собой альтернативу указанию таблицы стилей в XML-документе при помощи инструкции обработки <?xml-stylesheet?>. Это также позволяет выбирать таблицу стилей в сценарии, поэтому для одного и того же документа могут быть использованы разные таблицы стилей.

Метод transformNode

Этот метод является, возможно, наиболее полезным, поскольку он возвращает строку с результатом преобразования.

```
strResult = objXMLNode.transformNode(objStylesheet)
```

В качестве параметра objXMLNode, как правило, выступает объект DOM Document, однако часто в этой роли выступает элемент документа. В действительности, метод может быть вызван для любого узла документа. В области видимости таблицы стилей по-прежнему находится весь исходный документ, но первым применяемым шаблонным правилом будет не «match="/"», как обычно, а правило, соответствующее выбранному узлу. Это может оказаться полезным, когда нужно получить выходные данные, сформированные из фрагмента исходного XML-документа.

Параметр objStylesheet также должен быть объектом DOM, будь то экземпляр объекта Document, содержащий действительную таблицу стилей XSLT, или объект Node, являющийся вложенной таблицей стилей XSLT (вложенные таблицы стилей описаны в главе 3).

Возвращаемое значение представляет собой строку, в которой содержится результат преобразования. Обычно это кусок HTML-кода, который может быть вставлен в HTML-страницу для отображения пользователю. Типичная HTML-страница будет содержать в теле следующий код:

```
<body>
<div id="divResults"></div>
</body>
```

именно сюда и будут вставлены результаты преобразования:

```
var objResults = document.all['divResults'];
objResults.innerHTML = objXML.transformNode(objXSL);
```

Такое преобразование показано на примере генеалогического дерева семейства Кеннеди в главе 10; другой пример можно найти на Интернет-сайте книги в файле msxml_transform\default.htm.

> ***Средство просмотра результатов XSL-преобразования не работает, если преобразование выполнено из сценария. В этом случае оно выдаст сообщение об ошибке «Not XML Document». Поэтому следует сначала отладить таблицу стилей в другой среде разработки.***

Метод transformNodeToObject

Этот метод полезен тогда, когда результат преобразования передается в другой объект, такой как DOM Document или Stream:

```
objXMLNode.transformNodeToObject(objStylesheet, objOutput)
```

Параметр objXMLNode должен быть объектом DOM: или непосредственно объектом Document или объектом Node внутри XML-документа (в этом случае этот узел со всеми своими потомками рассматривается как отдельный XML-документ).

Параметр objStylesheet также должен быть объектом DOM, будь то экземпляр объекта Document, содержащий действительную таблицу стилей XSLT, или объект Node, являющийся вложенной таблицей стилей XSLT.

Объект objOutput – это, как правило, другой DOM Document, в который передается результат преобразования. Для этого в результате преобразования должен получиться корректный XML-документ. Этот Document может быть также использован в качестве входных данных для другого преобразования.

Следует помнить, что если результат преобразования сохранять в объекте DOM Document, то XSLT-процессор не будет выполнять сериализацию, и, следовательно, применение элемента <xsl:output> не принесет результата. Если сериализовать документ последовательно при помощи метода save или свойства xml объекта DOM, тогда DOM выполнит собственную сериализацию, не принимая в расчет то, что было указано в таблице стилей.

Пример: Использование JScript на стороне клиента для преобразования документа

В этом примере показано, как загрузить, осуществить синтаксический анализ и преобразование XML-документа при помощи JScript на стороне клиента в Internet Explorer 5 или выше. Файлы для этого примера находятся в папке msxml_transform.

В примере показана HTML-страница с двумя кнопками. При помощи этих кнопок пользователь может выбрать, в каком виде отобразить данные. Щелчок на кнопке означает выбор соответствующей таблицы стилей для преобразования исходного XML-документа.

Исходный XML-документ

Текст исходного XML-документа для этого примера содержится в файле tables_data.xml. В нем определяются столы (те самые столы, за которыми вы сидите во время обеда). Это выглядит следующим образом:

```
<столы>
<стол>
   <название-стола>Conference</название-стола>
   <число-ножек>4</число-ножек>
   <материал-поверхности тип="ламинат">пепельный</материал-поверхности>
   <форма-стола>продолговатый</форма-стола>
   <розничная-цена валюта="USD">1485</розничная-цена>
</стол>
...
</столы>
```

Таблица стилей

Здесь используются две таблицы стилей, tables_list.xsl и tables_catalog.xsl. Данный пример призван продемонстрировать использование JScript для осуществления преобразования, а не само XSLT-преобразование, поэтому листинг для таблиц стилей здесь не приводится.

HTML-страница

HTML-страница default.htm содержит обычную информацию о стиле документа, а затем следует код на JScript, который загружает XML- и XSL-файлы, проверяет на наличие ошибок, а затем производит преобразование. Заметьте, что функция transformFiles принимает в качестве параметра имя таблицы стилей, поэтому можно выбрать ее во время выполнения:

```
<html>
<head>
<style type="text/css">
       body {font-family:Tahoma,Verdana,Arial,sans-serif; font-size:14px}
       .head {font-family:Tahoma,Verdana,Arial,sans-serif;
                                 font-size:18px; font-weight:bold}
</style>

<script language="JScript">
function transformFiles(strStylesheetName) {

  // получаем ссылку на элемент DIV
  var objResults = document.all['divResults'];

   // создаем два экземпляра документа
   var objXML = new ActiveXObject('MSXML2.DOMDocument.3.0');
   var objXSL = new ActiveXObject('MSXML2.DOMDocument.3.0');

   // устанавливаем свойства синтаксического анализатора
   objXML.validateOnParse = true;
   objXSL.validateOnParse = true;

   // загружаем XML-документ и проверяем, не произошла ли ошибка
```

```
    objXML.load('tables_data.xml');
    if (objXML.parseError.errorCode != 0) {
       // найдена ошибка, поэтому выводим сообщение и прекращаем обработку
       objResults.innerHTML = showError(objXML)
       return false;
    }

   // загружаем таблицу стилей XSL и проверяем, не произошла ли ошибка
   objXSL.load(strStylesheetName);
   if (objXSL.parseError.errorCode != 0) {
     // найдена ошибка, поэтому выводим сообщение и прекращаем обработку
     objResults.innerHTML = showError(objXSL)
     return false;
   }

    // все в порядке - выполняем преобразование
    strResult = objXML.transformNode(objXSL);

    // отображаем результаты в элементе DIV
    objResults.innerHTML = strResult;
    return true;
}
```

При условии отсутствия ошибок эта функция выполняет преобразование файла `tables_data.xml` с помощью таблицы стилей, имя которой передается в качестве параметра `strStylesheetName` при вызове. Результат преобразования вставляется в элемент `<div>`, атрибут `id` которого имеет значение «divResults». Ниже будет показано, в каком месте HTML-файла определяется этот элемент.

Если какой-либо из вызовов функции `load` завершился неудачно, возможно, из-за некорректной структуры документа, то вызывается функция `showError`. Эта функция принимает в качестве параметра ссылку на документ, в котором произошла ошибка, и возвращает строку с описанием ошибки. Вместо результатов преобразования пользователю выводится это сообщение об ошибке:

```
function showError(objDocument) {
   // создаем сообщение об ошибке
   var strError = new String;
   strError = 'Неправильный XML-файл!<BR />'
           + 'URL файла: ' + objDocument.parseError.url + '<BR />'
           + 'Номер строки: ' + objDocument.parseError.line + '<BR />'
           + 'Символ: ' + objDocument.parseError.linepos + '<BR />'
           + 'Позиция в файле: ' + objDocument.parseError.filepos + '<BR />'
           + 'Исходный текст: ' + objDocument.parseError.srcText + '<BR />'
           + 'Код ошибки: ' + objDocument.parseError.errorCode + '<BR />'
           + 'Описание: ' + objDocument.parseError.reason
   return strError;
}

//-->
</script>
```

Дальше идет HTML-код, который отображает исходный вид страницы. В открывающем теге элемента <body> указан атрибут onload, который осуществляет вызов функции transformFiles() по окончании загрузки страницы:

```
...
</head>
<body onload="transformFiles('tables_list.xsl')">
<p><span class="head">Трансформация XML-документа с использованием кода на
стороне клиента</span></p>
...
```

Так как в качестве параметра функции используется «tables_list.xsl», то именно эта таблица стилей применяется для формирования первоначального вида. С ее помощью данные представляются в табличном виде.

Затем следует код для создания на странице двух HTML-элементов <button> с надписями Каталог и Простой список. Атрибут onclick каждой из кнопок предписывает при нажатии выполнить функцию transformFiles() с соответствующей таблицей стилей в качестве параметра:

```
...
Показать столы в виде  
<button onclick="transformFiles('tables_catalog.xsl')">Каталога</button>
  или  
<button onclick="transformFiles('tables_list.xsl')">Простого списка</button>
<hr />
```

И наконец, в конце кода определяется элемент , в который и вставляется результат преобразования.

```
<!-- место для вставки результатов преобразования -->
<div id="divResults"></div>

</body>
</html>
```

Результат

При первой загрузке HTML-страницы она выглядит следующим образом:

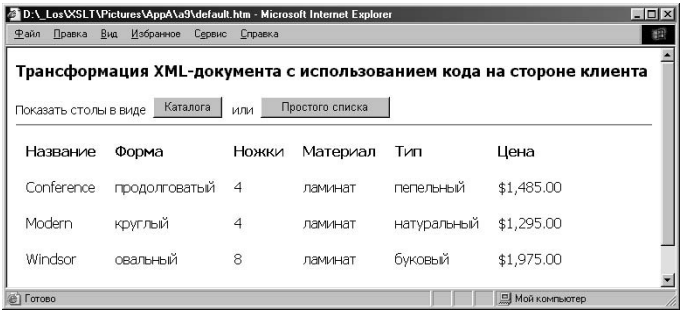


***Рис. А.9.** Первоначальный вид HTML-страницы, демонстрирующей преобразование XML-документа при помощи JScript*

Если нажать на кнопку Каталог, то можно увидеть альтернативное графическое представление тех же данных, полученное после применения другой таблицы стилей.

Использование объекта XSLTemplate

В вышеприведенном примере мы создали два объекта Document для исходного XML-файла и для таблицы стилей, а затем применили к исходному документу метод transformNode для выполнения преобразования и получения HTML-кода.

Такой способ может оказаться неэффективным, если одна и та же таблица стилей применяется неоднократно, особенно на сервере, поскольку в этом случае нужно каждый раз выполнять анализ и проверку действительности таблицы стилей. Этих лишних действий можно избежать, если скомпилировать таблицу стилей в другой объект, который достаточно обманчиво называется XSLTemplate. Этот объект может быть использован для повторяющихся преобразований. Подобные преобразования были показаны на примере генеалогического дерева семейства Кеннеди в главе 10, где каждое преобразование применялось к одному и тому же исходному документу с различными значениями параметра, определяющего, какую часть XML-данных нужно отобразить.

Чтобы скомпилировать таблицу стилей, нужно создать объект XSLTemplate и установить его свойство stylesheet равным узлу DOM (обычно это сам документ), содержащему элемент <xsl:stylesheet>:

```
objStyle = new ActiveXObject("MSXML2.FreeThreadedDOMDocument");
objStyle.async = false;
objStyle.load('stylesheet.xsl');

objTemplates = new ActiveXObject("MSXML2.XSLTemplate");
objTemplates.stylesheet = objStyle.documentElement;
```

Объект XSLTemplate можно использовать столько раз, сколько нужно выполнить преобразований. Для осуществления преобразования нужно создать объект XSLProcessor, который можно рассматривать как хранилище информации, касающейся конкретного преобразования. Обычно этот объект создается каждый раз, когда нужно применить данную таблицу стилей к другому XML-документу. Вам не нужно создавать объект XSLProcessor самостоятельно, объект XSLTemplate сделает это за вас:

```
var objTransformer = objTemplates.createProcessor();
```

Теперь для запуска преобразования нужно указать объекту XSLProcessor, какой исходный документ использовать и передать любые параметры (эти параметры устанавливают значение глобальных элементов <xsl:param> в таблице стилей), а затем вызвать метод transform(). Результат преобразования можно будет получить через свойство output объекта XSLProcessor:

```
objTransformer.input = source;
objTransformer.addParameter("название1", "значение1", "");
```

```
objTransformer.addParameter("название2", "значение2", "");
objTransformer.transform();
objResults.innerHTML = objTransformer.output;
```

Третий аргумент функции addParameter() – это URI пространства имен данного параметра. Обычно значением этого аргумента является пустая строка.

Использование <object> и островков данных XML

Вместо создания экземпляра MSXML Document при помощи сценария можно воспользоваться элементом <object> в HTML-странице. Или же можно создать островок данных XML в HTML-странице с помощью элемента <xml>. Как <object>, так и <xml> являются элементами HTML, а не XML.

Создание экземпляров документа с помощью элемента <object>

Экземпляр MSXML Document можно создать с помощью элемента HTML <object>, указав соответствующий ClassID в зависимости от того, какая версия анализатора используется. В приведенном ниже коде создаются документы, использующие MSXML-анализатор версии 3.0, а затем свойства async и validateOnParse устанавливаются в значение false:

```
<object id="XMLDocument" width="0" height="0"
  classid="clsid:f5078f32-c551-11d3-89b9-0000f81fe221">
  <param name="async" value="false">
  <param name="validateOnParse" value="false">
</object>

<object id="XSLDocument" width="0" height="0"
  classid="clsid:f5078f32-c551-11d3-89b9-0000f81fe221">
  <param name="async" value="false">
  <param name="validateOnParse" value="false">
</object>
```

Единственное изменение, которое требуется внести в использовавшийся ранее код, состоит в том, чтобы получить ссылки на два объекта Document вместо непосредственного их создания внутри сценария:

```
// получаем ссылку на анализатор XML-документа
var objXML = document.all['XMLDocument'];

// получаем ссылку на анализатор таблицы стилей XSL
var objXSL = document.all['XSLDocument'];

// загружаем XML-документ и проверяем, не произошла ли ошибка
objXML.load('tables_data.xml');
if (objXML.parseError.errorCode != 0) {
   // найдена ошибка - показываем сообщение и прерываем процесс
```

```
        objResults.innerHTML = showError(objXML)
        return false;
    }

    // загружаем таблицу стилей и проверяем, не произошла ли ошибка
    objXSL.load(strStylesheetName);
    if (objXSL.parseError.errorCode != 0) {
        // найдена ошибка - показываем сообщение и прерываем процесс
        objResults.innerHTML = showError(objXSL)
        return false;
    }

    // все в порядке - выполняем преобразование
    strResult = objXML.transformNode(objXSL);
```

Использование островков данных XML

В Internet Explorer 5 впервые введен HTML-элемент <xml>, который автоматически создает XML-документ внутри HTML-страницы как изолированный островок данных (data island). Такие документы используют версию MSXML-анализатора по умолчанию. Заметьте, что элемент <xml> – это часть синтаксиса HTML, а не XML. В нижеприведенном коде HTML создаются два островка данных, один из которых содержит исходный XML-документ, а другой таблицу стилей XSL:

```
<xml id="Исходник" src="tables_data.xml"></xml>
<xml id="Таблица стилей" src="tables_list.xsl"></xml>
```

Преобразование в этом случае можно произвести при помощи сценария, подобного приводимому раньше:

```
function transformFiles() {

    // получаем ссылку на конечный элемент DIV
    var objResults = document.all['divResults'];

    // получаем ссылку на XML-документ
    var objXML = document.all['Source'].XMLDocument;

    // получаем ссылку на XSL-документ
    var objXSL = document.all['Stylesheet'].XMLDocument;
    // выполняем преобразование
    strResult = objXML.transformNode(objXSL);

    // и выводим результат в элемент DIV
    objResults.innerHTML = strResult;
}
```

Заметьте, что в этом случае нужно указать, что вы хотите использовать свойство XMLDocument островков данных. В отличие от объектов DOMDocument, созданных непосредственно с помощью сценария или элемента <object>, островок данных, созданный с помощью элемента <xml>, есть всего лишь оболочка для XML-документа. Так как это объект HTML, для получения его

XML-содержимого нужно воспользоваться свойством XMLDocument объекта элемента <xml>.

Использование островков данных <xml> – это наиболее простой метод создания объектов Document, однако он менее гибок, так как нужно указать исходный документ в атрибуте src элемента <xml>. Однако при желании можно указать значение атрибута src и динамически:

```
// получаем ссылку на анализатор таблицы стилей XSL
var objDI = document.all['XSLParser'];
objDI.src = 'tables_list.xsl';
var objXSL = objDI.XMLDocument;
```

Такой способ дает ту же гибкость, что и методы, описанные выше.

И, наконец, если вы еще не пресытились выбором, есть еще один способ создания идентичных островков данных – с помощью элемента <script>, а не <xml>:

```
<script language="XML" src="tables_data.xml"></script>
```

Динамическое изменение XML-документов

Кроме возможности преобразовывать XML-документы с помощью XSLT, анализатор MSXML обеспечивает также полную поддержку объектной модели документа (W3C Document Object Model, DOM) со многими расширениями от Microsoft. Эта поддержка предоставляет целый набор свойств и методов для доступа и манипулирования XML-документом при загрузке его в память.

Можно, например, загрузить XML-документ, изменить в нем некоторые значения и уже после этого применить таблицу стилей для выполнения преобразования. Или же можно использовать методы DOM для извлечения части XML-документа в отдельный документ и затем применить преобразование к этому новому документу. Кроме того, можно загрузить как XML-документ, так и таблицу стилей XSLT, а потом изменить их значения, чтобы они отвечали некоторому условию перед тем, как выполнить преобразование для получения результата.

В качестве альтернативы можно использовать следующий подход – создать пустой объект Document как для XML-документа, так и для таблицы стилей, а затем с помощью методов DOM наполнить его необходимым содержанием. Такой подход может показаться более сложным по сравнению с обычной загрузкой предварительно созданного XML-файла или строки документа. Однако в некоторых ситуациях этот метод может оказаться полезным, особенно в случае небольших документов, преобразования для которых варьируются в широких пределах; в этом случае таблицы стилей XSLT легче создать с нуля, чем держать в виде набора файлов на диске.

Динамическое изменение таблиц стилей XSLT

Во многих случаях можно изменить поведение таблицы стилей, передав ей параметры так, как было описано ранее. Однако некоторые фрагменты таблиц стилей должны быть жестко закодированы. Типичный пример – выражение, используемое в элементе `<xsl:sort>` для определения порядка сортировки. Если же такое ограничение не подходит, то его можно обойти, изменив таблицу стилей после того, как она загружена в память.

Возможным способом такого изменения, конечно, является применение таблицы стилей! Если же предполагаемое изменение невелико, то, вероятно, более эффективно использовать методы DOM непосредственно, то есть осуществить модификацию на месте.

Чтобы дать представление о существующих возможностях, в следующем примере загружаются XML-документ и таблица стилей XSLT, а результат преобразования отображается в виде HTML. Но, кроме того, имеется возможность изменять порядок сортировки конечных данных с помощью кнопок.

Пример: Изменение порядка сортировки

Файлы для этого примера находятся в папке `msxml_dynamic`.

Исходный XML-документ

Исходные XML-данные для этого примера находятся в файле `tables.data.xml`. В нем определены несколько элементов `<table>`, представляющих данные об этих предметах мебели в следующем виде:

```
<столы>
<стол>
  <название-стола>Conference</название-стола>
  <число-ножек>4</число-ножек>
  <материал-поверхности тип="ламинат">пепельный</материал-поверхности>
  <форма-стола>продолговатый</форма-стола>
  <розничная-цена валюта="USD">1485</розничная-цена>
</стол>
...
</столы>
```

Таблица стилей

Таблица стилей находится в файле `tables_list.xsl`. Его можно найти в вышеупомянутой папке; поскольку данный пример призван продемонстрировать использование JScript для управления преобразованием, а не само XSLT-преобразование, то таблица стилей не приводится здесь целиком.

Важная для данного примера часть таблицы стилей – это инструкция `<xsl:apply-templates>`, которая управляет сортировкой и выглядит следующим образом:

```
<xsl:template match="/">
```

```
    <html>
      . . .
       <xsl:apply-templates select="/столы/стол">
           <xsl:sort select="название-стола" order="ascending"/>
       </xsl:apply-templates>
      . . .
    </html>
  </xsl:template>
```

Сценарий в HTML-странице используется для изменения атрибутов элемента `<xsl:sort>`, которые отвечают за то, по какому полю и в каком направлении будут отсортированы данные.

HTML-страница

Ниже приведен код HTML, предназначенный для формирования внешнего вида страницы. Он находится в файле `default.htm`.

Атрибут `onload` элемента `<body>` имеет значение «`if (preparePage()) sort('table-name', 'ascending')`». В этом выражении при загрузке HTML-страницы вызывается функция `preparePage()`. Если возвращаемое значение этой функции – `истина` (то есть процесс завершился удачно), то вызывается функция `sort()`, которая отображает данные, отсортированные по полю «`название-стола`» в порядке возрастания.

```
. . .
<body onload="if (preparePage()) sort('название-стола', 'ascending')">
<p><span class="head">Трансформация XML-документа с использованием кода на
стороне клиента</span></p>
```

Следующий далее HTML-код выводит на экран ряд кнопок, позволяющих пользователю изменить порядок сортировки. Заметьте, что при щелчке на каждой из кнопок выполняется функция `sort`, меняется только значение параметра. Это значение используется для изменения атрибутов `select` и `order` элемента `<xsl:sort>` в таблице стилей.

```
Сортировать по:  
<button onclick="sort('название-стола', 'ascending')">Названию</button>
<button onclick="sort('форма-стола', 'ascending')">Форме</button>
<button onclick="sort('число-ножек', 'ascending')">Кол. ножек</button>
<button onclick="sort('материал-поверхности/@тип',
                                  'ascending')">Материалу</button>
<button onclick="sort('материал-поверхности', 'ascending')">Типу</button>
<button onclick="sort('розничная-цена', 'ascending')">Цене +</button>
<button onclick="sort('розничная-цена', 'descending')">Цене -</button>
<hr/>
```

Затем в HTML-странице определяются объекты, в которых будут храниться исходный документ, таблица стилей и получаемая HTML-страница:

```
<object id="XMLDocument" width="0" height="0"
   classid="clsid:f5078f32-c551-11d3-89b9-0000f81fe221">
```

```
    <param name="async" value="false">
    <param name="validateOnParse" value="false">
</object>

<object id="XSLDocument" width="0" height="0"
    classid="clsid:f5078f32-c551-11d3-89b9-0000f81fe221">
    <param name="async" value="false">
    <param name="validateOnParse" value="false">
</object>

<!--сюда вставляются результаты анализа объектной модели -->
<div id="divResults"></div>

</body>
</html>
```

Код JScript, который реализует эти функции, приведен ниже. Сначала определяется функция `preparePage()`**, в которой создаются глобальные переменные для ссылок на конечный элемент** `<div>` **и на два объекта** `Document`**, а затем в эти объекты загружаются документы XML и XSL:**

```
// глобальные переменные, в которых хранятся ссылки на объекты
var gobjResults;
var gobjXML;
var gobjXSL;

function preparePage() {

    // получаем ссылку на конечный элемент DIV
    gobjResults = document.all['divResults'];

    // получаем ссылку на анализатор XML-документа
    gobjXML = document.all['XMLDocument'];

    // получаем ссылку на анализатор таблицы стилей XSL
    gobjXSL = document.all['XSLDocument'];

    // загружаем XML-документ и проверяем, не произошла ли ошибка
    gobjXML.load('tables_data.xml');
    if (gobjXML.parseError.errorCode != 0) {
        // обнаружена ошибка - показываем сообщение и выходим
        gobjResults.innerHTML = showError(gobjXML)
        return false;
    }

//загружаем таблицу стилей XSL и проверяем, не произошла ли ошибка
    gobjXSL.load('tables_list.xsl');
    if (gobjXSL.parseError.errorCode != 0) {
        // обнаружена ошибка - показываем сообщение и выходим
        gobjResults.innerHTML = showError(gobjXSL)
        return false;
    }

    // все в порядке - возвращаем значение истина
    return true;
}
```

Если какой-либо из вызовов load() завершится неудачно, то для вывода сообщения об ошибке будет использована функция showError() (та же самая, что использовалась в предыдущих примерах), а сама функция возвратит значение ложь. Если же все в порядке, то функция возвратит значение истина, и можно будет вызывать функцию sort().

Теперь мы готовы к тому, чтобы написать функцию sort(), которая изменяет значения атрибутов select и order элемента <xsl:sort>. Поскольку таблица стилей загружается в память как Document DOM, то это можно сделать с помощью DOM-интерфейсов. Для того чтобы получить непосредственный доступ к упомянутым двум атрибутам, очень удобно использовать выражение XPath.

Реализованный Microsoft метод selectSingleNode() принимает в качестве параметра выражение XPath, а возвращает первый выбранный из документа узел. К счастью, в нашей таблице стилей содержится только один элемент <xsl:sort>.

С технической точки зрения функция selectSingleNode() *ожидает, что выражение соответствует синтаксису Microsoft WD-xsl, если только свойство* selectionLanguage *объекта* DOMDocument *не установлено в* «XPath». *Но для такого простого выражения, как использованное в данном примере, синтаксис одинаков в обоих языках.*

Таким образом, функция sort() имеет следующий вид:

```
function sort(strSortBy, strOrder) {

   // получаем ссылку на атрибуты 'select' и 'order'
   // элемента 'xsl:sort' в корневом шаблоне
   var objSelect = gobjXSL.selectSingleNode("//xsl:sort/@select");
   var objOrder = gobjXSL.selectSingleNode("//xsl:sort/@order");

   // изменяем значение атрибута на указанное в выражении XPath
   objSelect.nodeValue = strSortBy;
   objOrder.nodeValue = strOrder;

   // выполняем преобразование
   strResult = gobjXML.transformNode(gobjXSL);

   // и обновляем содержимое элемента DIV
   gobjResults.innerHTML = strResult;
}
```

Результат

Для каждой колонки в полученной таблице существует своя кнопка. Для колонки Цена на странице присутствуют две кнопки, чтобы сортировку можно было осуществлять как в порядке возрастания, так и в порядке убывания (рис. А.10).

Щелчок на любой кнопке приводит к тому, что данные сортируются по выбранной колонке без повторной загрузки страницы, как показано на рис. А.11.

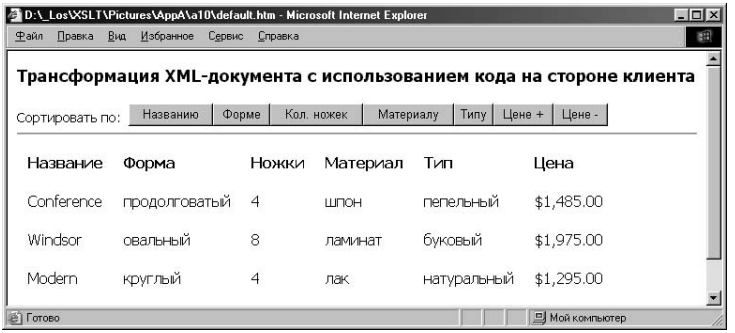


***Рис. А.10.** Начальный вид HTML-страницы с таблицей характеристик столов*

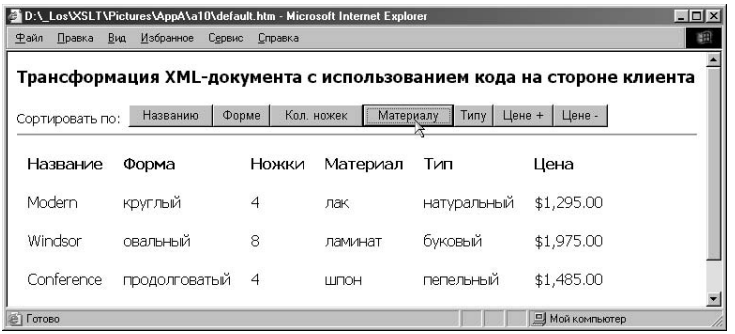


***Рис. А.11.** HTML-страница с характеристиками столов; данные отсортированы по колонке «Материал»*

Использование XPath в DOM

В предыдущем примере для выбора узла DOM-документа использовался метод `selectSingleNode()`, в который мы передавали выражение XPath. Такой способ работает, потому что MSXML позволяет использовать выражения XPath независимо от таблицы стилей XSLT. Если же нужно просто извлечь некоторую часть информации из документа, то существует другой удобный способ для запуска преобразования XSLT.

Следующая функция создает документ DOM и загружает в него XML-файл. Затем она вызывает функцию `selectNodes` и передает туда выражение XPath `«//стол/название-стола»`; возвращаемым значением функции является объект `NodeList`, содержащий все элементы `<table-name>` документа:

```
function getTableNames() {

   // получаем ссылку на конечный элемент DIV
   var objResults = document.all['divResults'];

   // создаем экземпляр XML-документа
   var objXML = new ActiveXObject('MSXML2.DOMDocument.3.0');
```

```
    // Загружаем XML-документ и проверяем на ошибки
    objXML.load('tables_data.xml');
    if (objXML.parseError.errorCode != 0) {
       // обнаружена ошибка - показываем сообщение и выходим
       objResults.innerHTML = showError(objXML)
       return false;
    }

    var strResult = new String;   // для хранения результата
    strResult = 'Названия столов: ';

    // получаем список узлов, содержащий все элементы <стол>
    var objTableNodeList = objXML.selectNodes('//стол/название-стола');
```

Получив список элементов `<название-стола>`, теперь уже несложно пройтись по ним и извлечь значения (из дочернего текстового узла каждого элемента), которые затем добавляются к конечной строке, помещаемой в элемент `<div>` в произвольном месте страницы:

```
    // проходим по всем узлам <название-стола>
    for (var i = 0; i < objTableNodeList.length; i++) {

       // получаем значение данного узла из дочернего текстового узла
       strName = objTableNodeList(i).childNodes(0).nodeValue;
       strResult += '"<b>' + strName + '</b>" ';
    }

    // и обновляем содержимое элемента DIV
    objResults.innerHTML = strResult;
}
```

Результат применения данной функции к использованному ранее XML-документу показан на рис. А.12.

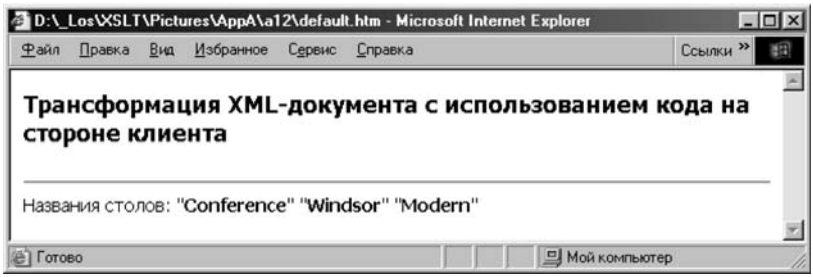


Рис. А.12. *Результат использования выражений XPath в DOM-модели*

Использование анализатора MSXML3 на стороне сервера

В рассмотренных ранее примерах мы использовали MSXML3 для осуществления XSLT-преобразований исключительно на стороне клиента. Все замечательно, пока клиент поддерживает использование анализатора MSXML3.

Даже если бы все броузеры поддерживали XSLT, все равно сценарии, написанные нами в HTML-страницах, работали бы только через API от Microsoft.

Если же вы разрабатываете веб-сайт, доступный для пользователей с различными типами броузеров, или же по причинам безопасности не хотите передавать XML-данные клиенту, то вам придется осуществлять преобразование из XML в HTML на стороне сервера, а затем уже передавать полученный HTML-код клиенту. В других приложениях я опишу ряд процессоров XSLT, которые можно использовать на сервере, но если используется сервер Windows, то для этой цели можно использовать и процессор MSXML3.

Анализатор MSXML3 – это обычный COM-объект, поэтому его можно создавать и применять с помощью любого языка, поддерживающего модель COM. К таким средствам относятся сценарии VBScript и JScript, запускаемые из ASP-файла на сервере, а также серверные приложения, построенные с помощью Visual Basic, C++, Java, Delphi и т. п.

Использование MSXML3 с ASP

Для создания экземпляра анализатора в ASP на стороне сервера используются практически те же методы, что мы использовали на стороне клиента. Однако для его корректной инициализации в контексте страницы ASP нужно использовать метод `CreateObject` объекта `Server`:

В VBScript:

```
Dim objDocument
Set objDocument = Server.CreateObject("MSXML2.DOMDocument.3.0")
```

В JScript:

```
var objDocument = Server.CreateObject('MSXML2.DOMDocument.3.0');
```

Можно, конечно, для создания экземпляра документа использовать тот же элемент `<object>`, который мы использовали ранее на стороне клиента. При этом нужно установить атрибут `runat:server`. Такой подход имеет то преимущество, что фактически экземпляр не создается до тех пор, пока на него не ссылаются, в то время как метод `Server.CreateObject` сразу же создает экземпляр. Конечно, любое преимущество является таковым, если вы нужным образом структурируете свой код (в данном случае это зависит от того, когда вы вызываете метод `Server.CreateObject`).

> ***При создании экземпляра документа внутри ASP-файла*** `global.asa` ***таким образом, что он доступен в течение всего пользовательского сеанса или до окончания работы приложения, не забудьте указать многопоточную версию анализатора, задав нужные имена объектов, приведенные на стр. 759.***

После того как документ создан, дальнейшие действия по его использованию очень похожи. При этом, конечно, нельзя использовать окна сообщений или всплывающие диалоги, кроме того, для вывода на экран клиента информации нужно использовать метод `Response.Write` вместо ее динамической вставки в страницу, как мы делали раньше.

Пример: XSLT-преобразование на стороне сервера

Данная ASP-страница вместе с необходимыми файлами содержится в папке `msxml_asp`. Здесь продемонстрировано простое XSLT-преобразование на стороне сервера, результат которого посылается клиенту в виде HTML-страницы.

Исходный XML-документ

Исходный XML-документ представляет собой тот же мебельный каталог, что использовался в предыдущих примерах данной главы. Он начинается с такого фрагмента:

```
<столы>
<стол>
  <название-стола>Conference</название-стола>
  <число-ножек>4</число-ножек>
  <материал-поверхности тип="шпон">пепельный</материал-поверхности>
  <форма-стола>продолговатый</форма-стола>
  <розничная-цена валюта="USD">1485</розничная-цена>
</стол>
```

Таблица стилей

Таблица стилей находится в файле `tables_style.xsl` в той же папке. Мы здесь не касаемся содержимого таблицы стилей; она практически та же самая, что и в предыдущих примерах. Единственный важный момент состоит в том, что на выходе она формирует HTML-страницу.

ASP-страница

Ниже приводится ASP-страница. Файл называется `simple_transform.asp`:

```
<%@LANGUAGE="VBScript"%>

<%
' сообщить клиенту, что мы посылаем HTML-данные
Response.ContentType = "text/html"

' процедура обработки ошибок
Sub ShowError(objDoc)
   ' создаем и выводим сообщение об ошибке
   Dim strError
   strError = "Неправильный XML-файл!<BR />" _
            & "URL файла: " & objDoc.parseError.url & "<BR />" _
            & "Номер строки: " & objDoc.parseError.line & "<BR />" _
            & "Символ: " & objDoc.parseError.linepos & "<BR />" _
            & "Позиция в файле: " & objDoc.parseError.filepos & "<BR />" _
            & "Исходный текст: " & objDoc.parseError.srcText & "<BR />" _
            & "Код ошибки: " & objDoc.parseError.errorCode & "<BR />" _
            & "Описание: " & objDoc.parseError.reason
   Response.Write strError
End Sub
Dim objXML
```

```
Dim objXSL

' создаем два экземпляра документа
Set objXML = Server.CreateObject("MSXML2.DOMDocument.3.0")
Set objXSL = Server.CreateObject("MSXML2.DOMDocument.3.0")

' устанавливаем свойства анализатора
objXML.ValidateOnParse = True
objXSL.ValidateOnParse = True

' загружаем исходный XML-документ и проверяем на ошибки
objXML.load Server.MapPath("tables_data.xml")
If objXML.parseError.errorCode <> 0 Then
   ' обнаружена ошибка - показываем сообщение и выходим
   ShowError objXML
   Response.End
End If
```

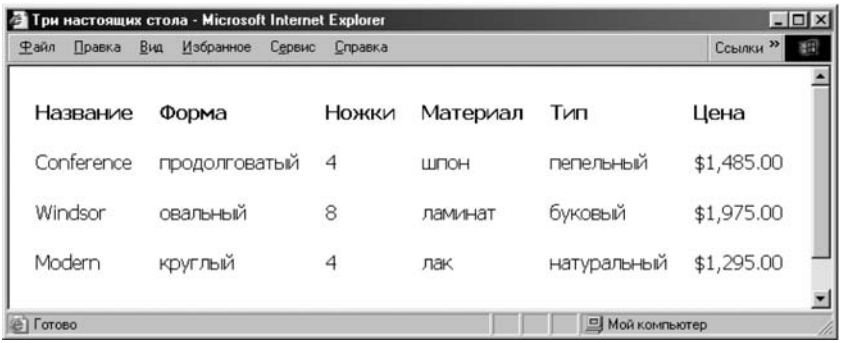
Заметьте, что для преобразования имен XML- и XSLT-документов в корректную форму полного пути к файлам в ASP следует использовать метод `Server.MapPath`. Если этого не сделать, то метод `load` не сможет найти файлы и выдаст сообщение об ошибке.

```
' загружаем таблицу стилей XSL и проверяем на ошибки
objXSL.load Server.MapPath("tables_style.xsl")
If objXSL.parseError.errorCode <> 0 Then
   ' обнаружена ошибка - показываем сообщение и выходим
   ShowError objXSL
   Response.End
End If

' все в порядке - выполняем преобразование
strResult = objXML.transformNode(objXSL)

' вставляем результат в страницу
Response.Write strResult
%>
```

Результат



Название	Форма	Ножки	Материал	Тип	Цена
Conference	продолговатый	4	шпон	пепельный	$1,485.00
Windsor	овальный	8	ламинат	буковый	$1,975.00
Modern	круглый	4	лак	натуральный	$1,295.00

Рис. А.13. *Результат XSLT-преобразования на стороне сервера при помощи ASP*

Можно, конечно, было бы выбрать XML-документ и таблицу стилей XSLT динамически при выполнении страницы вместо использования имен непосредственно в коде сценария. Например, секция <FORM> другой страницы могла бы ссылаться на данную страницу и использовать для имени таблицы стилей значение из элемента управления HTML, возможно, с именем "stylesheet_name". После этого в коде ASP-страницы нужно использовать это значение в методе load:

```
objXSL.load Server.MapPath(Request.Form("stylesheet_name"))
```

Установка соответствующего заголовка типа содержимого

Обратите внимание, что в предыдущем примере использовалась следующая инструкция ASP:

```
Response.ContentType = "text/html"
```

тем самым устанавливался заголовок типа содержимого HTTP, чтобы клиент знал, что данные нужно рассматривать как страницу HTML. Дело в том, что мы создаем HTML-страницу в результате преобразования на стороне сервера. В данном случае, если эта инструкция не будет включена, данная страница все равно будет работать, так как броузер по умолчанию воспринимает содержимое как «text/html», если не указан другой тип.

Тип содержимого указывается как один из стандартных типов MIME и его всегда следует указывать с помощью свойства Response.ContentType. Это особенно важно, если формируемые данные не являются HTML-страницей. Например, если на выходе XSLT-преобразования или другого кода ASP формируется XML-документ, то следует использовать:

```
Response.ContentType = "text/xml"
```

Если вы создаете таблицу стилей XSLT, чтобы послать ее клиенту, следует использовать «text/xml». Internet Explorer также воспринимает тип «text/xsl», однако, поскольку это не стандартный тип MIME, то другие броузеры могут не распознать такой тип. Для обычных текстовых файлов используйте «text/plain». Между тем нужно понимать, что клиент сам (или, скорее, операционная система, в которой он работает) решает, как поступить с полученными данными, основываясь на типе содержимого, который вы указали. Вы только сообщаете клиенту, какой тип данных формируется, а уж клиент сам решает, как их обработать, когда они поступят.

В действительности Internet Explorer пытается предугадать MIME-тип в заголовке HTTP. В частности, он придает значение конечной части URL: если, например, он заканчивается на «.html», то броузер ожидает получить HTML-файл, а если на «.xml», то ожидается загрузка XML-данных. Однажды я потратил полдня на то, чтобы понять, почему в ответ на следующий URL:

```
http://www.company.com/transform.asp?style=my.xsl&source=my.xml
```

данные отображаются так, как будто бы это XML-данные, в то время как в коде тип содержимого HTTP четко указан как «text/html». Причина заклю-

чалась в присутствии «.xml» в конце строки URL. Добавление фиктивного параметра «&x=y» решает проблему.

Установка параметров

Когда ASP-страницы используются для управления преобразованием, то очень часто требуется устанавливать параметры таблицы стилей в соответствии с параметрами, переданными через URL. Например, если указан следующий URL:

```
http://www.company.com/transform.asp?employee=517541
```

то хотелось бы выполнить преобразование, установив значение параметра employee в таблице стилей в значение 517541. К счастью, сделать это достаточно просто.

Если требуется передать параметры, то нельзя непосредственно использовать упрощенный метод вызова transformNode() объекта DOMDocument. Вместо этого нужно выполнить следующие действия: создать объект XSLTemplate, представляющий собой откомпилированную таблицу стилей; затем создать объект XSLTProcessor, вызвав метод createProcessor() объекта XSLTemplate и после этого вызвать метод addParameter() объекта XSLTProcessor для установки значения параметра. В предыдущем примере мы писали:

```
strResult = objXML.transformNode(objXSL)
```

теперь же нам нужно раскрыть этот процесс:

```
Dim templates
Dim transformer

templates = Server.CreateObject("MSXML2.XSLTemplate")
templates.stylesheet = objXSL.documentElement
transformer = templates.createProcessor()
transformer.input = objXML
transformer.addParameter("employee", Request.QueryString("employee"), "")
strResult = transformer.transform()
```

В действительности вы вряд ли захотите проделать все это на одной и той же ASP-странице. Преимущество компилирования таблицы стилей в объект XSLTemplate состоит в возможности повторного использования данной таблицы стилей. Поэтому в промышленном приложении вам захочется лишь однажды построить объект XSLTemplate, храня ссылку на него на уровне приложения в файле global.asa. Учтите, что при таком подходе очень важно использовать многопоточную версию объекта DOMDocument.

Создание ссылок на другие ASP-страницы

Рассмотрим другую проблему: когда в подобных приложениях создаются HTML-страницы, то внутри этих страниц будут находиться ссылки на другие ASP-страницы. Зачастую возникает проблема с использованием символа «&» в таких, например, URL:

```
«http://www.company.com/transform.asp?emp=517541&loc=bracknell»,
```

то «&» будет выведен как «&». Не беспокойтесь об этом, для HTML-страницы это корректный способ представления URL, и броузер поймет это. Фактически написание «&» без экранирования является технической ошибкой, однако многие авторы поступают таким образом постоянно.

Решаем, где осуществлять XSLT-преобразование

Принимая во внимание то, что некоторые броузеры поддерживают XSLT-преобразование на стороне клиента, а некоторые нет, представляется естественным осуществлять преобразование на стороне клиента, если оно там поддерживается, и на стороне сервера в противном случае. В принципе такой выбор может быть произведен с помощью ASP-сценария, который определяет версию броузера (по заголовку HTTP, посылаемому в каждом сообщении) и посылает обратно соответствующее содержимое. В действительности Microsoft поставляет расширение ISAPI для своего веб-сервера, которое выполняет именно эти действия. Но когда речь идет об XSLT, то тут возникает одна загвоздка, так как определение версии броузера еще не говорит о том, установлен ли у пользователя MSXML3.

С помощью достаточно несложного сценария можно легко определить версию броузера – в данном примере логическая переменная `blnIsIE5` устанавливается в `True`, если данный броузер – Internet Explorer 5 или более поздняя версия:

```
blnIsIE5 = False
strUA = Request.ServerVariables("HTTP_USER_AGENT")
If InStr(strUA, "MSIE") Then
  intVersion = CInt(Mid(strUA, InStr(strUA, "MSIE") + 5, 1))
  If intVersion > 4 Then blnIsIE5 = True  'это IE5 или выше
End If
```

Единственная оставшаяся проблема – определить, какая версия анализатора MSXML3 доступна.

Вообще говоря, можно было бы создать две таблицы стилей: одну написанную на XSLT и другую на WD-xsl, а затем послать клиенту HTML-страницу, содержащую сценарий, определяющий тип установленного программного обеспечения и загружающий подходящую таблицу стилей. Такой метод был показан раньше в примере *Использование JScript на стороне клиента для преобразования документа* на стр. 773. Все, что нужно сделать, прежде чем решить, какую таблицу стилей загрузить, – это добавить логику, подобную той, что используется в сценарии, написанном Ярно Эловирта (Jarno Elovirta) и включенном в нашу тестовую страницу `msxml-version.html`.

На практике же лишь малое количество проектов может позволить себе такую роскошь, как написание двух экземпляров каждой таблицы стилей! Более практично предоставлять пользователям выбор: либо загрузить и установить MSXML3, либо использовать основанную на серверных преобразованиях версию сайта. Это практически то же самое, что просить пользователя

выбрать между версиями сайта с фреймами и без них: здесь нет особой трудности, особенно если вы сохраняете информацию о пользовательских предпочтениях на сайте или в «cookie».

Поскольку приложение знает, где выполняется преобразование: на стороне клиента или на стороне сервера, то его уже не трудно настроить. Примечательно, что и в том и в другом случае используются одни и те же исходные XML-документы и практически одинаковые таблицы стилей; в примере с генеалогическим деревом семейства Кеннеди в главе 10 мы видели, как произвести требуемые незначительные изменения с помощью `<xsl:import>`.

Если XML-данные формируются на сервере (например, извлекаются из базы данных), то туда можно без труда добавить инструкцию обработки `<?xml-stylesheet?>`. Преимущество такого подхода в том, что отпадает необходимость в HTML-странице, предназначенной для управления преобразованием. На сервере можно выбирать различные таблицы стилей программным образом, а инструкция `<?xml-stylesheet?>` может быть просто проигнорирована.

В случае хранения XML-данных на сервере добавлять инструкцию `<?xml-stylesheet?>` перед их отправкой клиенту не очень хорошая идея. Так можно потерять многие преимущества производительности, которые дает осуществление преобразования на стороне клиента: в идеале данные на сервере вообще не стоит трогать. Лучше все-таки управлять преобразованием из HTML-страницы показанным выше образом.

Если все сделано правильно, то представленные данные будут иметь один и тот же внешний вид независимо от того, где было осуществлено преобразование: на стороне клиента или на стороне сервера, что можно видеть на приведенном ниже рисунке:

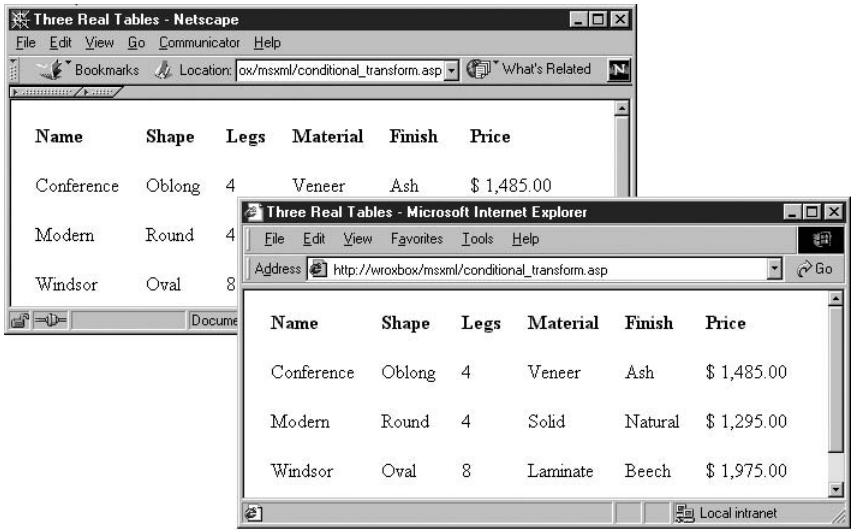


Рис. A.14. *Внешний вид данных, полученных в результате преобразования на стороне клиента и на стороне сервера*

В результате броузер Internet Explorer версии 5 и выше будет получать исходный XML-документ, содержащий инструкцию <xsl:stylesheet>, а затем извлекать эту таблицу стилей для применения при XSLT-преобразовании на стороне клиента. Тем временем другие броузеры будут получать просто HTML-страницу, полученную в результате преобразования на сервере.

Единственное отличие – производительность: если можно перемещаться по последовательным страницам из одного исходного набора данных XML, как в случае с примером генеалогического дерева в главе 10, то переходы по ссылкам будут гораздо быстрее, если для них не требуется посылать дополнительные запросы к серверу.

Выполнение значительного количества преобразований

Если для вывода информации клиенту используются преобразования на стороне сервера, то тем самым на него оказывается излишняя нагрузка. Вебсерверы работают с большей производительностью, когда данные хранятся на диске и передаются клиенту по запросу, а не формируются каждый раз заново. Если хранящиеся на сайте XML-данные достаточно статичны, то одним из способов сократить нагрузку является выполнение требуемых преобразований при размещении данных на сервере и запись результатов в файлы, расположенные на серверном диске. После этого клиент может получать доступ и загружать нужные ему файлы без преобразования в реальном времени.

Записать результаты преобразования на серверный диск не составит труда, если использовать FileSystemObject:

```
...
'все в порядке - выполняем преобразование
strResult = objXML.transformNode(objXSL)

'и записываем результаты на диск сервера
Set objFSO = Server.CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.CreateTextFile("resultpage.htm", True)
                                'перезаписываем существующие файлы
objFile.Write strResult
objFile.Close
```

В качестве альтернативы можно взглянуть на программу XSLTransform от StoneBroom, которая использует пакетный файл для автоматического выполнения нескольких преобразований и записи результатов в файлы на диске. За более подробными сведениям обращайтесь на сайт *http://www.stonebroom.com*.

Краткий справочник по MSXML3

Объекты, методы, события и свойства, имеющиеся в MSXML3, приведены в файле помощи, который поставляется вместе с SDK. Инструкции по загрузке SDK приведены в начале данного приложения.

Сюда включены только те части интерфейса, которые имеют отношение к обработке при помощи XSLT и XPath. Здесь описаны только основные интерфейсы и опущены некоторые, представляющие интерес, которые (на момент написания книги) столь кратко описаны в документации, что для того чтобы заставить их работать, пришлось бы пройти по пути проб и ошибок. Полное описание интерфейса Microsoft DOM (за исключением черт, появившихся в MSXML3) можно найти в книге издательства Wrox «XML профессионалам» («Professional XML» – ISBN 1-861003-11-0).

Объекты

Объекты, представляющие особый интерес для обработки с помощью XSLT и XPath, приведены ниже:

Объект	Описание
`IXMLDOMDocument`	Корневой узел XML-документа
`IXMLDOMNode`	Любой узел в DOM
`IXMLDOMNodeList`	Набор объектов `Node`
`IXMLDOMParseError`	Подробности последней ошибки, произошедшей при синтаксическом анализе
`IXMLDOMSelection`	Выборка узлов
`IXSLProcessor`	Выполнение таблицы стилей XSLT
`IXSLTemplate`	Откомпилированная таблица стилей XSLT в памяти

Эти объекты описываются в приведенных ниже разделах.

IXMLDOMDocument и IXMLDOMDocument2

Класс `IXMLDOMDocument` наследует все свойства и методы `IXMLDOMNode`. `IXMLDOMDocument2` – это более поздняя версия данного интерфейса, в которой появились некоторые дополнительные методы и свойства. В этом разделе приводятся дополнительные свойства и методы, имеющие отношение к обработке при помощи XSLT и XPath, иначе говоря, все то, что отсутствует у `IXMLDOMNode`, который будет описан на стр. 799.

Дополнительные методы

Ниже подробно описаны методы, относящиеся к обработке при помощи XPath и XSLT.

Методы `validate()` и `setProperty()` принадлежат интерфейсу `IXMLDOMDocument2`, который является расширением класса `IXMLDOMDocument`, введенного в MSXML2.

Имя	Тип возвращаемого значения	Описание
abort	(Нет)	Прерывает загрузку
load	Boolean	Загружает документ из указанного источника XML
loadXML	Boolean	Загружает документ из строки
save	(Нет)	Сохраняет документ в указанный приемник
setProperty	(Нет)	Устанавливает разнообразные системные свойства
validate	(Нет)	Проверяет действительность документа в соответствии с текущим DTD или схемой

abort() ⇒ (Ничего не возвращает)

Если документ загружается асинхронно, то в любой момент можно вызвать метод abort(), чтобы прервать процесс.

load(URL) ⇒ Логическое значение

Как правило, аргументом является строка, содержащая URL – этот URL должен идентифицировать XML-документ. Результатом действия данного метода является очистка любого существующего содержимого объекта Document и помещение туда результата синтаксического анализа исходного XML-документа из указанного URL. Если действие прошло успешно, то возвращается значение True, в противном случае False.

Можно, конечно, использовать и относительный URL, такой как «../article.xml»; когда содержащий метод код выполняется из HTML-страницы, такой URL будет воспринят относительно URL самой страницы. На сервере вы, скорее всего, захотите использовать URL, полученный с помощью функции mapPath().

loadXML(строка) ⇒ Логическое значение

В передаваемой строке содержится текст XML-документа. Результатом действия данного метода является очистка любого существующего содержимого объекта Document и помещение туда результата синтаксического анализа строки XML-данных. Если действие прошло успешно, то возвращается значение True, в противном случае False.

save(место назначения) ⇒ (Ничего не возвращает)

В качестве параметра используется имя файла, представленное в виде строки. В результате действия метода XML-документ сериализуется и выводится в файл. В качестве параметра можно также указывать другие объекты, такие как еще один объект Document, в этом случае получается два документа. Можно произвести сохранение непосредственно в объект Response.

setProperty(имя, значение) ⇒ (Ничего не возвращает)

Этот метод применяется для установки различных системных свойств, которые оказывают влияние на поведение других методов, в особенности на методы `selectNodes()` и `selectSingleNode()`. Существует также соответствующий метод `getProperty()`. Вот три важнейших свойства:

- `SelectionLanguage`. Принимает значения «XPath» или «XSLPattern». По умолчанию используется «XSLPattern». От данного свойства зависит синтаксис выражений, передаваемых в методы `selectNodes()` и `selectSingleNode()`. Для использования в этих выражениях синтаксиса XPath 1.0 нужно установить данное свойство равным «XPath». Значение по умолчанию «XSLPattern» относится к диалекту Microsoft WD-xsl.
- `SelectionNamespaces`. Значение этого свойства представляет собой список объявлений пространств имен, разделенных пробелами, например:

```
«xmlns:a='http://company-a.com/' xmlns:b='http://company-b.com/'»
```

- Таким образом определяются префиксы пространств имен, которые могут быть использованы в выражениях, передаваемых в методы `selectNodes()` и `selectSingleNode()`.
- `ServerHTTPRequest`. Это свойство принимает логические значения `True` или `False`; по умолчанию используется `False`. Его нужно устанавливать в значение `True` при исполнении сценария на сервере в том случае, когда имеется существенное количество конкурирующих процессов. Если оно установлено в `True`, то свойство `async` должно быть установлено в `False`. Это свойство было введено для предотвращения проблем, с которыми столкнулись некоторые пользователи бета-версии MSXML3 на сильно загруженных серверах. **Примечание:** если вы используете данное свойство во внутренней сети intranet, то вам также нужно настроить прокси WinHTTP. Для этого требуется утилита настройки, которую можно загрузить отдельно. Инструкции по этому вопросу можно найти в файле помощи пакета SDK.

validate() ⇒ (Ничего не возвращает)

Этот метод проверяет, что данный документ является действительным, то есть удовлетворяет указанному в нем DTD или схеме. Если документ действителен, то метод завершается обычным образом, в противном случае выдается ошибка.

Данный метод является альтернативой проверке на действительность во время синтаксического анализа. Он может быть полезен, например, для проверки того, что создаваемый в XSLT-преобразовании документ является действительным.

Типичная последовательность вызовов в JScript такова:

```
try {
   xmldoc.validate();
} catch(e) {
   alert("Документ недействителен: " + e.description);
}
```

Дополнительные свойства

Имя	Тип значения	Описание
async	Boolean	Равно True в случае асинхронной загрузки
parseError	IXMLDOMParseError	Последняя ошибка анализатора
readyState	Long	Текущее состояние готовности
validateOnParse	Boolean	Требуется ли проверка документа на действительность

async ⇒ Логическое значение

Если это свойство установлено в значение по умолчанию (True), то обработка может совершаться в то время, когда документ еще анализируется и загружается. При использовании таблиц стилей свойство async обычно устанавливается равным False, поэтому преобразование может быть запущено только после того, как метод load() вернет управление.

parseError ⇒ IXMLDomParseError

Это свойство возвращает объект IXMLDOMParseError. Чтобы определить, произошла ли ошибка при синтаксическом анализе, нужно проверить, отличается ли свойство errorCode возвращаемого объекта от нуля.

readyState ⇒ Длинное целое

При асинхронной загрузке документа это свойство принимает одно из следующих значений:

0	UNINITIALIZED Метод load() еще не вызывался
1	LOADING Метод load() находится в состоянии выполнения
2	LOADED Документ загружен и в данный момент анализируется
3	INTERACTIVE Часть данных была прочитана, и можно уже получить доступ к структуре данных DOM в режиме чтения
4	COMPLETED Данные полностью доступны

Можно назначить функцию, которая будет вызываться при изменении состояния, используя событие onReadyStateChange, или же функцию, которая будет вызываться при переходе состояния в COMPLETED при помощи события onDataAvailable.

validateOnParse ⇒ Логическое значение

Если значение этого свойства равно `True`, то документ будет проверяться на соответствие DTD или схеме во время загрузки.

IXMLDOMNode

Этот объект представляет собой узел в дереве документа. Заметьте, что данное дерево удовлетворяет модели DOM, которая не всегда совпадает с моделью XPath, описанной в главе 2; так, например, отличаются способы моделирования пространств имен, а текстовые узлы не обязательно нормализованы.

Для каждого типа узла в дереве существует свой подкласс `IXMLDOMNode`. Описания всех этих подклассов здесь не приводятся, поскольку они не относятся напрямую к обработке при помощи XSLT и XPath. Единственный подкласс, который здесь описывается, это `IXMLDOMDocument`, который можно рассматривать как целый документ или как его корневой узел, в зависимости от вашего желания.

Методы

Ниже приведены методы объекта `IXMLDOMNode`, которые имеют отношение к обработке с помощью XSLT и XPath. Чаще всего эти методы применяются к корневому узлу (объект DOM `Document`), хотя, вообще говоря, могут применяться к любому узлу.

Имя	Тип возвращаемого значения	Описание
`selectNodes`	IXMLDOMNodeList	Вычисляет выражение XPath и возвращает соответствующие ему узлы
`selectSingleNode`	IXMLDOMNode	Вычисляет выражение XPath и возвращает первый соответствующий узел
`transformNode`	String	Применяет таблицу стилей к поддереву, корнем которого является данный узел, возвращая результат в виде строки
`transformNode ToObject`	(Нет)	Применяет таблицу стилей к поддереву, помещая результат в указанный документ или поток

selectNodes(Строковое выражение) ⇒ IXMLDOMNodeList

В качестве параметра этого метода выступает выражение XPath, а возвращаемое значение представляет собой список узлов, которые выбираются при помощи этого выражения.

Для того чтобы в этом выражении использовать синтаксис XPath 1.0, а не диалект Microsoft WD-xsl, нужно установить свойство `selectionLanguage` объекта документа равным «XPath». Для многих простых выражений синтаксис обоих диалектов совпадает, однако если в выражении используются функции или предикаты XPath или же оси, отличающиеся от используемой по умолчанию оси child, то синтаксис может сильно различаться.

Документация Microsoft для этого метода весьма ограничена. По всей видимости, передаваемое выражение XPath должно возвращать набор узлов, а узел, к которому применяется выражение, выступает в роли контекстного узла. Любые префиксы пространств имен интерпретируются на основе объявлений, указанных в свойстве `selectionNamespaces` объекта `DOMDocument`.

Если ни одного узла не выбрано, то метод возвращает пустой список.

В документации не гарантируется, что узлы возвращаются в каком-либо определенном порядке.

Если выходной документ поддерживает интерфейс `IXMLDOMDocument2`, являющийся расширением базового интерфейса `IXMLDOMDocument`, то возвращаемый объект `IXMLDOMNodeList` будет также реализовывать интерфейс `IXMLDOMSelection`.

selectSingleNode(Строковое выражение) ⇒ IXMLDOMNode

Этот метод аналогичен методу `selectNodes()`, описанному выше, за исключением того, что возвращается только первый выбранный узел в порядке появления в документе.

Так же как и в случае с `selectNodes()`, если в выражении требуется использовать синтаксис XPath 1.0 вместо Microsoft WD-xsl, нужно установить свойство `selectionLanguage` объекта документа равным «XPath».

Если ни одного узла не выбрано, то метод возвращает пустое значение.

transformNode(таблица стилей IXMLDOMNode) ⇒ Строковое значение

Этот метод применяет таблицу стилей к документу, содержащему данный узел. Указанный узел используется в качестве исходного контекста для таблицы стилей (то есть обработка начинается с поиска шаблонного правила, соответствующего этому узлу), но сама таблица стилей имеет доступ ко всему документу.

Аргумент обозначает таблицу стилей XSLT. Как правило, это объект `Document`, но это также может быть объект `Node`, представляющий таблицу стилей, вложенную в `Document`.

Результат преобразования сериализуется и возвращается в виде строки. Данная строка соответствует типу данных `BSTR`, что означает игнорирование кодировки, указанной в `<xsl:output>`.

transformNodeToObject(таблица стилей IXMLDOMNode, результат типа Variant) ⇒ (Ничего не возвращает)

Этот метод применяет таблицу стилей к документу, содержащему данный узел. Указанный узел используется в качестве исходного контекста для таблицы стилей, но сама таблица стилей имеет доступ ко всему документу.

Аргумент обозначает таблицу стилей XSLT. Как правило, это объект Document, но это также может быть объект Node, представляющий таблицу стилей, вложенную в Document.

Результат преобразования записывается в объект, указанный во втором аргументе. Чаще всего это объект Document, но может быть и объект Stream.

Свойства

Наиболее полезные свойства приводятся ниже. Свойства, основное назначение которых – осуществлять перемещение по документу, здесь не приводятся, так как для перемещения лучше использовать выражения XPath.

Имя	Тип значения	Описание
baseName	String	Локальное имя узла без префикса пространства имен
namespaceURI	String	URI пространства имен
nodeName	String	Имя узла вместе с префиксом пространства имен, если таковой имеется. Заметьте, что в отличие от модели XPath, неименованные узлы получают условные названия, такие как "#document", "#text" и "#comment"
nodeTypeString	String	Возвращает тип узла в виде строки. Например "element", "attribute" или "comment"
nodeValue	Variant	Значение, хранимое в узле. Это не то же самое, что строковое значение в XPath; так, для элементов оно всегда равно null
prefix	String	Префикс пространства имен, используемого для данного узла
text	String	Текст, содержащийся в данном узле (подобно строковому значению XPath)
xml	String	Представление данного узла и его потомков в виде XML

IXMLDOMNodeList

Данный объект представляет собой список узлов. Наш интерес к нему обусловлен тем, что он является результатом применения метода selectNodes().

Объект IXMLDOMNodeList возвращается методом selectNodes(). Он содержит список узлов, выбранных указанным выражением XPath. Можно обойти все узлы списка, либо используя метод nextNode(), либо непосредственно указывая индекс узла с помощью свойства item. В документации ничего не говорится о том, что узлы списка каким-либо образом упорядочены.

Если документ, из которого выбраны узлы, реализует интерфейс IXMLDOMDocument2, введенный в MSXML2, то список выбранных с помощью метода selectNodes() узлов будет также реализовывать интерфейс IXMLDOMSelection, описанный на стр. 803.

Методы

Имя	Тип возвращаемого значения	Описание
item	IXMLDOMNode	Выдает набор узлов
nextNode	IXMLDOMNode	Выдает следующий узел
reset	(Нет)	Сбрасывает текущую позицию

item(Длинное целое *позиция*) ⇒ IXMLDOMNode

Возвращает узел в заданной позиции. Это коллекция по умолчанию, так что на узел, занимающий позицию n среди узлов NodeList nodes, можно также ссылаться с помощью nodes[n].

nextNode() ⇒ IXMLDOMNode

Возвращает следующий узел или пустое значение, если все узлы уже пройдены.

reset() ⇒ (Ничего не возвращает)

Сбрасывает текущее положение в списке, так что следующим выбранным узлом будет первый узел.

Свойства

Имя	Тип значения	Описание
length	Long	Показывает количество узлов в наборе

IXMLDOMParseError

Этот объект доступен через свойство parseError интерфейса IXMLDOMDocument. Примеры того, как использовать этот объект для диагностики ошибок, были приведены ранее в данном приложении на стр. 768.

Свойства

Имя	Тип значения	Описание
errorCode	Long	Код ошибки
filepos	Long	Положение символа, где произошла ошибка в XML-документе
line	Long	Номер строки, содержащей ошибку
linepos	Long	Положение символа в строке, содержащей ошибку
reason	String	Описание ошибки
srcText	String	XML-данные, содержащие ошибку
url	String	URL документа, в котором произошла ошибка

IXMLDOMSelection

Данный объект представляет собой выборку узлов. Он возвращается методом `selectNodes()`, если обрабатываемый документ реализует интерфейс `IXMLDOMDocument2`.

Проще всего представлять этот объект хранимым выражением, которое по требованию возвращает список узлов. Он ближе всего к представлению (view) из реляционной теории: вам не требуется знать, хранятся результаты или же вычисляются при каждом запросе.

Интерфейс этого объекта является расширением интерфейса `IXMLDOMNodeList`.

Методы

Имя	Тип возвращаемого значения	Описание
`clone`	`IXMLDOMSelection`	Создает копию `IXMLDOMSelection`
`getProperty`	**String**	Возвращает значение указанного свойства
`item`	`IXMLDOMNode`	Выдает набор узлов
`matches`	`IXMLDOMNode`	Проверяет, является ли данный узел членом набора узлов
`nextNode`	`IXMLDOMNode`	Выдает следующий узел
`reset`	**(Нет)**	Сбрасывает текущую позицию

clone() ⇒ IXMLDOMSelection

Создает копию данного `IXMLDOMSelection`.

getProperty(Строка *имя*) ⇒ Строковое значение

Возвращает значение указанного свойства. Единственным свойством, определенным на данный момент, является `SelectionLanguage`, которое воспроизводит одноименное свойство объекта `IXMLDOMDocument`.

item(Длинное целое *позиция*) ⇒ IXMLDOMNode

Возвращает узел в заданной позиции. Это набор узлов по умолчанию, так что на узел, занимающий позицию `n` среди узлов `NodeList nodes`, можно также ссылаться с помощью `nodes[n]`.

matches(IXMLDOMNode *узел*) ⇒ Логическое значение

Определяет, принадлежит ли данный узел к набору узлов, возвращаемому в выражении выборки. (В действительности это использование выражения XPath в качестве образца для совпадения, во многом похожее на использование его в XSLT.)

nextNode() ⇒ IXMLDOMNode

Возвращает следующий узел или пустое значение, если все узлы уже пройдены.

reset() ⇒ (Ничего не возвращает)

Сбрасывает текущее положение в списке, так что следующим узлом, выбранным при помощи `nextNode()`, будет первый узел.

Свойства

Имя	Тип значения	Описание
`expr`	String	Выражение XPath, определяющее, какие узлы будут выбираться. Его можно изменить в любое время, тем самым будет произведена неявная замена текущего списка узлов на новый список
`context`	`IXMLDOMNode`	Устанавливает контекстный узел для вычисления выражения. Изменение контекстного узла неявным образом заменяет текущий список узлов новым списком
`length`	Long	Показывает количество узлов в наборе

IXSLProcessor

Объект `IXSLProcessor` представляет собой отдельное применение таблицы стилей для преобразования исходного документа.

Этот объект создается, как правило, при вызове метода `createProcessor()` объекта `IXSLTemplate`.

Преобразование осуществляется вызовом метода `transform()`.

Методы

Имя	Тип возвращаемого значения	Описание
`addParameter`	(Нет)	Устанавливает значение `<xsl:param>`
`reset`	(Нет)	Сбрасывает состояние процессора и прерывает текущее преобразование
`setStartMode`	(Нет)	Устанавливает режим XSLT и его пространство имен
`transform`	Boolean	Начинает или завершает процесс XSLT-преобразования

addParameter(Строка localName, значение типа Variant, Строка namespaceURI) ⇒ (Ничего не возвращает)

Этот метод присваивает значение глобальному параметру таблицы стилей, объявленному в элементе `<xsl:param>` верхнего уровня. Значение `localName` указывает на локальное имя параметра, а `namespaceURI` на его пространство имен. В общем случае, когда параметр не имеет URI пространства имен, третий аргумент должен быть пустой строкой.

Значение параметра передается во втором аргументе. Оно может быть логического типа, числом или строкой, а также может быть объектом Node или NodeList. В двух последних случаях внутри таблицы стилей значение будет рассматриваться как набор узлов. (Использование в качестве параметра таблицы стилей узла Document является удобной альтернативой использованию XSLT-функции document(), особенно в тех случаях, когда документ уже находится в памяти.)

reset() ⇒ (Ничего не возвращает)

Если преобразование уже запущено, то данный метод прерывает его и устанавливает процессор в исходное состояние, так что он может быть использован вновь. Этот метод **не** очищает значений добавленных параметров, а также не меняет режима startMode.

setStartMode(Строка режим, Строка namespaceURI) ⇒ (Ничего не возвращает)

Как правило, при использовании таблицы стилей обработка начинается с поиска шаблонного правила, соответствующего корневому узлу и определенного без атрибута mode. Иногда удобно использовать одну и ту же таблицу стилей по-разному в различных ситуациях, поэтому MSXML3 позволяет начинать обработку в режиме, отличном от режима по умолчанию. Режимы описываются в главе 4: см. описание элементов <xsl:apply-templates> и <xsl:template>.

Режимы идентифицируются при помощи полного имени, поэтому данный метод позволяет указывать как локальное имя, так и URI пространства имен. В качестве второго аргумента чаще всего передается пустая строка.

transform() ⇒ Boolean

Этот метод применяет таблицу стилей (из которой получен этот XSLProcessor) к исходному документу, определенному в свойстве input. Результат преобразования можно получить через свойство output.

Если преобразование завершено, то метод возвращает значение True. Если исходный документ загружается асинхронно, то метод transform() может вернуть значение False, показывая тем самым, что следует подождать, пока на вход не поступит больше данных. В этом случае можно возобновить преобразование, вызвав метод transform() заново чуть позже. Текущее состояние процесса преобразования можно определить с помощью свойства readyState.

Свойства

Имя	Тип значения	Описание
input	Variant	Исходный XML-документ, над которым осуществляется преобразование. Как правило, передается в виде DOM Document, но также может быть передан как Node или IStream.

Имя	Тип значения	Описание
output	Variant	Результат преобразования. Если для выходного документа не указан объект, то процессор создает строку с результатом, прочесть которую можно с помощью данного свойства. При желании для сформированного документа можно использовать такие объекты, как DOM Document, DOM Node или IStream.
ownerTemplate	IXSLTemplate	Объект XSLTemplate, применяемый для создания объекта процессора.
readyState	Long	Текущее состояние преобразования. По завершении преобразования его значение равно READYSTATE_COMPLETE (3).
startMode	String	Название стартового режима XSLT. См. также метод setStartMode() выше.
startModeURI	String	Пространство имен стартового режима XSLT. См. также метод setStartMode() выше.
stylesheet	IXMLDOMNode	Используемая в данный момент таблица стилей.

IXSLTemplate

Объект IXSLTemplate представляет собой откомпилированную таблицу стилей, помещенную в память. Если нужно использовать одну и ту же таблицу стилей несколько раз, то создание IXSLTemplate и его повторное применение более эффективно, чем неоднократное использование исходной таблицы с помощью transformNode().

Методы

Имя	Тип возвращаемого значения	Описание
createProcessor	IXSLProcessor	Создает объект IXSLProcessor

createProcessor() ⇒ IXSLProcessor

Этот метод следует вызывать только после того, как установлено свойство stylesheet объекта IXSLTemplate.

Данный метод создает объект IXSLProcessor, который можно использовать для преобразования указанного исходного документа.

Свойства

Имя	Тип значения	Описание
stylesheet	IXMLDOMNode	Указывает на таблицу стилей, из которой получен данный объект IXSLTemplate

Установка данного свойства вызывает компиляцию указанной таблицы стилей; объект IXSLTemplate является представлением этой откомпилированной таблицы стилей, которое может быть использовано неоднократно.

Объект DOM `Node`, представляющий таблицу стилей, обычно имеет тип DOM `Document`, но также может быть и элементом, являющимся встроенной таблицей стилей. (Встроенные таблицы стилей описаны в главе 3.)

Документ, указанный в свойстве `stylesheet`, должен быть многопоточным объектом документа.

Заключение

В этом приложении мы сделали обзор приемов и интерфейсов прикладного программирования, доступных при использовании продукта MSXML3 от Microsoft, делая акцент на тех аспектах, которые связаны с XSLT-преобразованиями.

Было показано, как просматривать XML-данные непосредственно в броузере IE5 с использованием как таблицы стилей, определенной по умолчанию, так и указанной в инструкции обработки `<?xml-stylesheet?>`. Затем было продемонстрировано, как использовать сценарии на стороне клиента для управления процессом преобразования. Более того, было показано, каким образом можно изменить таблицу стилей во время применения.

Затем было рассмотрено применение MSXML3 в качестве серверной технологии, выполняющей преобразования, управляемые из ASP-страниц, а также выполнение преобразований на сервере либо на клиенте, в зависимости от возможностей броузера пользователя.

И завершили мы приложение обзором классов и методов, используемых в Microsoft API для управления XSLT-преобразованием.

Microsoft часто подвергается критике, когда дело доходит до соответствия стандартам. Действительно, они произвели огромное замешательство на рынке, когда стали поставлять вместе с Internet Explorer свой диалект WD-xsl, до того как стандарт устоялся. Несомненно, они и сейчас вызывают путаницу, поскольку поставляемый вместе с продуктами Microsoft процессор XSL на момент написания книги по-прежнему поддерживал только WD-xsl, а не XSLT 1.0 (не говоря уже о XSLT 1.1).

Однако MSXML3 в том виде, в котором он окончательно представлен, поддерживает высокий уровень соответствия стандарту и обещает быть устойчивым средством для высокоэффективных преобразований XSLT. Он по праву может считаться единственным в своем роде (пока Netscape не наверстает свое, что не должно заставить себя ждать), предоставляя возможность осуществлять XSLT-преобразования на стороне клиента. Такая возможность предоставляет большой потенциал, переворачивая наши представления о том, как должна распределяться обработка данных между клиентом и сервером, и в то же время предоставляя возможности гораздо более интерактивного и гибкого общения с пользователем.

B

Oracle

Разнообразные технологии XML от Oracle собраны воедино в пакете разработчика XML (XML Developer's Kit, XDK) от Oracle. Он состоит из ряда компонентов, которые можно установить отдельно друг от друга. Одним из таких компонентов является XML-анализатор Oracle для Java; также имеется анализатор Oracle для C++. Оба этих анализатора, несмотря на их названия, в действительности содержат XSLT-процессоры как составную часть.

В этом приложении мы сделаем краткий обзор общей структуры XDK, а затем вернемся к процессору XSLT. Важно понимать контекст, в котором созданы анализаторы от Oracle. В действительности в Oracle не пытались разработать свою XML-технологию как некий самостоятельный продукт, и если вы будете рассматривать каждый компонент по отдельности, то вряд ли найдете его преимущество по сравнению с альтернативными продуктами, поставляющимися с исходными кодами. Основной причиной использования XML-продуктов от Oracle, кроме, возможно, ценимой вами технической поддержки, является то, что они хорошо сочетаются с другими продуктами платформы Oracle.

Набор разработчика XML от Oracle

За полным описанием пакета XDK обращайтесь к книге издательства Wrox Press «Написание приложений в Oracle 8i для профессионалов» («Professional Oracle 8i Application Programming», ISBN 1861004842). В этой книге мы можем лишь сделать краткий обзор, который поможет вам ознакомиться с продуктом перед тем, как обратиться за подробностями куда-либо еще.

Все эти компоненты можно загрузить с веб-сайта Oracle, посвященного технологиям XML по адресу *http://technet.oracle.com/tech/xml*. Их можно сво-

бодно использовать для собственных экспериментов, но для коммерческого использования нужны соответствующая лицензия и соглашение о поддержке. Прежде чем загрузить программное обеспечение, вам придется пройти достаточно длинную процедуру регистрации, предоставляя информацию о вас и вашей компании и отвечая на многочисленные вопросы: не являетесь ли вы осужденным террористом, наркодилером или жителем Ливии (если вы ответите утвердительно, то прежде чем продолжить, вас вежливо попросят изменить свой ответ).

Никакого исходного кода не предоставляется, даже если вы не террорист.

Анализаторы

Oracle предоставляет анализаторы для языков Java, C++ (также поддерживается C) и PL/SQL.

Анализаторы для Java и C++ включают в себя процессоры XSLT. Они оба поддерживают интерфейсы DOM и SAX. К моменту написания книги выпускаемые анализаторы поддерживали SAX 1.0 и DOM 1.0, но уже существовала бета-версия анализатора для Java (версии 2.1), которая поддерживала SAX 2.0 и DOM 2.0. Эту версию можно загрузить с того же веб-сайта. Все анализаторы могут работать в режимах с проверкой действительности и без проверки.

Большая часть данного приложения посвящена описанию XSLT-процессора, включенного в анализатор для Java. Я даже не пытаюсь описывать здесь процессор XSLT для C++, по той простой причине, что я не смог извлечь из предоставляемой Oracle документации информации о том, как его установить и заставить работать. Если вы хорошо знакомы с процедурами компоновки и установки приложений C++ на используемой вами платформе, то, возможно, вам удастся преодолеть все преграды и установить этот продукт, но даже в этом случае вы обнаружите, что документация по работе с процессором весьма скудная. К примеру, когда я загрузил данный продукт, то обнаружил, что файл `XSLProcessor.html` пуст. Если вы не увлеченная натура, считающая, что это вызов вашему интеллекту, и не водите дружбу с консультантом Oracle, который имеет доступ к информации для избранных, то мой совет – дождитесь, пока данный продукт не будет комплектоваться более полно.

Существует два анализатора для PL/SQL, вызов которых может осуществляться из хранимых процедур Oracle. Один из них – просто оболочка для Java-анализатора. А второй – это непосредственно анализатор для PL/SQL, поддерживающий интерфейс DOM и совместимый с Oracle версии 7.3.4 и выше. Однако у этого анализатора более ограниченная функциональность. Это все, что я собирался сказать о продуктах для PL/SQL, поскольку они не относятся напрямую к применению XSLT.

Поддержка XML-схем

Компания Oracle одной из первых осуществила реализацию спецификации XML-схем от W3C, находящейся в данный момент в статусе Кандидата в ре-

комендацию. Схемы XML вполне могут заменить DTD, будучи более предпочтительным способом определения и проверки действительности структуры документа.

Процессор схем нужно загружать отдельно, хотя он и не имеет своего собственного API: он вызывается автоматически Java-анализатором XML, если в исходном документе встречается ссылка на схему. Все, что требуется сделать – это поместить путь к процессору схем (файл `xschema.jar`) в переменную окружения `CLASSPATH`, а остальное сделает анализатор.

Страницы XSQL

В отличие от других компонентов XDK, в которых реализованы признанные стандарты W3C, XSQL является собственной технологией Oracle. Как можно ожидать от компании, специализирующейся на технологиях баз данных, данный продукт представляет собой часть претенциозной попытки соединить реляционные возможности SQL с иерархической структурой XML.

По сути дела XSQL-страница является XML-документом, содержащим запросы SQL. Например:

```
<?xml version 1.0?>
<xsql:query xmlns="urn:oracle-xsql" connection="productdb">
   SELECT description FROM product WHERE code='PZ662431'
</xsql:query>
```

Имя соединения (`productdb`) указывает на запись в файле конфигурации, которая описывает параметры соединения с базой данных.

XSQL-страница расположена на сервере подобно страницам ASP или JSP. Когда пользователь запрашивает эту страницу, указывая соответствующий URL в адресной строке броузера, то вместо передачи самой страницы веб-сервер исполняет логику данной страницы, а затем уже возвращает результат. XSQL-страница исполняется с помощью сервлета, и в файле последних сведений, поставляемом вместе с продуктом, объясняется, как использовать XSQL с множеством популярных веб-серверов, которые поддерживают API сервлетов. Существует также автономный сервер Web-to-Go, включенный в дистрибутив для Windows 2000.

Запуск страницы из вышеприведенного примера приводит к выполнению запроса к базе данных, результат которого возвращается броузеру в виде XML-документа:

```
<?xml version="1.0"?>
<ROWSET>
  <ROW num="1">
    <description>High-speed electric drill</description>
  </ROW>
</ROWSET>
```

Конечно же, существует множество дополнительных элементов и атрибутов, которые могут применяться на странице запроса для воздействия на возвра-

щаемые данные, но данный тривиальный пример помогает понять основную идею.

XSQL-страница может непосредственно ссылаться на параметры, указанные в URL, поэтому, если URL имеет следующий вид:

```
http://www.megacorp.com/xsql/catalog.xsql?prodcode=PZ87453
```

то SQL-запрос будет выглядеть следующим образом:

```
<xsql:query xmlns="urn:oracle-xsql" connection="productdb">
   SELECT description FROM product WHERE code='{@prodcode}'
</xsql:query>
```

Можно, конечно, преобразовать результат запроса из XML в HTML при помощи таблицы стилей XSLT. Для этого нужно просто добавить в страницу XSQL инструкцию обработки <?xml-stylesheet?>. Фактически можно добавить несколько таких инструкций обработки, указав различные таблицы стилей для разных запрашивающих агентов (что в действительности означает сочетание устройства клиента и броузера). Например:

```
<?xml version 1.0?>
<?xml-stylesheet type="text/xsl" media="Mozilla" href="navigator.xsl"?>
<?xml-stylesheet type="text/xsl" media="Spyglass" href="mosaic.xsl"?>
<?xml-stylesheet type="text/xsl" href="default.xsl"?>
<xsql:query xmlns="urn:oracle-xsql" connection="productdb">
   SELECT description FROM product WHERE code='{@prodcode}'
</xsql:query>
```

В качестве альтернативы можно передавать имя требуемой таблицы стилей как параметр в строке URL. Там же можно указать, где нужно осуществлять преобразование: на сервере или на клиенте. К примеру, в следующем URL (введенном в одной строке) дается указание использовать таблицу стилей `iexplorer.xsl` и осуществлять преобразование в броузере:

```
http://www.megacorp.com/xsql/catalog.xsql?prodcode=PZ87453&
   xml-stylesheet=iexplorer&transform=client
```

Помните, что когда вы используете такой URL в HTML-странице или в таблице стилей XSLT, символ «&» надо писать как «&». HTML-броузеры воспримут и «&», хотя такая запись и является технически некорректной, но в XML правила более строгие.

Утилита Oracle XML SQL

Код сервлета, в котором реализованы страницы XSQL, описанные в предыдущем разделе, является на самом деле не более чем тонкой надстройкой над подсистемой, называемой утилитой Oracle XML SQL. Доступ к этой подсистеме можно также осуществлять с помощью командной строки или через Java API.

Две основные функции этой подсистемы представлены Java-классами `OracnsleXMLQuery`, формирующими XML-документ в ответ на запрос SQL, и `OracleXMLSave`, который выполняет обратное: загружает данные в базу из документа XML.

Oracle определяет стандартное представление реляционных данных в виде XML: простейший способ увидеть, как ваши собственные таблицы будут представлены в этом виде, – это поместить простой запрос типа «`SELECT * FROM products`» в XSQL-страницу и посмотреть результаты непосредственно в Internet Explorer, без какой-либо таблицы стилей. По сути дела, это трехуровневая структура: элемент `<ROWSET>` для определения таблицы, элемент `<ROW>` для каждой записи, и элемент, чье имя совпадает с названием поля SQL для определения каждого столбца внутри записи. Можно использовать различные опции для изменения имен элементов и атрибутов, использованных в данном представлении, или же просто оставить все как есть, возложив всю работу по преобразованию в какую-нибудь другую форму на таблицу стилей XSLT.

После того как вы разобрались с форматом, в котором выдает XML-данные инструмент `OracleXMLQuery`, вы можете писать таблицы стилей, которые будут конвертировать данные из других источников в этот формат. Получившиеся XML-файлы можно затем использовать для загрузки информации в базу данных при помощи инструмента `OracleXMLSave`.

Генераторы классов XML

В то время как утилита Oracle XML SQL обеспечивает соответствие между структурами реляционных данных и XML-данных, генераторы классов Oracle осуществляют соответствие между структурой XML и определениями классов Java или C++.

В настоящее время генераторы классов чаще используют DTD, нежели схемы XML. На основе DTD, используемого на входе рассматриваемого средства, формируются классы Java или C++ для каждого определения элемента в DTD.

Идеей данного средства является написание приложений на Java или C++ в терминах реальных объектов предметной области приложения (например, Customer (заказчик), Account (счет) или Invoice (счет-фактура)), вместо использования таких абстрактных объектов, как `Element`, `Attribute` или `Document`, определенных в DOM. Должна существовать возможность использовать классы, формируемые генератором классов Oracle, добавив туда бизнес-логику, вместо статичных объектов данных, отражающих структуру XML, в этих классах могут быть методы, соответствующие поведению приложения.

Но мы больше не будем обсуждать этот аспект, потому что я не хочу отвлекать ваше внимание от написания этой самой логики на языке более высокого уровня, а именно на XSLT!

XSLT–процессор, написанный на Java

XSLT-процессор от Oracle поставляется вместе с программным обеспечением для XML-анализатора. Он доступен по адресу *http://technet.oracle.com/tech/xml/*. XSLT-процессор на Java является частью продукта **XML: Parser for Java v2**. Этот продукт по-прежнему предназначен для предварительного тестирования, хотя, на самом деле он пользуется популярностью уже довольно давно. К концу 2001 года уже можно ожидать новую версию, которая будет полностью интегрирована с Oracle 9i, – следите за событиями на веб-сайте Oracle. Лицензия на свободное использование дается только для внутренней обработки данных, однако при определенных условиях возможно и распространение третьей стороне. Исходный код не предоставляется, а формальная поддержка доступна только в том случае, когда имеется отдельное соглашение с Oracle о поддержке. Тем не менее, вы можете посетить активный форум, где команда разработчиков (включая главного архитектора, Стива Мюнча (Steve Muench)) регулярно отвечает на вопросы пользователей.

Помимо XSLT-процессора в продукт Oracle входит XML-анализатор, поддерживающий интерфейсы DOM и SAX, с поддержкой рекомендации для пространств имен XML. Анализатор функционирует как в режиме с проверкой на действительность, так и без нее. Текущий промышленный релиз, 2.0, поддерживает DOM 1.0 и SAX 1.0. Но в существующей бета-версии, 2.1, поддерживаются DOM 2.0 и SAX 2.0.

В компании Oracle заявляют, что их реализация XSLT полностью совместима с окончательными (ноябрь 1999 года) рекомендациями XSLT 1.0 и XPath 1.0. Как и в случае с другими производителями, вам остается лишь принять подобные заявления на веру, так как в данный момент не производится независимого тестирования на соответствие стандартам.

В Oracle мало говорят о том, как воплощены возможности, которые стандарт оставляет необязательными или зависящими от производителя, поэтому в таких ситуациях вам придется действовать методом проб и ошибок. Вообще, документация к рассматриваемому продукту на удивление скупа: это больше похоже на старания новичка, нежели на работу второй по размерам компании, производящей программное обеспечение. Будьте готовы к изучению в ключевых местах API-документации `javadoc` или к поиску ответа на свой вопрос в форумах. Не сомневайтесь, ситуация изменится к лучшему, как только данная технология получит статус продукта.

Интерфейс командной строки

XSLT-процессор Oracle можно запускать с помощью простого интерфейса командной строки. Если у вас установлена виртуальная машина Java и программное обеспечение от Oracle, а также установлены правильные значения переменных окружения `PATH` и `CLASSPATH`, то можно запустить процессор из командной строки следующим образом:

```
java oracle.xml.parser.v2.oraxsl source.xml stylesheet.xsl result.xml
```

Команда `java`, как правило, запускает виртуальную машину Java от Sun. Если вы предпочитаете использовать виртуальную машину Microsoft's Java VM (которая уже установлена на вашей машине, если вы пользуетесь последними версиями Internet Explorer), то используйте команду `jview`.

Вот список имеющихся параметров:

Опция	Значение
`-d directory`	Имя каталога, содержащего файлы, подлежащие преобразованию
`-e error-log`	Имя файла, в который будут записываться сообщения об ошибках
`-i source-extension`	Расширения включаемых исходных файлов
`-l xml-file-list`	Список исходных файлов для преобразования
`-o directory`	Имя каталога, в который будут записаны конечные файлы
`-p parameter-list`	Список параметров таблицы стилей, в форме имя=значение (см. ниже)
`-r result-extension`	Расширение, используемое для конечных файлов
`-s stylesheet`	Имя файла таблицы стилей
`-t number-of-threads`	Количество используемых потоков
`-v`	Подробный режим
`-w`	Показывать предупреждения
`-x source-extension`	Расширения исключаемых исходных файлов

Параметры таблицы стилей можно указать с помощью параметра `-p`. Я не смог найти в документации, как правильно применять этот параметр, но метод проб и ошибок показал, что нижеприведенная инструкция работает (для запуска таблицы стилей, описывающей обход шахматной доски конем из главы 10, начальная позиция коня – квадрат d3):

```
java oracle.xml.parser.v2.oraxsl -p start='d3' dummy.xml tour10.xsl out.html
```

Как и в других продуктах, интерфейс командной строки предназначен служить инструментом разработчика. Для промышленного использования нужно написать приложение, в котором непосредственно вызывается Java API. В следующем разделе описывается, как это сделать.

API для преобразований

В XSLT-процессоре от Oracle пока не реализован интерфейс TrAX API, описанный в приложении F, так же как и другие части интерфейса JAXP 1.1, хотя Стив Мюнч (Steve Muench) в ответ на вопросы пользователей в форуме утверждает, что такая реализация входит в планы новой версии анализатора, которая будет поставляться с Oracle 9i. Поэтому информация данного

раздела носит временный характер: следите за новостями, касающимися API, на веб-сайте Oracle. Помимо преимущества переносимости приложений между различными XSLT-процессорами, интерфейс API TrAX более удобен в использовании, так как он лучше документирован по сравнению с Oracle API.

Подробности об используемом в настоящий момент API можно узнать в документации `javadoc` для данного продукта, там же имеется несколько примеров приложений для введения в курс дела. Двумя ключевыми классами являются класс `XSLStylesheet`, представляющий собой предварительно обработанную (или скомпилированную) таблицу стилей, которую можно использовать неоднократно, и класс `XSLProcessor`, используемый для применения таблицы стилей к исходному документу. Объект `XSLStylesheet` является аналогом объекта `Templates` их интерфейса TrAX, в то время как объект `XSLProcessor` является аналогом TrAX `Transformer`.

Обработку можно начать с создания объекта `XSLStylesheet`. Для этих целей имеется четыре конструктора. Во всех случаях первый аргумент означает источник таблицы стилей, а второй – URL, который можно использовать в качестве базового URI для разрешения относительных URI, имеющихся в элементах `<xsl:include>` и `<xsl:import>`.

В качестве первого аргумента могут выступать:

- Объект `XMLDocument` (то есть документ DOM, построенный с помощью анализатора Oracle)
- URL, указывающий расположение исходной таблицы стилей
- Объект `InputStream`, из которого может быть прочитана таблица стилей
- Объект `Reader`, из которого может быть прочитана таблица стилей

Объект `XSLStylesheet` представляет собой откомпилированную форму таблицы стилей. Он может быть использован столько раз, сколько потребуется для преобразования различных исходных документов. Эти преобразования могут осуществляться последовательно или параллельно в нескольких потоках. Для каждого преобразования требуется отдельный объект `XSLProcessor`.

(Так обстоит дело в теории. К сожалению, если преобразование требует передачи параметров, то их нужно передавать в объект `XSLStylesheet`, а не `XSLProcessor`. Это означает, что выполнение параллельных преобразований с применением одной и той же таблицы стилей возможно только в тех случаях, когда все они используют одни и те же значения параметров.)

Объект `XSLProcessor` создается с помощью конструктора без аргументов: изначально он не связан с какой-либо таблицей стилей. После этого можно произвести преобразование с помощью одного из многочисленных методов `processXSL()`, перечисленных в нижеприведенной таблице:

Метод	Действие
`XMLDocumentFragment processXSL( XSLStylesheet xsl, java.io.InputStream xml, java.net.URL baseURI)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ считывается из потока `InputStream`, используя переданный базовый URI для разрешения относительных URI в документе. Возвращаемое значение представляет собой объект DOM `DocumentFragment`.
`XMLDocumentFragment processXSL( XSLStylesheet xsl, java.io.Reader xml, java.net.URL baseURI)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ считывается из объекта `Reader`, используя переданный базовый URI для разрешения относительных URI в документе. Возвращаемое значение представляет собой объект DOM `DocumentFragment`.
`XMLDocumentFragment processXSL( XSLStylesheet xsl, java.net.URL xml, java.net.URL baseURI)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Первый URL указывает местоположение считываемого документа, переданный базовый URI используется для разрешения относительных URI в документе. Возвращаемое значение представляет собой объект DOM `DocumentFragment`.
`XMLDocumentFragment processXSL( XSLStylesheet xsl, XMLDocument xml)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ указывается в форме DOM. Возвращаемое значение представляет собой объект DOM `DocumentFragment`.
`void processXSL( XSLStylesheet xsl, XMLDocument xml, ContentHandler handler)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM. Конечное дерево передается в виде потока в указанный объект SAX2 `ContentHandler`.
`XMLDocumentFragment processXSL( XSLStylesheet xsl, XMLDocumentFragment inp)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `DocumentFragment`. Возвращаемое значение представляет собой объект DOM `DocumentFragment`.
`void processXSL( XSLStylesheet xsl, XMLDocumentFragment xml, java.io.OutputStream os)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `DocumentFragment`. Результат записывается в указанный объект `OutputStream`.
`void processXSL( XSLStylesheet xsl, XMLDocumentFragment xml, java.io.PrintWriter pw)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `DocumentFragment`. Результат записывается в указанный объект `PrintWriter`.

Метод	Действие
`void processXSL( XSLStylesheet xsl, XMLDocumentFragment inp, XMLDocumentHandler handler)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `DocumentFragment`. Конечное дерево передается в виде потока в указанный объект SAX1 `DocumentHandler`.
`void processXSL( XSLStylesheet xsl, XMLDocument xml, java.io.OutputStream os)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `Document`. Результат записывается в указанный объект `OutputStream`.
`void processXSL( XSLStylesheet xsl, XMLDocument xml, java.io.PrintWriter pw)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `Document`. Результат записывается в указанный объект `PrintWriter`.
`void processXSL( XSLStylesheet xsl, XMLDocument xml, XMLDocumentHandler handler)`	Преобразует входной XML-документ с помощью указанной таблицы стилей. Входной документ передается в форме DOM `Document`. Конечное дерево передается в виде потока в указанный объект SAX1 `DocumentHandler`.

Учтите, что в то время как процессор Oracle XSLT может принимать на входе исходное дерево в форме DOM `Document` или `DocumentFragment`, это должны быть объекты `Document` или `DocumentFragment`, которые были созданы с помощью инструментов Oracle.

Типичной последовательностью для выполнения преобразования будет приведенный ниже пример кода.

```
// создаем анализатор и настраиваем его

parser = new DOMParser();
parser.setPreserveWhitespace(true);
// формируем из исходного XSLT-файла DOM-документ
xslURL = createURL(args[0]);
parser.parse(xslURL);
xsldoc = parser.getDocument();

// формируем из исходного XML-файла DOM-документ
xmlURL = createURL(args[1]);
parser.parse(xmlURL);
xmldoc = parser.getDocument();
// обрабатываем таблицу стилей
XSLStylesheet xsl = new XSLStylesheet(xsldoc, xslURL);

// создаем XSL-процессор и настраиваем его
XSLProcessor processor = new XSLProcessor();
processor.showWarnings(true);
processor.setErrorStream(System.err);
// выполняем преобразование
```

```
DocumentFragment result = processor.processXSL(xsl, xmldoc);
// создаем выходной документ, в который помещается
// результат преобразования
out = new XMLDocument();
// создаем для выходного документа пустой
//элемент документа
Element root = out.createElement("root");
out.appendChild(root);
// добавляем к пустому элементу документа
// преобразованное дерево
root.appendChild(result);

// выводим преобразованный документ
out.print(System.out);
```

Другим способом выполнения преобразования будет вызов метода `transformNode()` объекта `XMLNode` (реализация объекта `org.w3c.dom.Node` от Oracle). Этот метод принимает объект `XSLStylesheet` в качестве параметра и возвращает объект `DocumentFragment`.

Расширения производителя

Компания Oracle предоставляет два расширения XSLT 1.0: функцию для преобразования фрагмента конечного дерева в набор узлов и элемент расширения для создания нескольких выходных файлов.

Они не очень хорошо документированы; фактически я смог найти ссылку на них только в паре строчек, затерянных среди длинного списка об исправленных ошибках в файле `readme`.

Функция расширения oracle:node-set()

Эта функция имеет то же предназначение, что и функции `node-set()` в других популярных продуктах XSLT 1.0: в качестве аргумента она принимает фрагмент конечного дерева и возвращает набор узлов, содержащий единственный узел, корень дерева.

Эта функция заметно увеличивает возможности языка, так как это означает, что вы можете производить преобразование в несколько этапов: результат первого этапа – фрагмент конечного дерева, который может быть преобразован в набор узлов для передачи на следующий этап. Эта функция устареет в XSLT 1.1, который позволяет неявное преобразование фрагмента конечного дерева в набор узлов при необходимости.

С этой функцией можно использовать любой префикс пространства имен, если он объявлен с использованием следующего URI пространства имен: *http://www.oracle.com/XSL/Transform/java*.

Пример: Применение функции oracle:node-set()

Здесь приводится пример таблицы стилей, которая использует функцию `oracle:node-set()`. Ее применение не несет никакой практической пользы, но позволяет продемонстрировать функциональность.

Исходный документ

Нижеприведенную таблицу стилей можно применить к любому исходному документу. (Можно, например, использовать саму таблицу стилей в качестве исходного документа.)

Таблица стилей

Таблица стилей находится в файле `oracle-nodeset.xsl`:

```
<xsl:stylesheet
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
<xsl:template match="/">
  <xsl:variable name="дерево">
    <table><tr><td val="85"/></tr></table>
  </xsl:variable>
  <xsl:variable name="ns" select="oracle:node-set($дерево)"
      xmlns:oracle="http://www.oracle.com/XSL/Transform/java"/>
  <вывод><xsl:value-of select="$ns//td/@val"/></вывод>
</xsl:template>
</xsl:stylesheet>
```

Результат

Результатом применения таблицы стилей будет следующий документ:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<вывод>85</вывод>
```

Элемент расширения oracle:output

Элемент расширения предусмотрен для формирования нескольких выходных файлов. Таким образом, он соответствует элементу `<xsl:document>`, введенному в XSLT 1.1.

Oracle решила разделить эту функцию на две части: одна представляет собой инструкцию (используемую внутри тела шаблона), которая используется для выбора места вывода, а другая – это элемент верхнего уровня, описывающий формат выходного файла. Обе эти функции выполняются элементами с именем `<oracle:output>`, но инструкция `<oracle:output>` имеет атрибут `use`, который ссылается на атрибут `name` элемента `<oracle:output>` верхнего уровня. Элемент верхнего уровня может содержать любые атрибуты, разрешенные для `<xsl:output>`, например метод, кодировку или отступ. Имя выходного файла определяется атрибутом `href` инструкции `<oracle:output>`.

В следующем примере продемонстрирована работа этого элемента. Он выполняет ту же функцию, что и <xsl:document> в примере, приведенном в главе 4, выполняя разбиение стихотворения на строфы, однако используется синтаксис Oracle.

Пример: Применение элемента расширения oracle:output

Здесь приводится пример таблицы стилей, которая использует элемент расширения `oracle:output` для разбиения стихотворения на строфы, которые выводятся в отдельные файлы.

Исходный файл

Исходный файл `стих.xml` – это стихотворение Руперта Брука (которое, при внимательном чтении книги, вы уже знаете наизусть). Этот файл содержит элемент <стихотворение>, который, помимо всего прочего, выступает в качестве родительского элемента для нескольких элементов <строфа>.

Таблица стилей

Таблица стилей находится в файле `oracle-split.xsl`. Каждая строфа из стихотворения выводится в отдельный файл, а основной выходной файл содержит ссылки на файлы со строфами.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0"
    xmlns:oracle="http://www.oracle.com/XSL/Transform/java"
    extension-element-prefixes="oracle">

<xsl:template match="стихотворение">
    <стихотворение>
        <xsl:copy-of select="заголовок"/>
        <xsl:copy-of select="автор"/>
        <xsl:copy-of select="дата"/>
        <xsl:apply-templates select="строфа"/>
    </стихотворение>
</xsl:template>

<xsl:template match="строфа">
    <xsl:variable name="файл"
                  select="concat('строфа', position(), '.xml')"/>
    <строфа номер="{position()}" href="{$файл}"/>
    <oracle:output href="{$файл}" use="формат-строфы">
            <xsl:copy-of select="."/>
    </oracle:output>
</xsl:template>

<oracle:output name="формат-строфы" method="xml" encoding="iso-8859-1"/>

</xsl:stylesheet>
```

Результат

Основной выходной файл выглядит следующим образом (отступы даны для наглядности):

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<стихотворение xmlns:oracle="http://www.oracle.com/XSL/Transform/java">
  <заголовок>Song</заголовок>
  <автор>Руперт Брук</автор>
  <дата>1912</дата>
  <строфа номер="1" href="строфа1.xml"/>
  <строфа номер="2" href="строфа2.xml"/>
  <строфа номер="3" href="строфа3.xml"/>
</стихотворение>
```

Так же формируется по одному файлу для каждой строфы; так, например, файл `строфа1.xml` выглядит следующим образом:

```
<?xml version = '1.0' encoding = 'iso-8859-1'?>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
```

Расширяемость

Версия XSLT-процессора от Oracle, написанная на Java, позволяет применять пользовательские функции расширения, написанные на Java. На данный момент отсутствуют механизмы для создания пользовательских элементов расширения.

Функцию расширения можно вызвать из выражения XPath, с помощью следующего синтаксиса: `префикс:метод(аргументы)`. Под обозначением `префикс` здесь подразумевается префикс пространства имен, который был объявлен с помощью объявления пространства имен в форме:

```
xmlns:префикс="http://www.oracle.com/XSL/Transform/java/package.name.Classname"
```

Например, для вызова методов, определенных в Java-классе `java.util.Date`, нужно объявить пространство имен следующим образом:

```
xmlns:Date="http://www.oracle.com/XSL/Transform/java/java.util.Date"
```

Локальной частью имени, используемого при вызове функции XPath, должно быть имя метода, определенного в выбранном классе, или `new` в случае конструктора.

Для конструктора или для статических методов передаваемые в функцию XPath аргументы должны в точности соответствовать аргументам, объявленным в сигнатуре Java-метода.

Если внешняя функция возвращает Java-объект, который не может быть преобразован в один из типов данных XPath, то его следует хранить в таблице стилей как внешний объект. Внешний объект можно хранить в параметре или переменной XPath и передавать в качестве аргумента в другие внешние методы.

Для метода экземпляра (нестатического метода) вызов функции XPath будет содержать дополнительный первый аргумент, который отсутствует в сигнатуре Java-метода: этот аргумент задает внешний объект Java, выступающий в роли целевого объекта для вызова метода.

Передаваемые аргументы XPath будут автоматически преобразованы к нужным типам данных Java. Ниже приводится непосредственное соответствие типов данных:

Тип данных Java	Тип данных XPath
`java.lang.String`	строка
`int, double, float`	число
`boolean`	логический
`org.w3c.dom.NodeList` (см. примечание)	набор узлов
`org.w3c.dom.DocumentFragment`	фрагмент конечного дерева

Примечание: в документации говорится о типе `XMLNodeList`, являющемся реализацией данного интерфейса от Oracle; но класс `XMLNodeList` не является открытым, поэтому вы не можете его использовать. Я попробовал использовать непосредственно интерфейс DOM `NodeList`, и все прекрасно работало. Точно так же, в документации говорится о классе реализации `XMLDocumentFragment`, но с таким же успехом работает интерфейс DOM `DocumentFragment`.

Если типы аргументов не соответствуют напрямую, то предпринимается попытка преобразования согласно правилам XPath. Например, если ожидается тип `String`, а передается набор узлов, то этот набор преобразуется в строку в соответствии с правилами для XPath-функции `string()`.

В ограниченных пределах допускается перегрузка методов. Класс может содержать несколько методов с одинаковыми именами (или несколько конструкторов), если они имеют различное количество аргументов. Он также может содержать несколько методов (или конструкторов) с одинаковыми именами и количеством аргументов, при условии, что между соответствующими типами данных не допускается преобразования. Например, класс `java.lang.Math` имеет четыре метода `max()`, каждый из которых ожидает два аргумента, а именно `max(int, int)`, `max(double, double)`, `max(long, long)` и `max(float, float)`. Эти методы нельзя вызвать непосредственно из таблицы стилей, потому что непонятно, какой именно метод должен быть вызван.

Возвращаемым значением может быть любое значение или объект Java. Если его можно преобразовать к одному из типов данных XSLT 1.0 (строка,

число, логическое значение, набор узлов или фрагмент конечного дерева), то возвращаемое значение будет иметь этот тип; в противном случае оно будет храниться как внешний объект.

Примеры

В приведенном ниже коде переменной присваивается случайное значение в диапазоне от 0.0 до 1.0. В примере вызывается статический метод класса `java.lang.Math`.

```
<xsl:variable name="random" select="Math:random()"
xmlns:Math=="http://www.oracle.com/XSL/Transform/java/java.lang.Math"/>
```

Далее следует код, выводящий текущую дату и время в формате, определенном в Java по умолчанию. В данном примере вызывается конструктор по умолчанию класса `java.util.Date`. Получившийся объект `Date` рассматривается в XPath как внешний объект. Он сразу же передается в качестве аргумента в другую внешнюю функцию, метод `toString()` класса `java.util.Date`, который создает представление даты в виде, пригодном для печати. Этот метод экземпляра не ожидает аргументов, поэтому указанный в вызове функции объект используется в качестве целевого объекта для вызова метода.

```
<xsl:value-of select="Date:toString(Date:new())"
xmlns:Date="http://www.oracle.com/XSL/Transform/java/java.util.Date"/>
```

И, наконец, функция расширения, которую я написал до того, как обнаружил затерявшееся в документации упоминание о функции `oracle:node-set()`. Она работает не совсем как функция `oracle:node-set()`, которая возвращает единственный узел, корень дерева: эта функция `children()` возвращает набор узлов, содержащий всех непосредственных потомков корня дерева.

Здесь приводится код на языке Java, который можно найти в файле `OracleChildren.java`. Да, эта мощная функция умещается в одну строчку:

```
import org.w3c.dom.*;
public class OracleChildren {
    public static NodeList children(DocumentFragment tree) {
        return tree.getChildNodes();
    }
}
```

А ниже приводится таблица стилей, в которой применяется эта функция, `oracle-children.xsl`. Вы можете применить эту таблицу стилей к любому исходному документу, если путь к классу `OracleChildren` прописан в переменной `CLASSPATH`. На выходе имеем `<вывод>85</вывод>`:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <xsl:variable name="дерево">
    <table><tr><td val="85"/></tr></table>
  </xsl:variable>
  <xsl:variable name="ns" select="my:children($дерево)"
```

```
        xmlns:my="http://www.oracle.com/XSL/Transform/java/OracleChildren"/>
    <вывод><xsl:value-of select="$ns//td/@val"/></вывод>
  </xsl:template>
  </xsl:stylesheet>
```

Поддержка XPath

Тщательно покопавшись в документации javadoc для рассматриваемого продукта, вы обнаружите, что класс XMLNode, являющийся реализацией стандартного интерфейса org.w3c.dom.Node, поддерживает методы для выборки узлов из DOM-документа с помощью выражений XPath, независимо от процессора XSLT.

Вот эти четыре метода:

Метод	Действие
selectNodes(String expression)	Вычисляет указанное выражение, возвращая результат в виде DOM NodeList
selectNodes(String expression, NSResolver resolver)	Вычисляет указанное выражение, применяя указанный объект NSResolver для разрешения любых префиксов пространств имен, и возвращает результат в виде DOM NodeList
selectSingleNode(String expression)	Вычисляет указанное выражение, возвращая результат в виде DOM Node
selectSingleNode(String expression, NSResolver resolver)	Вычисляет указанное выражение, применяя указанный объект NSResolver для разрешения любых префиксов пространств имен, и возвращает результат в виде DOM Node

NSResolver представляет собой объект, который может преобразовать префикс пространства имен в его URI. При этом удобно, что любой объект XMLElement или XMLDocument (реализации узлов DOM Element и Document от Oracle) может использоваться как NSResolver.

Как это часто случается с документацией, полученной из комментариев в исходном коде, опубликованные описания javadoc для этих методов, к сожалению, неточны. В документации говорится, что первый аргумент является образцом XSL, и методом возвращаются все узлы, которые соответствуют этому образцу. В действительности это не совсем так; если аргумент передается в виде "автор", то согласно документации будут возвращены все элементы <автор>, присутствующие в документе. Метод проб и ошибок показал, что на самом деле метод возвращает элементы <автор>, являющиеся непосредственными потомками контекстного узла (то есть узла, к которому применяется метод). Другими словами, система рассматривает аргумент как выражение XPath, где целевой узел выступает в качестве контекстного, – что в точности соответствует поведению одноименных методов Microsoft.

Дальнейшее исследование показало, что передача выражения XPath, которое не возвращает набора узлов, может либо вызвать исключение XSLException, либо возвратить пустое значение, поэтому следует учесть обе возможности.

В следующем небольшом приложении (которое, слегка улучшив, я позаимствовал из книги издательства Wrox «Professional Oracle 8i Application Programming») используется метод `selectNodes()`, позволяющий вводить выражения XPath с консоли и видеть на экране выбранные узлы. В этом выражении корень документа всегда рассматривается как контекстный узел. Это приложение можно найти (как в исходной, так и в скомпилированной форме) среди файлов с примерами для данного приложения.

```
import java.io.*;
import oracle.xml.parser.v2.*;
import org.xml.sax.*;
import org.w3c.dom.*;

public class XPathToy {
    private  XMLDocument document;
    private  String query;
public XPathToy(String filename) {
    try {
        document = loadXMLDocument(filename);
        BufferedReader in =
           new BufferedReader(new InputStreamReader(System.in));
        do {
            System.out.print("XPath> ");
            query = in.readLine();
            try {
                NodeList nodes = ((XMLNode)document).selectNodes(
                            query,
                           (NSResolver)document.getDocumentElement());
                if (nodes==null) {
                    System.out.println("Выражение должно возвращать набор узлов");
                } else {
                    for (int i = 0; i < nodes.getLength(); i++) {
                        System.out.println(
                            "Имя:  " + nodes.item(i).getNodeName());
                        System.out.println(
                            "Значение: " + nodes.item(i).getNodeValue());
                    }
                }
            } catch (XSLException e) {
                System.out.println("Неверное выражение: " + query);
            }
        } while (!(query.equals("quit")));
    } catch (IOException e) {
        System.err.println(
            "Невозможно загрузить '" + filename + "': " + e.toString());
    } catch (SAXException e) {
        System.err.println("Невозможно проанализировать XML: " + e.toString());
    }
}

private XMLDocument loadXMLDocument(String filename)
throws  IOException, SAXException {
```

```
        File file = new File(filename);
        FileReader reader = new FileReader(file);

        DOMParser parser = new DOMParser();
        parser.parse(reader);

        return parser.getDocument();
    }

    public static void main(String[] args) {
        if (args.length == 1)
            new XPathToy(args[0]);
        else
            System.err.println("Использование: java XPathToy <имя_файла>");
    }

    }  // конец класса XPathToy
```

Для запуска этого приложения убедитесь, что в переменную окружения CLASSPATH занесены как каталог, в котором находится класс XPathToy, так и файл xmlparserv2.jar, и после этого введите в командной строке:

```
java XPathToy poem.xml
```

Приложение предложит вам ввести выражение XPath. Введите, например, следующее:

```
XPath>//строфа[1]/строка[2]/текст()
```

и получите в ответ:

```
Имя:  #text
Значение: And Spring is here again;
```

Чтобы выйти из приложения, введите «quit».

Имена и значения, которые выводятся этим игрушечным приложением, возвращаются DOM-методами getNodeName() и getValue(). Они не эквивалентны именам и значениям XPath, например значение узла элемента всегда пустое. Для того чтобы получить текст внутри узла элемента, нужно использовать функцию «text()» как часть выражения XPath, как показано выше.

Кодировки символов

Анализатор от Oracle в настоящий момент поддерживает следующие кодировки как для входных, так и для выходных данных:

BIG 5	ISO-2022-KR
EBCDIC-CP-*	От ISO-8859-1 до ISO-8859-9
EUC-JP	KOI8-R
EUC-KR	Shift_JIS

GB2312	US-ASCII
ISO-10646-UCS-2	UTF-16
ISO-10646-UCS-4	UTF-8
ISO-2022-JP	

Можно также использовать любые другие кодировки, основанные на ASCII или EBCDIC, которые поддерживаются виртуальной машиной Java. Однако их нужно указывать с использованием имени Java для кодировок, которое не всегда совпадает с официальным названием наборов символов, определенных IANA.[1]

Заключение

Я стараюсь избегать субъективных оценок различных имеющихся продуктов, но в конце данного приложения я вынужден сказать, что технология XSLT, предлагаемая Oracle, сильно разочаровывает плохим качеством доступной документации. Представляется странным то, что, затратив значительные ресурсы на разработку сложного программного обеспечения, создатели не удосужились объяснить, как им пользоваться. На сегодняшний день создается впечатление, что рассматриваемый продукт от Oracle предназначался для отчаянных энтузиастов. Но не стоит отчаиваться, как только будет ясно, что данная технология готова к массовому показу, официальная документация не заставит себя долго ждать.

Если работа вашего сайта связана с базой данных Oracle, то технология XSQL Pages представляется очень привлекательным способом для передачи информации из базы данных прямо в сеть, без дополнительного процедурного кодирования. Прибавьте к этому XSLT-преобразование извлеченных данных, и вы получите все, что нужно для быстрого создания приложений, и, возможно, при таком положении вещей недостаток подробной документации не является такой уж сильной помехой.

[1] Администрация адресного пространства Internet (Internet Assigned Numbers Authority, IANA). Основная организация, занимающаяся вопросами координации выделения IP-адресов и номеров автономных систем. – *Примеч. перев.*

C

Saxon

Saxon, это реализация XSLT с открытым исходным кодом. Ее создал автор данной книги Майкл Кэй (Michael Kay). Разработка изначально поддерживалась компанией ICL, поставщиком IT-услуг, а затем компанией Software AG, тем не менее, это, по сути, авторский, нежели корпоративный продукт.

Saxon доступен по адресу *http://users.iclway.co.uk/mhkay/saxon/*.

Продукт распространяется на условиях лицензии Mozilla Public License, которая, в действительности, позволяет использовать его бесплатно для любых целей. Также доступен исходный код, который может изменяться и улучшаться согласно лицензии. Никаких гарантий или поддержки не предоставляется, однако существует список рассылки по адресу `http://saxon.xsl.listbot.com/`, куда можно сообщить о возникших проблемах. Продукт написан на Java и работает на любой платформе Java 1.1 или Java 2.

Существует две разновидности продукта: полный продукт Saxon, в поставку которого входят все исходные коды на Java, двоичные файлы, документация по API и примеры приложений; и продукт Instant Saxon, который представляет собой обычный исполняемый файл для платформы Windows. Оба продукта собраны из одного и того же исходного кода и имеют идентичную функциональность.

В текущем релизе, версии 6.2, в полном объеме реализованы рекомендации W3C XSLT 1.0 и XPath 1.0, а также основная часть новых возможностей рабочего проекта XSLT 1.1 по состоянию на 5 декабря 2000 года. В эту версию включено большинство возможностей, которые являются опциональными или зависящими от поставщика (спецификация XSLT свободно расширяема в таких аспектах, как поддержка кодировок символов или схем сортировки). Вместе с продуктом находится подробная документация.

Установка

Instant Saxon поставляется в виде zip-файла, `instant-saxon.zip`. Просто загрузите его в подходящий каталог и откройте с помощью WinZip или другой архивирующей программы. Внутри архива вы найдете файл `saxon.exe`, являющийся исполняемым процессором XSLT. Распакуйте этот файл в подходящий каталог и добавьте его для удобства в переменную `PATH` в файле `autoexec.bat`.

Полный продукт Saxon является гораздо более объемным, поскольку в него входят исходные файлы, полная документация javadoc по API и примеры приложений. Он также распространяется в виде zip-файла, `saxon.zip`. Внутри этого архива содержится важный компонент, файл `saxon.jar`, который нужно распаковать в любой подходящий каталог. После этого нужно добавить `saxon.jar` (а **не** каталог, в котором он содержится) в переменную окружения `CLASSPATH`. В операционной системе Windows 98 это можно сделать, определив переменную `CLASSPATH` в файле `autoexec.bat`; в Windows NT и Windows 2000 переменные окружения устанавливаются с помощью значка Система (System) в панели управления. Если же вы пользуетесь какой-либо другой операционной системой, то я отсылаю вас к справочнику по этой системе.

Saxon не требует какого-либо другого программного обеспечения, кроме виртуальной машины Java. Он будет работать с любым SAX2-совместимым анализатором XML, но по умолчанию использует свой встроенный анализатор, который является немного измененной версией анализатора Ælfred Дэвида Мэгинсона (David Megginson) (это подправленная версия оригинала, который можно найти по адресу *http://www.opentext.com/microstar*; куда также добавлен драйвер SAX2, написанный Дэвидом Браунелом (David Brownell)). Существует интерфейс для подключения процессора форматирующих объектов (Formatting Object Processor, FOP), являющегося реализацией Форматирующих объектов XSL (XSL Formatting Objects), который вы можете получить по адресу *http://xml.apache.org/*, но он является необязательным.

Запуск процессора Saxon

Saxon написан на языке Java и является реализацией интерфейса TrAX API, определенного в JAXP 1.1, который подробно описан в приложении F. Это позволяет вызывать процессор из Java-приложений. Предусмотрен также интерфейс командной строки. Кроме того, Saxon предоставляет оболочку-сервлет, позволяющую запускать таблицу стилей, непосредственно указывая в броузере URL, однако это в большей мере демонстрационное приложение, нежели составная часть продукта.

Графический интерфейс пользователя отсутствует; тем не менее Saxon можно применять с такими графическими средствами, как XML Spy (*http://www.xmlspy.com*), для создания более дружественного интерфейса пользователя.

Работа с Saxon из командной строки

Если вы просто хотите использовать продукт в Windows и не нуждаетесь в спецификациях API, исходном коде или примерах приложений, то проще всего загрузить продукт *Instant Saxon*, который упакован в виде исполняемого файла, предназначенного для работы с виртуальной машиной Java от Microsoft. Тогда можно применить таблицу стилей с помощью следующей команды:

```
saxon source.xml style.xsl >output.html
```

Эту команду можно ввести из командной строки MS-DOS в Windows, но я не рекомендую ею пользоваться, поскольку в ней трудно исправлять ошибки набора и останавливать пробегающий вывод символов на экран. Гораздо лучше установить какой-нибудь текстовый редактор, который поддерживает работу с командной строкой. Я использую два таких редактора: Programmer's File Editor и UltraEdit.

> *Если каталог, содержащий файл* `saxon.exe`*, не является текущим и не прописан в переменной окружения* `PATH`*, то при вызове нужно использовать полное имя файла, например* `C:\saxon-dir\saxon`.

На других платформах или в случае полного продукта Saxon он вызывается как Java-приложение. Тогда команда выглядит следующим образом:

```
java com.icl.saxon.StyleSheet source.xml style.xsl >output.html
```

если вы пользуетесь виртуальной машиной Java от Sun, или

```
jview com.icl.saxon.StyleSheet source.xml style.xsl >output.html
```

если вы предпочитаете Microsoft JVM. Для промышленного использования Sun JDK 1.3 значительно быстрее, чем Microsoft JVM, хотя при единичных вызовах из командной строки Instant Saxon (который использует Microsoft JVM) часто выигрывает по скорости, потому что затраты на запуск ниже.

Существует ряд параметров, которые можно использовать в командной строке. В случае Instant Saxon они указываются непосредственно после слова `saxon` перед именем исходного файла следующим образом:

```
saxon -t -w2 source.xml style.xsl >out.html
```

В полной версии Saxon они указываются после имени класса, например:

```
java com.icl.saxon.StyleSheet -t -w2 source.xml style.xsl >out.html
```

Ниже приводятся имеющиеся параметры командной строки:

Параметр	Описание
`-a`	Использовать инструкцию обработки `<?xml-stylesheet?>` исходного документа для указания применяемой таблицы стилей. В этом случае аргумент, задающий название таблицы стилей, должен быть опущен.

Параметр	Описание
-ds -dt	Выбирает реализацию внутренней модели дерева. -dt означает модель компактного дерева («tinytree» model) (используется по умолчанию). -ds означает обычную модель дерева. Этот параметр регулирует производительность: модель компактного дерева быстрее построить, она занимает меньше памяти, но перемещение по такому дереву иногда медленнее. По умолчанию используется -dt.
-l	Включает нумерацию строк в исходном документе. Номера строк доступны через функцию расширения saxon:line-number() или через трассировку. Нумерация строк будет включена автоматически, если применяется параметр -T.
-m classname	Указывает полное имя Java-класса, применяемого для обработки результатов инструкций <xsl:message>, встречающихся в таблице стилей. Этот класс должен расширять класс Saxon com.icl.saxon.output.Emitter, который разработан на базе интерфейса SAX2 ContentHandler. Если аргумент опущен, то вывод инструкции <xsl:message> поступает в поток System.err.
-o filename	Определяет имя файла, в который следует поместить результат преобразования. Если в качестве исходного файла используется имя каталога, то и выходным файлом должен быть каталог: такой подход позволяет преобразовать все файлы каталога с помощью одной и той же таблицы стилей или с помощью таблиц стилей, связанных с каждым файлом, если указан параметр -a. Если этот аргумент опущен, то вывод поступает в поток System.out (откуда его можно перенаправить в файл с помощью конструкции >имя_файла).
-r classname	Указывает полное имя Java-класса, применяемого для разрешения URI, встречающихся в таблице стилей в функции document() или в элементах <xsl:include> и <xsl:import>. Этот класс также применяется для разрешения URI, определяющих исходный файл и таблицу стилей в командной строке. Этот класс должен быть реализацией интерфейса TrAX javax.xml.transform.URIResolver.
-t	Этот параметр включает отображение на экране информации о версиях применяемых продуктов Saxon и Java, а также информации о ходе обработки файлов и времени, потраченном на основные стадии обработки.
-T	Включает трассировку таблицы стилей. Результат трассировки состоит из перечисления каждой выполненной инструкции и элемента верхнего уровня и номера строки, указывающего их положение в таблице стилей. Результаты трассировки записываются в поток System.err. Эти результаты записываются в формате XML-документа, поэтому для анализа этого документа опять же можно использовать таблицу стилей!
-TL classname	Отслеживает исполнение с помощью определенной пользователем процедуры трассировки. Имя класса должно указывать на класс, реализующий интерфейс com.icl.saxon.trace.TraceListener. Этот интерфейс обеспечивает интеграцию с отладчиками сторонних производителей, например с отладчиком tbug, написанным Эдвином Глейзером (Edwin Glaser) и доступным по адресу *http://tbug.sourceforge.net*.

Параметр	Описание
-u	Указывает на то, что имена исходного документа и таблицы стилей, переданные в командной строке, должны рассматриваться как URL, а не имена файлов. (Если имена начинаются с «http:» или «file:», это будет предполагаться автоматически.)
-w0	Указывает на то, что если восстановление после ошибки в таблице стилей возможно, Saxon должен предпринимать восстановительные действия, определенные в спецификации XSLT, без предупреждения пользователя. Если, к примеру, один узел соответствует двум шаблонным правилам с одинаковым преимуществом и приоритетом, то будет выбрано последнее.
-w1	Указывает на то, что в случае возможного восстановления после ошибки в таблице стилей Saxon должен предпринимать восстановительные действия, определенные в спецификации XSLT, и выдавать предупреждение. Если, к примеру, один узел соответствуют двум шаблонным правилам с одинаковым преимуществом и приоритетом, то будет выбрано последнее, но перед этим будет показано сообщение о неоднозначности правил.
-w2	Указывает, что в случае ошибки в таблице стилей Saxon должен сообщить о ней и завершить выполнение, даже в случае, если восстановление после ошибки возможно.
-x classname	Определяет XML-анализатор, применяемый к исходному документу и любому другому документу, загруженному при помощи функции document(). Имя класса classname должно быть именем анализатора, реализующего интерфейс SAX1 org.xml.sax.Parser или SAX2 org.xml.sax.XMLReader.
-y classname	Определяет XML-анализатор, применяемый к таблице стилей и к любому другому модулю таблиц стилей, загруженному при помощи элементов <xsl:include> или <xsl:import>. Имя класса classname должно быть именем анализатора, реализующего интерфейс SAX1 org.xml.sax.Parser или SAX2 org.xml.sax.XMLReader.

(Зачем может понадобиться использование различных анализаторов для исходного документа и таблицы стилей? Одной из причин является то, что исходный документ не всегда может быть в формате XML; взгляните на пример GEDCOM в главе 10. Другой причиной может быть то, что вы захотите использовать анализатор с проверкой действительности для исходного документа и без проверки для таблицы стилей.)

Вы можете указать значения для глобальных параметров, определенных в таблице стилей, с помощью обозначения параметр=значение, например:

```
saxon source.xml style.xsl param1=value1 param2=value2
```

или для полной версии Saxon:

```
java com.icl.saxon.StyleSheet source.xml style.xsl param1=value1 param2=value2
```

Если имена параметров принадлежат какому-либо пространству имен, то можно использовать синтаксис TrAX для расширенных имен, например, «{namespace-uri}local-name». Значения параметров рассматриваются как строки. Если в строке присутствуют пробелы, ее нужно заключить в кавычки, например «param1="John Brown"».

Использование Saxon в Java-приложениях

Saxon также может быть вызван из Java-приложения с помощью интерфейса прикладного программирования TrAX, описанного в приложении F. Это позволяет скомпилировать таблицу стилей в объект Templates, который может быть затем неоднократно использован (последовательно или в параллельных потоках) для обработки различных исходных документов с помощью одной и той же таблицы стилей. Это может значительно улучшить производительность веб-сервера. С продуктом поставляется приложение в форме Java-сервлета, которое может это продемонстрировать.

Saxon реализует весь пакет javax.xml.transform, включая подпакеты dom, sax и stream, как для входных, так и для выходных данных. В нем также реализована фабрика SAXTransformerFactory, которая позволяет осуществлять преобразование как один из этапов конвейера SAX.

В пакет saxon.jar включен файл, который приводит к тому, что Saxon выбирается в качестве XSLT-процессора по умолчанию для TrAX TransformerFactory. Это означает, что в том случае, когда Saxon является единственной реализацией TrAX, указанной в переменной окружения CLASSPATH, вызов метода TransformerFactory.newInstance() автоматически приведет к созданию экземпляра TrAX в реализации Saxon. Если же в путях к классам указано несколько реализаций, возможно, как Saxon, так и Xalan, то выбор реализации зависит от порядка следования записей в путях к классам. В такой ситуации безопаснее всего явно выбирать одну из реализаций. Этого можно добиться несколькими способами:

- Можно выбрать Saxon, установив системное свойство Java, javax.xml.transform.TransformerFactory в значение com.icl.saxon.TransformerFactoryImpl. Для установки системных свойств используется параметр -D в командной строке java, вызывающей приложение. Помните, что этот параметр указывается перед именем исполняемого класса:

```
java -Djavax.xml.transform.TransformerFactory=
      com.icl.saxon.TransformerFactoryImpl com.my-com.appl.Program
```

  Вся команда набирается в одной строке. На практике вы вряд ли захотите набирать это свойство каждый раз, поэтому создайте пакетный файл или сценарий при помощи своего текстового редактора и вызывайте его.

- Создать файл с именем jaxp.properties в каталоге $JAVA_HOME/lib (где $JAVA_HOME – это каталог, в котором установлена Java) и включить в этот файл строку в виде свойство=значение, где свойство равно javax.xml.

transform.TransformerFactory, а значение — это класс Saxon com.icl.saxon.TransformerFactoryImpl.

- Поместить вызов

```
System.setProperty ("javax.xml.transform.TransformerFactory",
                    "com.icl.saxon.TransformerFactoryImpl")
```

в свое приложение для запуска в момент выполнения. Это единственный способ, подходящий для работы с различными TrAX-процессорами из одного и того же приложения, возможно, в целях сравнения их результатов или оценки производительности.

На момент написания книги в Saxon не была реализована другая половина спецификации JAXP 1.1, пакет javax.xml.parsers, который используется для выбора анализаторов SAX и DOM. Эти интерфейсы также не используются для выбора загружаемого анализатора. Чтобы указать объект SAX2 XMLReader или SAX1 Parser, который будет применяться для синтаксического анализа исходных документов и модулей таблиц стилей (они могут использовать разные объекты), нужно вызвать метод setAttribute() класса TransformerFactory:

```
import com.icl.saxon.FeatureKeys;
TransformerFactory factory = TransformerFactory.newInstance();
factory.setAttribute(
     FeatureKeys.SOURCE_PARSER_CLASS,
     "org.apache.xerces.parsers.SAXParser");
factory.setAttribute(
     FeatureKeys.STYLE_PARSER_CLASS,
     " com.icl.saxon.aelfred.SAXDriver");
```

Если эти атрибуты не установлены, то Saxon будет использовать встроенную версию анализатора Ælfred.

Непосредственная установка параметров конфигурации для выбранного анализатора, например включение проверки действительности XML-данных или указание конкретного SAX EntityResolver, не предусмотрена. Если вам нужно это сделать, то лучше всего написать Java-класс, являющийся подклассом выбранного анализатора, и установить нужные параметры в коде инициализации данного подкласса. После этого можно выбрать этот подкласс в качестве анализатора для использования в Saxon.

Например, можно определить следующий класс:

```
import org.xml.sax.*;
import com.icl.saxon.aelfred.*;
public class CustomizedParser extends SAXDriver implements EntityResolver {
  public CustomizedParser () {
    setEntityResolver(this);
  }
  public InputSource resolveEntity(String publicId, String systemId)
      throws org.xml.sax.SAXException, java.io.IOException {
    if (systemId.equals("http://www.acme.com/standard.dtd")) {
```

```
        return new InputSource("file:///e:/dtds/standard.dtd");
      } else {
        return null;
      }
    }
  }
```

А затем выбрать его в качестве анализатора исходного файла:

```
factory.setAttribute(
     FeatureKeys.SOURCE_PARSER_CLASS,
     "CustomizedParser");
```

Единственное особое действие, которое выполняет этот анализатор, – это перенаправление ссылок на конкретное DTD к локальной копии. (Если метод `resolveEntity()` возвращает пустое значение, то это означает, что анализатор должен разрешать ссылку на сущность своим обычным способом.)

Ниже приводятся остальные свойства конфигурации Saxon, которые можно установить с помощью интерфейса `setAttribute()`. Приведенные имена являются константами, определенными в классе `com.icl.saxon.FeatureKeys`, а не строковыми значениями, передаваемыми во время выполнения.

Имя атрибута	Значение
`TIMING`	Логическое значение. Аналогично параметру `-t` в командной строке.
`TREE_MODEL`	Целое значение, равное либо `Builder.STANDARD_TREE`, либо `Builder.TINY_TREE`. Определяет используемое внутреннее представление дерева. (Несмотря на названия, по умолчанию используется `TINY_TREE`.)
`TRACE_LISTENER`	Экземпляр класса, реализующего интерфейс `com.icl.saxon.trace.TraceListener`.
`LINE_NUMBERING`	Логическое значение. Аналогично параметру `-l` в командной строке.
`RECOVERY_POLICY`	Целое значение. Принимает одно из нижеприведенных значений: `Controller.RECOVER_SILENTLY`, `Controller.RECOVER_WITH_WARNINGS`, `Controller.DO_NOT_RECOVER`. Аналогично параметрам `-w0`, `-w1` или `-w2` в командной строке.
`MESSAGE_EMITTER_CLASS`	Строка, содержащая имя класса, экземпляр которого используется для обработки вывода из `<xsl:message>`. Этот класс должен быть расширением `com.icl.saxon.output.Emitter`.
`SOURCE_PARSER_CLASS`	Строка, содержащая имя класса, используемого для синтаксического анализа исходных документов, включая документы, загруженные с помощью функции `document()`. Этот класс должен быть либо SAX `Parser`, либо SAX2 `XMLReader`.

Имя атрибута	Значение
STYLE_PARSER_CLASS	Строка, содержащая имя класса, используемого для синтаксического анализа таблиц стилей, включая те, которые были загружены с помощью элементов <xsl:include> или <xsl:import>. Этот класс должен быть либо SAX Parser, либо SAX2 XMLReader.

После этого можно настроить объект TransformerFactory с помощью стандартных методов setURIResolver() и setErrorListener().

Saxon и DOM

В Saxon имеется два внутренних представления дерева: стандартное и компактное, оба они реализуют один и тот же интерфейс Java. Этот интерфейс является расширением базового интерфейса DOM первого уровня, определенного консорциумом W3C. Однако эта реализация предусматривает доступ только для чтения. Все методы, которые попытаются обновить DOM, вызовут исключительные ситуации.

Причиной этого является то, что при обработке нет необходимости производить изменения в дереве на месте; всегда можно добавить узлы к дереву в порядке появления в документе. Этот факт позволяет оптимизировать внутреннюю структуру дерева для обработки при помощи XSLT и XPath. Например, не нужно резервировать место для элемента, не имеющего атрибутов или объявлений пространств имен, на тот случай, если они у него появятся. Имеется возможность поддерживать крайне важные функции сортировки узлов в порядке появления в документе, включив простой последовательный номер в каждый узел дерева, что было бы невозможно в случае более позднего подключения новых узлов. Кроме того, гораздо легче регулировать многопоточные преобразования, когда обновление строго ограничено указанным выше способом.

В то время как мы получаем выигрыш в производительности, связь между Saxon и DOM слегка усложняется.

- Можно передать данные DOM на вход преобразования при помощи класса TrAX DOMSource, но это не самый эффективный способ передачи входных данных, поскольку сначала Saxon создаст копию документа с помощью собственного внутреннего представления дерева. Если изначально данные содержались в файле XML, то более эффективно передавать их в виде объекта SAXSource. В этом отношении Saxon сильно отличается от MSXML3, в котором переданные данные DOM используются в качестве внутреннего представления дерева.
- Можно получить конечное дерево в форме DOM, но если требуется, чтобы это была полностью обновляемая модель DOM, то нужно самим указать объект Document. Если предоставить Saxon создание конечного объекта Document, то он будет использовать собственную необновляемую реализацию DOM.

- Когда Saxon вызывает внешние функции Java, которые ожидают получить в качестве аргументов узлы или наборы узлов DOM, он предоставляет ссылки на собственную необновляемую структуру дерева. Эта структура включает только ту информацию, которая имеет отношение к модели дерева XPath, к примеру, она не включает секций CDATA или узлы ссылок на сущности, также в ней не будет текстовых узлов с пробельными символами, которые удаляются из дерева. Возврат узлов DOM какой-либо функцией в настоящее время разрешен только в том случае, когда они представлены в ограниченной реализации DOM в Saxon.

Использование выражений XPath в Saxon

Применяя Java API Saxon, можно вычислять выражения XPath относительно внутреннего представления дерева исходного документа, независимо от какой-либо таблицы стилей. Это может быть полезно как вне преобразований XSLT, так и внутри Java-функций расширения.

Основным классом здесь является `com.icl.saxon.expr.Expression`. Он имеет статический метод `make()`, который используется для компиляции выражения XPath, записанного в виде строки. Он действует как фабричный метод классов, возвращая экземпляр класса `Expression`, соответствующий скомпилированному выражению. Для вычисления выражения можно использовать метод `evaluate()`. В случае выражения, возвращающего набор узлов, можно получить результат в форме перечня узлов, вызвав метод `enumerate()`.

`Expression.make()` принимает два аргумента; первый является выражением XPath, записанным в виде строки, а второй представляет собой объект `StaticContext`, который содержит информацию, обычно зависящую от положения выражения в таблице стилей. Сюда входит такая информация, как привязки префиксов пространств имен, переменных и функций расширения. Для простого выражения, в котором не используются пространства имен, переменные или функции, можно сделать следующий простой вызов:

```
Expression exp = Expression.make("//item[@code='c']", new StandaloneContext());
```

`StandaloneContext` — это реализация `StaticContext`, которая предусматривает минимальный набор информации о контексте.

Вызов для вычисления выражения также принимает дополнительный аргумент; на этот раз это объект `Context`, соответствующий контексту времени выполнения. Он предоставляет информацию о контекстном узле, позиции и размере, о текущих значения переменных и тому подобном. Объект Saxon `Context` реализует интерфейс `org.w3.xsl.XSLTContext`, определенный в Рабочем проекте спецификации XSLT 1.1. Объект `Context` можно получить из объекта TrAX `Transformer` с помощью следующего вызова:

```
Context context = ((Controller)transformer).newContext();
```

После этого, скорее всего, потребуется инициализировать контекст, например:

```
context.setContextNode(node);
```

```
context.setPosition(1);
context.setLast(1);
```

И, наконец, можно вычислить выражение:

```
Value val = exp.evaluate(context);
```

Значение `Value` может иметь тип `BooleanValue`, `StringValue`, `NumericValue` или `NodeSetValue`, в зависимости от результата выражения XPath. Если нужно привести результат к определенному типу, скажем `String`, то для этого существуют сокращенные методы:

```
String s = exp.evaluateAsString(context);
```

Подробное описание см. в документации javadoc API Saxon.

Расширяемость

В этом разделе описываются те возможности, которые Saxon предоставляет для написания пользовательских функций и элементов расширения, а также другие интерфейсы для расширения функциональности продукта. После этого мы взглянем на те функции и элементы расширения, которые поставляются с Saxon в уже готовом виде.

Saxon поддерживает ряд механизмов расширяемости, описанных ниже.

Написание функций расширения

Функции расширения могут быть написаны на языке Java с помощью интерфейса, определенного в Рабочем проекте спецификации XSLT 1.1, как это описано в главе 8. К моменту написания книги (версия 6.2) в Saxon были реализованы не все, но большинство требований спецификации XSLT 1.1. Saxon поддерживает только внешние функции Java; JavaScript и другие языки не поддерживаются. Существует, однако, механизм для написания функций расширения с помощью самого XSLT, как будет описано на стр. 854 (в описании элемента `<saxon:function>`).

Внешний класс Java может быть определен с помощью элемента `<xsl:script>`, описанного в главе 4. Saxon игнорирует любые элементы `<xsl:script>`, которые не имеют атрибута «`language="java"`». Так, если вы хотите использовать методы, определенные в классе `java.util.Date`, вы можете написать:

```
<xsl:script language="java" implements-prefix="Date" src="java:java.util.Date"/>
```

а затем вызвать метод или конструктор, например:

```
<xsl:variable name="today" select="Date:new()"/>
```

Существует также сокращенный способ определения привязки. Если префикс пространства имен `Date` связан с URI пространства имен `java:java.util.Date`, то неявно подразумевается вышеприведенный элемент `<xsl:script>`. То есть можно сделать следующий вызов:

```
<xsl:variable name="today" select="Date:new()" xmlns:Date="java:java.util.Date"/>
```

без дополнительного объявления внешнего класса.

В текущем релизе, Saxon 6.2, не реализован полный набор правил XSLT 1.1 для нахождения метода внутри выбранного класса. Метод ищется по совпадению имени и количества аргументов, но если совпадений больше одного, то сообщается об ошибке, а не предпринимается попытка подобрать тот метод, который наилучшим образом соответствует переданным аргументам.

Как определено в рабочем проекте XSLT 1.1, Saxon позволяет внешним методам Java иметь дополнительный первый аргумент: он может быть объявлен как класс `org.w3c.xsl.XSLTContext` или, для совместимости с предыдущими релизами, как класс `com.icl.saxon.Context`. Этот аргумент, если он присутствует, передается не вызывающим кодом XPath, а самим Saxon. Это тот же самый объект `Context`, который используется в Saxon для вычисления автономных выражений XPath, поэтому его можно применять внутри функции расширения для вычисления таких выражений. Объект `Context` позволяет методу получить доступ к контекстной информации, такой как текущий узел и текущий список узлов. Он также предоставляет доступ к реализации класса TrAX `Transformer` (именуемый в Saxon `com.icl.saxon.Controller`), который в свою очередь дает доступ к приемнику ошибок, процедурам трассировки, настройкам фабрики и куче других подобных вещей, какие только могут понадобиться опытному системному программисту.

Написание элементов расширения

Saxon предоставляет возможность определять элементы расширения XSLT. Эта возможность позволяет разработчику определять собственные типы инструкций для использования в таблице стилей.

Если для элементов расширения будет использоваться префикс пространства имен, то он должен быть объявлен в атрибуте `extension-element-prefixes` элемента `<xsl:stylesheet>` или в атрибуте `xsl:extension-element-prefixes` любого охватывающего конечного литерального элемента.

Реализация элементов расширения – гораздо более трудная задача по сравнению с реализацией простых функций расширения. Вам следует быть достаточно подкованным в системном программировании, чтобы браться за это, тем более что документация в этой области ограничена. К Saxon в качестве образца прилагается набор элементов расширения, которые позволяют таблице стилей записывать данные в реляционную базу данных. Документация рекомендует использовать этот набор как пример для написания собственных элементов расширения.

Размеры данной книги не позволяют привести здесь полное описание этого интерфейса, но вот его основная структура.

Во-первых, для каждого пространства имен элементов расширения должен быть класс фабрики, реализующий интерфейс `com.icl.saxon.style.ExtensionElementFactory`.

URI пространства имен для элементов расширения должен иметь следующий вид: `xmlns:xxx="http://icl.com/saxon/extensions/full.class.Name"`, где `full.class.Name` – имя соответствующего класса фабрики.

Для каждого элемента расширения, определенного в этом пространстве имен, должен быть класс, являющийся расширением `com.icl.saxon.style.StyleElement`. Поставить имена элементов расширения в соответствие с именами классов – это задача объекта `ExtensionElementFactory` данного пространства имен. Для этого имеется метод `getExtensionClass()`, который принимает локальное имя элемента в качестве параметра и возвращает соответствующий класс.

Класс `StyleElement` для элемента расширения может реализовать ряд методов, наиболее важные из которых приведены ниже:

Метод	Действие
`isInstruction()`	Всегда должен возвращать истину для элемента расширения.
`mayContainTemplateBody()`	Необходимо возвращать истину, если данный элемент может содержать дальнейшие инструкции. Лучше, чтобы этот метод возвращал истину, иначе невозможно использование элемента `<xsl:fallback>`.
`prepareAttributes()`	Вызывается во время компиляции, чтобы класс мог произвести локальную проверку действительности и осуществить синтаксический анализ информации, содержащейся в его атрибутах.
`validate()`	Также вызывается во время компиляции. Он вызывается после того, как выполнены локальные проверки на действительность для всех элементов таблицы стилей; его предназначение состоит в том, чтобы осуществлять глобальную проверку, например проверку того, что у данного элемента имеются требуемые родительские и/или дочерние элементы.
`process(Context c)`	Вызывается во время выполнения для исполнения инструкции. Позволяет коду получить преимущество от использования многочисленных сервисов поддержки либо унаследованных от суперкласса `StyleElement`, либо доступных через объект `Context`. Для многопоточной целостности важно, чтобы этот метод не вносил изменений в содержимое дерева таблицы стилей.

Написание фильтров

Saxon принимает входные данные как поток событий SAX2. Как правило, этот поток формируется SAX2-совместимым экземпляром XMLReader. (Если входные данные передаются в формате DOM, то Saxon просто перемещается по модели для генерации соответствующих событий SAX.) Вместо этого можно формировать такие события из приложения.

Точно так же Saxon формирует выходные данные как поток событий SAX2. В действительности он использует несколько иной внутренний интерфейс, называемый `Emitter`, но существует класс `ContentHandlerProxy`, который преобразует события `Emitter` в стандартные события SAX2 `ContentHandler`. Поэтому для вывода преобразования можно написать постпроцессор. Можно указать объект `ContentHandler` для получения выходных данных Saxon либо в элементе `<xsl:output>`, либо через TrAX API.

Существует множество способов для использования этой возможности:

- Вы можете написать приложение, которое действует как фильтр между XML-анализатором и таблицей стилей, выполняя такие действия, как нормализация значений данных и установка атрибутов по умолчанию. Интерфейс TrAX облегчает вставку преобразования XSLT в конвейер, состоящий из нескольких фильтров, некоторые из которых могут быть определены с помощью XSLT, а другие с помощью Java. Пример такого фильтра приведен в приложении F. Фильтр может также разбить входной документ на несколько документов для раздельной обработки, сокращая тем самым размер хранимого в памяти дерева. За информацией по написанию SAX-фильтров обращайтесь к книге издательства Wrox «XML для профессионалов» («Professional XML», ISBN 1-861003-11-0).
- Ваше приложение может также передавать данные из источника, не являющегося XML-документом. Пример такой передачи был показан в главе 10, где генеалогическое дерево семейства Кеннеди было построено из источника в формате GEDCOM, а не XML. Приложение может также извлекать данные из реляционной базы данных. В действительности ваше приложение претендует на то, чтобы быть XML-анализатором, поэтому таблица стилей воспринимает входящие данные как переданные из XML-анализатора, хотя на самом деле они поступают из другого источника.
- Можно передать результат преобразования на вход другого приложения, разработанного для принятия входных данных в формате SAX2; например, процессор Apache FOP, являющийся частичной реализацией спецификации Форматирующих объектов XSL (XSL Formatting Objects). Конечно, это работает только в тех случаях, когда таблица стилей формирует XML-элементы и атрибуты, удовлетворяющие этой спецификации.

В качестве альтернативы написанию выходных фильтров на Java Saxon позволяет обрабатывать выходные данные с помощью еще одной таблицы стилей XSL. Это можно сделать с помощью интерфейса TrAX или просто указав имя следующей таблицы стилей в атрибуте `saxon:next-in-chain` элемента `<xsl:output>`. Это может быть полезно, если вы, например, хотите пронумеровать элементы в конечном дереве: инструкция `<xsl:number>` всегда формирует номера, которые относятся к позиции узла в исходном дереве, что не очень удобно, если таблица стилей сортирует данные. Поэтому можно осуществить сортировку с помощью одной таблицы стилей, а затем применить к полученным данным другую таблицу стилей, чтобы добавить номера разделов и оглавление.

Реализация последовательностей сортировки и нумерации

Saxon позволяет реализовать последовательности сортировки, используемые элементом <xsl:sort>. Это делается с помощью атрибута lang элемента <xsl:sort>. Эта возможность предназначена, прежде всего, для обеспечения сортировки в зависимости от используемого языка, но фактически она может быть использована для реализации произвольной последовательности сортировки: если вы, к примеру, желаете отсортировать названия месяцев – январь, февраль, март и т. д. в порядке их следования, то этого можно достичь, написав последовательность сортировки для языка «x-months».

Точно так же можно определить последовательность нумерации для использования в <xsl:number>. Пусть, например, имеется последовательность объектов, которые вы хотите обозначить как «январь», «февраль», «март» и т. д. В этом случае можно реализовать особый класс нумерации и вызвать его, указав <xsl:number format="январь" lang="x-months"/>.

Встроенные в Saxon расширения

Помимо поддержки с помощью описанных выше механизмов пользовательских расширений, Saxon также содержит обширную коллекцию встроенных расширений, описанных в последующих разделах.

Функции расширения

Saxon поставляется с мощной библиотекой функций расширения (то есть функций, не определяемых стандартом XSLT), описанных в нижеприведенной таблице. Эти функции должны использоваться с префиксом, который указывает на URI пространства имен http://icl.com/saxon.

Некоторые из этих функций используют **хранимые выражения**, которые в действительности являются дополнительным типом данных XPath. Это понятие, аналогичное хранимым процедурам в SQL или функциям первого сорта[1] (first-class functions) в функциональных языках программирования, является мощным дополнением к языку. В стандартном XSLT не существует способа для построения выражений XPath из строки. Это затрудняет осуществление привычных для программистов SQL операций, таких как построение запросов на основании значений параметров, переданных из формы, или сортировка таблицы по столбцу, выбранному пользователем. Кроме того, это не позволяет воспринимать выражения XPath, хранящиеся в виде составной части текста исходного документа, возможно, являющегося реализацией подмножества спецификации XPointer для определения связей между документами. Понятие хранимых выражений Saxon восполняет этот

[1] Функции, которые так же, как и данные, могут быть переданы в качестве аргументов в другие функции. – *Примеч. перев.*

пробел: можно воспользоваться функцией расширения saxon:expression() для создания хранимого выражения из строки, а функцией saxon:eval() для вычисления хранимого выражения, или же можно объединить две этих операции, применив функцию saxon:evaluate(), которая будет рассмотрена нами несколько подробнее в следующем разделе.

Вот список функций расширения, поддерживаемых в Saxon в настоящее время.

Функция	Объяснение
after(ns1, ns2)	Данная функция создана на основе оператора AFTER, определенного в рабочем предложении W3C XQuery (*http://www.w3.org/XML/Group/xmlquery/xquery*). Она выбирает все те узлы из набора ns1, которые следуют за первым узлом в наборе ns2 в порядке появления в документе.
base-uri()	Возвращает базовый URI контекстного узла. Если атрибуту xml:base не присваивалось значение базового URI, отличного от системного идентификатора содержащей сущности, то результат функции совпадает с результатом функции system-id().

before(ns1, ns2)	Данная функция создана на основе оператора BEFORE, определенного в рабочем предложении W3C XQuery. Она выбирает все те узлы из набора ns1, которые идут перед последним узлом в наборе ns2, в порядке появления в документе. Если, например, контекстным узлом является элемент <h1>, то для того, чтобы выбрать все элементы <h2>, идущие перед следующим <h1>, нужно написать:
	`saxon:before(`
	`following-sibling::h2, following-sibling::h1[1])`

difference (ns1, ns2)	Возвращает набор узлов, который является разностью двух указанных наборов, то есть набор, содержащий все узлы, которые содержатся в ns1, но отсутствуют в ns2. Вызов функции «saxon:difference($a, $b)» эквивалентен стандартному выражению XPath «$a[count(.|$b) != count($b)]».

distinct(ns1)	Возвращает набор узлов, содержащий все узлы из набора ns1, имеющие различные строковые значения. Узлы с повторяющимися значениями отбрасываются. Например:
	`<xsl:variable name="города" select="saxon:distinct(//город)"/>`
	создает набор узлов, содержащий один элемент для каждого уникального названия <город> из исходного документа. Если встречаются элементы <город> с одинаковыми именами, то остается только один – первый в порядке появления в документе. Следующий шаг, как правило, состоит в том, чтобы обработать этот набор как группу:
	`<xsl:for-each select="$города">`
	`<h2>Название города: <xsl:value-of select="."/></h2>`
	`<xsl:for-each select="//город[.=current()]">`
	`. . .`

Функция	Объяснение
	Если (и только если) элементы из $a являются одноуровневыми, то результат функции saxon:distinct($a) совпадает с результатом стандартного выражения XPath «$a[not(. = preceding-sibling::*]».

distinct(ns1, expression)	Возвращает набор узлов, содержащий все узлы из набора ns1, которые имеют различные значения для хранимого выражения, указанного во втором аргументе. Если, к примеру, требуется сгруппировать все элементы <город>, в соответствии с первой буквой их названия (для построения алфавитного указателя), то нужно написать:
	`<xsl:variable name="города" select="saxon:distinct(//город, saxon:expression('substring(., 1, 1)'))"/>`
	В результате будет получен набор узлов, содержащий по одному городу для каждой начальной буквы. Для нахождения остальных нужно использовать следующий код:
	`<xsl:for-each select="$города"><xsl:variable name="первая-буква" select="substring(., 1, 1)"/><h2><xsl:value-of select="$первая-буква"/></h2><xsl:for-each select="//город[substring(., 1, 1)=$первая-буква]"><p><xsl:value-of select="."/></p></xsl:for-each></xsl:for-each>`

eval(expression)	Данная функция вычисляет значение хранимого выражения, указанного в аргументе. Хранимое выражение строится с помощью функции saxon:expression(). Контекст для вычисления (например, контекстный узел) – это контекст, в котором вызывается функция saxon:eval().
evaluate(string)	Данная функция позволяет во время выполнения сформировать выражение XPath в виде строки и сразу после этого вычислить его. Это полезно, когда запрос не известен заранее, но должен быть сформирован на основании информации, переданной пользователем через параметры или считанной из исходного документа. Например, в следующем коде формируется набор узлов, состоящий из всех элементов <книга>, удовлетворяющих предикату, переданному в таблицу стилей в виде параметра: `<xsl:param name="предикат"/><xsl:variable name="выбранныеКниги" select="saxon:evaluate(concat('//книга[', $предикат, ']'))"/>`
exists(ns1, expression)	Данная функция возвращает истину, если набор узлов ns1 содержит, по крайней мере, один узел, для которого хранимое выражение истинно. Особой необходимости в этой функции нет, поскольку вызов функции «saxon:exists($a, expr)» эквивалентен выражению XPath «boolean($a[expr])», однако она предоставлена для полноты.
expression (string)	Данная функция формирует хранимое выражение на основе выражения XPath, содержащегося в переданной строке. Хранимое выра-

Функция	Объяснение
	жение может быть вычислено позже при помощи функции saxon:eval() или передано в другую функцию, которая принимает в качестве аргумента хранимое выражение, например saxon:distinct() или saxon:sum(). Если в выражении XPath содержатся префиксы пространств имен или ссылки на переменные, то они распознаются в момент создания хранимого выражения, а не тогда, когда это выражение вычисляется. Однако контекстный узел, его позиция и размер контекста определяются в момент вычисления выражения.
for-all(ns1, expression)	Данная функция возвращает истину в том случае, когда каждый узел из набора ns1, будучи подставленным в хранимое выражение, дает истину. Особой необходимости в этой функции нет, поскольку вызов функции «saxon:for-all($a, expr)» эквивалентен выражению XPath «not($a[not(expr)])», однако она предоставлена для полноты.
get-user-data(string)	Возвращает значение, которое было сохранено ранее с помощью функции saxon:set-user-data(). Строка указывает на элемент пользовательских данных. Пользовательские данные связаны с конкретным узлом дерева, поэтому данные можно извлечь только в том случае, когда контекстный узел тот же, что и в момент сохранения. Зачастую имеет смысл хранить пользовательские данные в корне дерева. Пользовательские данные также связаны с конкретным объектом TrAX Transformer, то есть данные доступны только в пределах одного преобразования.
has-same-nodes (ns1, ns2)	Возвращает истину тогда и только тогда, когда наборы ns1 и ns2 представляют собой одинаковые множества узлов. Учите, что эта функция отличается от оператора «=», который проверяет, существует ли пара узлов с одинаковыми строковыми значениями. Вызов функции «saxon:has-same-nodes($a, $b)» эквивалентен следующему коду в стандартном XSLT: `«count($a)=count($b) and count($a)=count($a\|$b)».`

highest(ns1, expression?)	Данная функция находит максимальное численное значение среди узлов указанного набора узлов ns1 и возвращает набор, содержащий все узлы, которые имеют это максимальное значение (таких узлов может быть несколько). Рассматриваемое значение представляет собой строковое значение узлов, если функция вызвана с одним аргументом, или значение указанного хранимого выражения, если указано два аргумента. Например, для того чтобы найти элементы <продажа> с максимальным значением «@количество * @цена», используйте:
	`<xsl:variable name="лучшие-продажи"`
	`select="saxon:highest(//продажа,`
	`saxon:expression('@количество * @цена'))"/>`

if(condition, v1, v2)	Первый аргумент вычисляется как логическое выражение; если оно истинно, то функция возвращает значение v1, если ложно, то

Функция	Объяснение
	функция возвращает v2. Значение может быть любого типа. Заметьте, что сначала v1 и v2 вычисляются, а затем одно из значений отбрасывается, поэтому нельзя применять эту функцию для того, чтобы скрыть ошибку в вычислении одного из аргументов.

intersection (ns1, ns2)	Возвращает набор узлов, который является пересечением двух переданных наборов узлов; то есть в полученном наборе содержатся все узлы, которые принадлежат как ns1, так и ns2.
	Эта функция может быть полезна при объединении по ключу, например для того, чтобы перечислить всех программистов из Денвера, нужно написать:
	`<xsl:for-each select="saxon:intersection(`
	`key('специальность', 'программист'),`
	`key('местонахождение', 'Денвер'))">`
	Вызов функции «saxon:intersection($a, $b)» эквивалентен стандартному выражению XPath «$a[count(.|$b) = count($b)]».

is-null(object)	Данная функция проверяет, является ли внешний объект значением null. Любая функция расширения может возвращать в качестве результата Java-объект; единственное, что может сделать с ним таблица стилей – это передать его в другую функцию расширения в качестве аргумента. Так как принято проверять, определено ли значение возвращаемого объекта, то данная функция служит именно этой цели.

leading(ns1, expression)	Данная функция рассматривает все узлы из набора ns1 в порядке появления в документе и возвращает те из них, которые возвращают истину для указанного хранимого выражения, пока не встретится первый узел, возвращающий ложь для данного выражения.
	Например, предположим, что элемент <section> имеет следующую последовательность дочерних элементов:
	`<h1/><p/><p/><h1/><p/><p/><p/>`
	Затем, начиная с контекстного узла <h1>, можно обработать все следующие за ним элементы <p> (вплоть до следующего <h1>), написав:
	`<xsl:apply-templates`
	`select="saxon:leading(`
	`following-sibling::*,`
	`saxon:expression('self::p'))"/>`
	В качестве альтернативы можно использовать вышеописанную функцию saxon:before().

line-number()	Возвращает номер строки текущего узла в исходном документе внутри содержащей его сущности. У функции нет аргументов. Это полезно для формирования сообщений об ошибках, а также для отладки. Номер строки доступен, только если дерево было построено с параметром -l командной строки, указывающей анализатору сохранять информацию о номерах строк.

Функция	Объяснение

`lowest(ns1, expression?)`	Данная функция находит минимальное численное значение среди узлов указанного набора узлов `ns1` и возвращает набор, содержащий все узлы, которые имеют это минимальное значение (таких узлов может быть несколько). Рассматриваемое значение представляет собой строковое значение узлов, если функция вызвана с одним аргументом, или значение указанного хранимого выражения, если указано два аргумента. Например, для того чтобы найти элементы `<продажа>` с минимальным значением `«@количество * @цена»`, используйте:
	`<xsl:variable name="худшие-продажи"`
	`select="saxon:lowest(//продажа,`
	`saxon:expression('@количество * @цена'))"/>`

`max(ns1, expression?)`	Данная функция находит максимальное численное значение среди узлов указанного набора `ns1` и возвращает это значение. (В отличие от функции `saxon:highest()`, которая возвращает сами узлы.) Рассматриваемое значение представляет собой строковое значение узлов, если функция вызвана с одним аргументом, или значение указанного хранимого выражения, если указано два аргумента. Например, чтобы найти максимальное значение `«@количество * @цена»` среди всех элементов `<продажа>`, используйте:
	`<xsl:variable name="максимальные-продажи"`
	`select="saxon:max(//продажа,`
	`saxon:expression('@количество * @цена'))"/>`

`min(ns1, expression?)`	Данная функция находит минимальное численное значение среди узлов указанного набора `ns1` и возвращает это значение. (В отличие от функции `saxon:lowest()`, которая возвращает сами узлы.) Рассматриваемое значение представляет собой строковое значение узлов, если функция вызвана с одним аргументом, или значение указанного хранимого выражения, если указано два аргумента. Например, чтобы найти минимальное значение `«@количество * @цена»` среди всех элементов `<продажа>`, используйте:
	`<xsl:variable name="минимальные-продажи"`
	`select="saxon:min(//продажа,`
	`saxon:expression('@количество * @цена'))"/>`

`node-set (tree)`	Данная функция является наследием стандарта XSLT 1.0, который не позволяет преобразовывать дерево в набор узлов. Она выполняет это преобразование явным образом. Теперь нужда в этой функции отпала, поскольку Saxon (следуя рабочему проекту XSLT 1.1) автоматически осуществляет такое преобразование, когда оно требуется.
`path()`	Данная функция возвращает в виде строки выражение XPath, которое является абсолютным путем к контекстному узлу. (Заметьте, что при необходимости повторно найти этот узел, вы должны убедиться в том, что все требуемые префиксы пространств имен определены в пределах видимости.)

Функция	Объяснение

`range(n1, n2)`	Формирует новый набор узлов, которые имеют численные значения в диапазоне от `n1` до `n2`. Функция создана как аналог привычного цикла for. В следующем примере создаются пять ячеек таблицы, содержащие числа от 1 до 5:
	`<xsl:for-each select="saxon:range(1,5)">`
	`  <td><xsl:value-of select="."/></td>`
	`</xsl:for-each>`

`set-user-data(string, value)`	Сохраняет значение, которое позже может быть извлечено с помощью функции `saxon:get-user-data()`. Строка указывает на элемент пользовательских данных. Пользовательские данные связаны с конкретным узлом дерева, поэтому данные можно извлечь только в том случае, когда контекстный узел тот же, что и в момент сохранения. Во многих случаях имеет смысл хранить пользовательские данные в корне дерева. Пользовательские данные также связаны с конкретным объектом TrAX Transformer, то есть данные доступны только в пределах одного преобразования. Учтите, что функция `saxon:set-user-data()` обладает побочными эффектами; так, результат зависит от порядка, в котором исполнялись различные инструкции, что не всегда предсказуемо.
`system-id()`	Возвращает системный идентификатор сущности, содержащей текущий узел в исходном документе. Функция не имеет аргументов. Полезна для формирования сообщений об ошибках.
`tokenize(string, delimiter?)`	Данная функция формирует новый набор узлов, узлы которого образованы разбиением указанной строки на группы, отделенные друг от друга либо указанным разделителем, либо пробелами. Функция разработана для таких операций, как замещение символов новой строки в исходном документе элементами ` ` в конечном документе или поиска заданного слова в текстовом содержимом элемента.

Использование saxon:evaluate()

Многие из функций расширения, приведенные выше, полагаются на возможность динамически формировать выражение XPath как строку, а затем вычислять его – простейший пример функция `evaluate()`.

В стандарте XSLT 1.0 (и 1.1) выражения XPath, всегда статически прописываются в таблице стилей. Это может быть значительным ограничением, особенно если вы знакомы с использованием SQL внутри страниц ASP или JSP, где динамическое формирование запросов SQL является нормой.

Saxon позволяет осуществлять подобные операции с помощью функции расширения `saxon:evaluate()`. (Подобная возможность предоставляется теперь и некоторыми другими XSLT-процессорами; подробности см. в описаниях соответствующих продуктов.)

Возможность динамического формирования запросов XPath дает ряд преимуществ:

- Можно сформировать запрос (вида «автор="Кэй" and издательство="Wrox"»), из значений параметров таблицы стилей, переданных во время выполнения.
- Можно без труда изменить порядок сортировки, используемый элементом <xsl:sort> на основе параметров, полученных во время выполнения.
- Можно использовать выражения XPath в исходном документе, например для определения ссылок между документами, и создавать в таблице стилей код для перехода по этим ссылкам.
- Вы можете использовать выражения XPath для определения бизнес-правил в отдельном документе.

Следующий пример демонстрирует использование последней возможности.

Пример: Использование saxon:evaluate() для применения бизнес-правил

В этом примере мы представим пример телефонного центра, который формирует счета потребителям за сделанные ими звонки. Мы хотим подготовить счет за период, указав все звонки и вычислив полную стоимость к оплате.

Исходный файл

Список звонков содержится в файле звонки.xml:

```
<звонки>
<звонок дата="2001-01-15" время="08.15" продолжительность="17"/>
<звонок дата="2001-01-16" время="10.42" продолжительность="8"/>
<звонок дата="2001-01-18" время="17.42" продолжительность="5"/>
<звонок дата="2001-01-18" время="22.10" продолжительность="06"/>
<звонок дата="2001-01-24" время="12:19" продолжительность="41"/>
<звонок дата="2001-01-25" время="06.40" продолжительность="13"/>
<звонок дата="2001-01-27" время="11.15" продолжительность="26"/>
</звонки>
```

Мы хотим поместить бизнес-правила по расчету стоимости в отдельный документ. Конечно, можно было бы разместить эти правила в таблице стилей, но это не очень хорошая практика; смешивание бизнес-правил и правил представления данных в одном месте не обеспечивает разделения ответственности. Поэтому мы поместим соответствующую формулу в отдельный документ тариф.xml в виде выражения XPath. Оно вычисляет полную стоимость, используя различные тарифы для рабочих и нерабочих часов дня:

```
<тариф>
   sum(звонок[@время &gt;= 08.00 and @время &lt; 18.00]/@продолжительность) * 1.50 +
   sum(звонок[@время &lt; 08.00 or @время &gt;= 18.00]/@продолжительность) * 2.50
</тариф>
```

Таблица стилей

Большая часть таблицы стилей (`счет.xsl`) является традиционной и имеет дело с отображением информации. Когда же дело доходит до расчета полной стоимости, то таблица стилей считывает выражение XPath из файла `тариф.xml` и вычисляет его (при помощи `saxon:evaluate()`) в контексте исходного документа.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:saxon="http://icl.com/saxon"
                version="1.0">

<xsl:template match="/">
  <html>
    <head>
      <title>Счет для периода, заканчивающегося
           <xsl:value-of select="(//@дата)[last()]"/></title>
    </head>
    <body>
      <h1>Счет для периода, заканчивающегося
           <xsl:value-of select="(//@дата)[last()]"/></h1>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
<xsl:template match="звонки">
   <table>
      <tr>
        <th width="100">Дата</th>
        <th width="100">Время</th>
        <th width="100">Продолжительность</th>
      </tr>
      <xsl:apply-templates/>
   </table>
   <xsl:variable name="сумма"
                 select="saxon:evaluate(document('тариф.xml'))"/>
   <p>Сумма к оплате за период:
      <xsl:value-of select="format-number($сумма, '$###0.00')"/>
   </p>
</xsl:template>
<xsl:template match="звонок">
   <tr>
     <td><xsl:value-of select="@дата"/></td>
     <td><xsl:value-of select="@время"/></td>
     <td><xsl:value-of select="@продолжительность"/></td>
   </tr>
</xsl:template>
</xsl:stylesheet>
```

Результат

Результат применения таблицы стилей выглядит в броузере следующим образом:

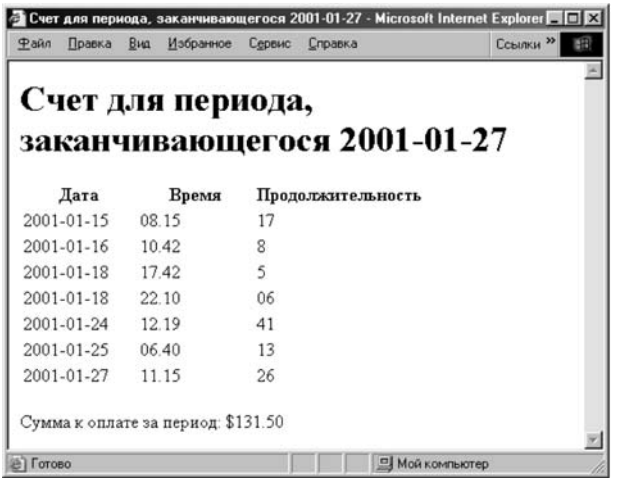


***Рис. С.1.** Результат таблицы, использующей* `saxon:evaluate()` *для применения бизнес-правил*

Элементы расширения Saxon

Поставляемые с Saxon элементы расширения доступны только в тех случаях, когда:

- Они используются с префиксом, указывающим на пространство имен `http://icl.com/saxon` **(за исключением элементов верхнего уровня).**
- Префикс этого пространства имен приведен в атрибуте `extension-element-prefixes` охватывающего элемента `<xsl:stylesheet>` или в атрибуте `xsl:extension-element-prefixes` охватывающего конечного литерального элемента.

Префикс `saxon` используется только в качестве традиционного префикса пространства имен: также как и в случае префикса `xsl`, вы можете использовать любой префикс, который соответствует верному URI пространства имен.

Ниже приведены элементы расширения Saxon:

Элемент	Описание
`<saxon:assign>`	Данный элемент используется для изменения значения локальной или глобальной переменной, которая была предварительно объявлена с помощью `<xsl:variable>` (или `<xsl:param>`). Например: `<saxon:assign name="n" select="$n+1"/>`

Элемент	Описание
	Использование подобных операций явного присваивания подрывает основной принцип XSLT как языка, не допускающего побочные эффекты. В главе 9 я назвал это «уловкой». Используйте эту возможность только тогда, когда все остальные средства бессильны. Обновляемая переменная должна иметь дополнительный атрибут «saxon:assignable="yes"», означающий, что ее значение можно изменить. Это предотвращает выполнение процессором оптимизации, которая основана на том, что значение не меняется. Несмотря на вышесказанное, существуют алгоритмы, такие как поиск пути через сеть, где использование <saxon:assign> может упростить кодирование и дать значительный выигрыш по производительности.

<saxon:doctype>	Этот элемент позволяет формировать внутри целевого XML-документа объявление DOCTYPE, содержащее внутреннее подмножество DTD. Инструкция <saxon:doctype> содержит представление DTD в синтаксисе XML. Например:
	`<saxon:doctype xsl:extension-element-prefixes="saxon">`
	`  <dtd:doctype name="книги"`
	`  xmlns:dtd="http://icl.com/saxon/dtd"`
	`  xsl:exclude-result-prefixes="dtd">`
	`  <dtd:element name="книги" content="(книга)*"/>`
	`  <dtd:element name="книга" content="EMPTY"/>`
	`  <dtd:attlist element="книга">`
	`  <dtd:attribute name="isbn" type="ID"  value="#REQUIRED"/>`
	`  <dtd:attribute name="название" type="CDATA"`
	`  value="#IMPLIED"/>`
	`  </dtd:attlist>`
	`  <dtd:entity name="реклама">'<i>Крутая</i> книга с`
	`  > 200 иллюстраций!'</dtd:entity>`
	`  <dtd:entity name="обложка" system="cover.gif"`
	`  notation="GIF"/>`
	`  <dtd:notation name="GIF" system="http://gif.org/"/>`
	`  </dtd:doctype>`
	`  </saxon:doctype>`
	Элементы из пространства имен «dtd» сами по себе не являются элементами расширения, они действуют просто как данные, рассматриваемые содержащей их инструкцией <saxon:doctype>.

<saxon:entity-ref>	Данный элемент полезен для формирования таких сущностей, как « », в выходном документе HTML. Например:
	`<saxon:entity-ref name="nbsp"/>`
	Вы можете достичь того же результата и стандартными средствами, написав:
	`<xsl:text disable-output-escaping="yes">&amp;nbsp;`
	`</xsl:text>`

Элемент	Описание
	(Но это не совсем хороший пример, поскольку, когда вы формируете на выходе неразрывные пробелы, например с помощью « », то по умолчанию Saxon представит его в выводе HTML как « ».)
<saxon:function> и <saxon:return>	С технической точки зрения элемент <saxon:function> не является элементом расширения, так как это элемент верхнего уровня таблицы стилей, а не инструкция. Этот элемент позволяет написать функцию расширения внутри таблицы стилей с помощью синтаксиса XSLT. Он вызывается так же, как и внешняя функция, написанная на Java или JavaScript. Элемент <saxon:return> применяется внутри <saxon:function> для выхода из функции и возвращения результата. В следующем разделе приводится пример использования элемента <saxon:function>.

<saxon:group> и <saxon:item>	Вместе эти элементы обеспечивают группировку элементов, имеющих одинаковые значения. Элемент <saxon:group> действует подобно <xsl:for-each>, но обладает дополнительным атрибутом group-by, который определяет ключ группировки.
	Элемент <saxon:group> должен располагаться где-нибудь внутри элемента <saxon:item>. Инструкции XSL вне элемента <saxon:item> исполняются один раз для каждой группы последовательных элементов с одинаковым значением ключа группировки. В то время как инструкции внутри <saxon:item> исполняются по одному разу для каждого элемента в выборке <saxon:group>.
	Например, для формирования списка книг, отсортированных и сгруппированных по автору, нужно написать:
	`<saxon:group select="книга" group-by="автор">`
	`  <xsl:sort select="автор"/>`
	`  Книги автора <xsl:value-of select="автор"/>:`
	`  <ul>`
	`  <saxon:item>`
	`  <li><xsl:value-of select="название"/></li>`
	`  </saxon:item>`
	`  </ul>`
	`</saxon:group>`

<saxon:preview>	Данная возможность полезна для обработки очень больших входных документов. Как правило, для обработки с помощью XSLT требуется, чтобы для всего исходного документа в памяти было построено дерево. В случае очень больших входных документов это может быть невозможно. Расширение <saxon:preview> позволяет обработать элемент как документ; как только элемент был считан, он обрабатывается с помощью таблицы стилей и затем удаляется из дерева вместе со своими потомками. Элемент <saxon:preview> имеет два атрибута: mode, который совпадает с режимом шаблонного правила, используемого для обработки элемента, и elements, представляющий собой список разде-

Элемент	Описание
	ленных пробелами имен элементов, которые должны обрабатываться таким образом. Обычно, если большой XML-документ состоит из множества элементов <item>, то указывается «elements="item"».
<saxon:while>	Этот элемент применяется для итераций, пока истинно определенное условие. Это условие задается логическим выражением в обязательном атрибуте test. Поскольку значение условия может измениться, только если что-то вызвало побочный эффект, то данный элемент используется, как правило, совместно с <saxon:assign> или с функциями расширения, подобными функции, считывающей следующую запись из файла. Например: <xsl:variable name="n" select="10"/> <saxon:while test="$n != 0"> <xsl:value-of select="$n"/> <saxon:assign name="n" select="$n - 1"/> </saxon:while>

Использование <saxon:function>

Элемент <saxon:function> добавляет к языку XSLT мощные возможности, поэтому на нем стоит остановиться более подробно.

XSLT позволяет создавать выражения, содержащие вызовы функций. Это могут быть как вызовы функций, определенные в спецификациях XPath и XSLT (например, not() или key()), так и вызовы пользовательских функций (называемых функциями расширения). Однако XSLT 1.0 не предоставляет какого-либо стандартного способа написания пользовательских функций. Спецификация XSLT 1.1, как можно было видеть в главе 8, определяет механизм для их написания на языках Java или JavaScript. Но она также включает сложные правила соответствия значений XPath значениям Java или JavaScript. Кроме того, XSLT-процессору очень трудно выполнить какую-либо оптимизацию, потому что он не знает, будет ли внешняя функция иметь побочные эффекты. Создавая элемент saxon:function, я рассчитывал на то, что многие простые и полезные функции можно будет написать на самом XSLT, и отпадет необходимость использования других языков для выполнения простого вычисления.

В следующем примере показано, как можно вызвать простую функцию из ключа сортировки.

Пример: Использование saxon:function для сортировки

Исходный файл

Здесь мы используем файл несортированные-звонки.xml, который очень похож на исходный файл из предыдущего примера, относившийся к звонкам, сделанным через телефонный центр. Но в этот раз мы предположим,

что звонки в исходном файле не расположены в каком-либо определенном порядке и что они не приведены в удобном формате ISO:

```
<звонки>
<звонок дата="27 Jan 2001" время="11.15" продолжительность="26"/>
<звонок дата="18 Jan 2001" время="17.42" продолжительность="5"/>
<звонок дата="18 Feb 2001" время="22.10" продолжительность="06"/>
<звонок дата="25 Jan 2001" время="06.40" продолжительность="13"/>
<звонок дата="15 Mar 2001" время="08.15" продолжительность="17"/>
<звонок дата="24 Jan 2001" время="12:19" продолжительность="41"/>
<звонок дата="16 Jan 2001" время="10.42" продолжительность="8"/>
</звонки>
```

Таблица стилей

Мы хотим перечислить эти звонки, отсортировав их по дате и времени. (Мы не рассматриваем здесь вычисление полной стоимости звонков, так как это не имеет отношения к нашему примеру.) Это достигается при помощи нижеприведенной таблицы стилей `отсортированный-счет.xsl`.

Сначала таблица стилей создает отсортированную копию исходного документа в глобальной переменной дерева, а затем выполняет остальную часть преобразования для этого отсортированного дерева, используя имеющуюся в XSLT 1.1 возможность обрабатывать временное дерево как набор узлов.

Интересной особенностью, однако, является вычисление ключа сортировки. Это делается с помощью функции `f:iso-date()`, которая преобразует значение даты в формат, который можно правильно отсортировать. Данная функция реализована в конце таблицы стилей с помощью элемента `<saxon:function>`.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:saxon="http://icl.com/saxon"
                xmlns:f="http://pricey-call-centres.com/namespace"
                extension-element-prefixes="saxon"
                exclude-result-prefixes="f" version="1.1">

<xsl:variable name="отсортированные-звонки">
  <xsl:for-each select="//звонок">
     <xsl:sort select="f:iso-date(@дата)"/>
     <xsl:sort select="@время"/>
     <xsl:copy-of select="."/>
  </xsl:for-each>
</xsl:variable>

<xsl:variable name="конец-периода"
   select="($отсортированные-звонки//@дата)[last()]"/>

<xsl:template match="/">
  <html>
```

```
    <head>
      <title>Счет за период, заканчивающийся
             <xsl:value-of select="$конец-периода"/></title>
    </head>
    <body>
      <h1>Счет за период, заканчивающийся
             <xsl:value-of select="$конец-периода"/></h1>
      <table>
         <tr>
            <th width="100">Дата</th>
            <th width="100">Время</th>
            <th width="100">Продолжительность</th>
         </tr>
         <xsl:apply-templates select="$отсортированные-звонки/звонок"/>
      </table>
    </body>
  </html>
</xsl:template/>

<xsl:template match="звонок">
   <tr>
     <td><xsl:value-of select="@дата"/></td>
     <td><xsl:value-of select="@время"/></td>
     <td><xsl:value-of select="@продолжительность"/></td>
   </tr>
</xsl:template>

<saxon:function name="f:iso-date">
<!--
  Преобразует дату из формата "21 Jan 1998" в формат "19980121"
-->
  <xsl:param name="дата"/>
  <xsl:variable name="месяцы" select="'JanFebMarAprMayJunJulAugSepOctNovDec'"/>
  <xsl:variable name="дд" select="substring($дата, 1, 2)"/>
  <xsl:variable name="мес" select="substring($дата, 4, 3)"/>
  <xsl:variable name="гггг" select="substring($дата, 8, 4)"/>
  <xsl:variable name="мм"  select="(string-length(
                           substring-before($месяцы, $мес)) div 3) +1"/>

  <saxon:return select="concat($гггг, format-number($мм, '00'), $дд)"/>
</saxon:function>

</xsl:stylesheet>
```

Результат

Результат преобразования выглядит в окне броузера следующим образом:

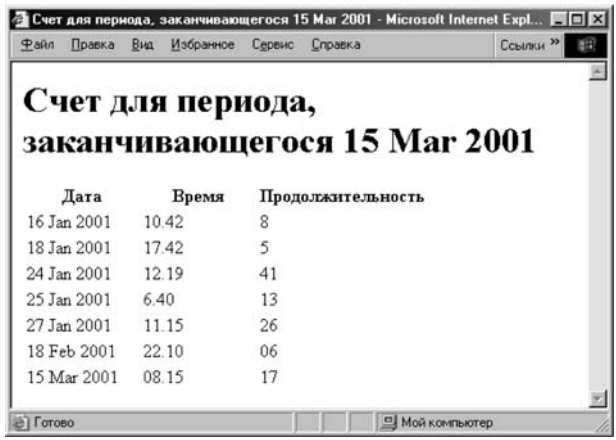


Рис. C.2. *Результат таблицы стилей, использующей* `saxon:function` *для сортировки*

Заметьте, что функция, определенная в `<saxon:function>`, должна не иметь побочных эффектов: в частности она не должна производить записи в дерево. Ее единственное действие – вернуть значение, которое может быть числом, логическим значением, строкой, набором узлов или временным деревом. Учтите также, что `<saxon:return>` является элементом расширения, поэтому пространство имен Saxon должно быть объявлено как пространство имен элемента расширения; если этого не будет сделано, то `<saxon:return>` будет рассматриваться как конечный литеральный элемент, который должен быть скопирован в конечное дерево, что приведет к ошибке, поскольку элементу `<saxon:function>` запрещено производить запись в конечное дерево.

Сериализация в Saxon

Saxon поддерживает стандартные методы вывода (xml, html и обычный текст), а также все атрибуты элементов `<xsl:output>` и `<xsl:document>`, определенные в рабочем проекте спецификации XSLT 1.1.

Атрибуты вывода могут быть указаны либо в элементах `<xsl:output>` и `<xsl:document>`, либо с помощью метода `setOutputProperties()`, имеющегося в TrAX API, как описано в приложении F.

Saxon также поддерживает некоторые дополнительные атрибуты или значения атрибутов для более точной настройки формата вывода. Все они принадлежат пространству имен Saxon (`http://icl.com/saxon`), поэтому любой другой процессор будет их игнорировать.

Вот эти атрибуты:

`method`	Этот атрибут поддерживает следующие дополнительные значения: `saxon:fop` Направляет вывод процессору Apache FOP для форматирующих объектов XSL. `saxon:xhtml` Форматирует выходные данные в соответствии с соглашениями XHTML, разработанными для традиционных броузеров. Например, ` ` выводится как ` ` (с пробелом). `saxon:`*`имя-класса`* Направляет вывод указанному классу Java, который должен реализовать либо интерфейс SAX2 `ContentHandler`, либо интерфейс Saxon `Emitter`.
`saxon:indent-spaces`	Целое значение, определяющее количество пробелов, применяемых для отступов, если установлен атрибут `«indent="yes"»`.
`saxon:character-representation`	Строка, определяющая желаемый способ вывода специальных символов. При атрибуте `«method="xml"»` воздействует только на символы, не принадлежащие выбранной кодировке. Значением может быть либо `decimal`, тогда на выходе формируются ссылки на символы в десятичном формате (например, `«&2345;»`), либо `hex`, тогда на выходе будут ссылки на символы в шестнадцатеричном формате (например, `«&x20a4;»`). При атрибуте `«method="html"»` значение может состоять из двух ключевых слов, разделенных точкой с запятой. Первое ключевое слово означает, как должны выглядеть символы кодировки, не принадлежащие ASCII, а второе – как должны выглядеть символы, не принадлежащие данной кодировке. Возможны четыре ключевых слова: `native`, `entity`, `decimal` и `hex`. Значения `decimal` и `hex` имеют тот же смысл, что и в случае XML-вывода. Кроме того, `native` означает, что на выход попадают сами символы в выбранной кодировке (что имеет смысл только для первого ключевого слова), а `entity` указывает системе использовать одно из предопределенных имен сущностей HTML, такое как `«&Aacute;»`, если это возможно. По умолчанию используется `«entity;decimal»`.
`saxon:omit-meta-tag`	Этот атрибут может принимать значения `yes` или `no`; по умолчанию используется `no`. Спецификацией XSLT требуется, чтобы при выводе в формате HTML процессор добавлял в заголовок документа элемент `<meta>`. Иногда это неудобно, если, например, вы преобразуете документ XHTML, который уже содержит этот элемент. Атрибут `saxon:omit-meta-tag` позволяет не выводить этот элемент.

saxon:next-in-chain	Это значение является URL другой таблицы стилей XSLT. Применение этого атрибута приводит к тому, что вывод данной таблицы стилей направляется на вход другой таблицы стилей. Конечное место назначения, используемое для данной таблицы стилей (например, файл, указанный в атрибуте href элемента <xsl:document>), передается другой таблице стилей и будет служить местом вывода результата ее преобразования.

Saxon поддерживает несколько кодировок, применяемых для выходного файла, а именно: ascii, us-ascii, utf-8, utf8, iso-8859-1, iso-8859-2, koi8-r, cp1250, windows-1250, cp1251, windows-1251.

Эти имена могут быть написаны как в верхнем, так и в нижнем регистре. Однако кодировки работают лишь в том случае, если они поддерживаются установленной у вас виртуальной машиной Java. Это происходит потому, что работа по формированию выходных данных в выбранной кодировке делится между Saxon и виртуальной машиной Java. Saxon отвечает за то, чтобы решить, какие символы Unicode могут быть представлены непосредственно, а какие нужно представлять как ссылки на символы. А виртуальная машина Java отвечает за перевод символов Unicode в нужную кодировку.

В Saxon имеется возможность добавить собственную кодировку, но при этом она должна поддерживаться виртуальной машиной Java. Подробности см. в документации по продукту.

Заключение

Как создателю продукта Saxon мне трудно дать объективную оценку его сильных и слабых сторон по сравнению с другими продуктами, и в любом случае это не является целью данной книги.

Однако к настоящему моменту Saxon завоевал одно из лидирующих положений среди доступных процессоров, и уж точно среди написанных на Java:

- Он обеспечивает строгое соответствие рекомендации по XSLT 1.0 и рабочему проекту XSLT 1.1, а также спецификации JAXP 1.1.
- Он предлагает исключительно богатую библиотеку расширений (которую вы можете с полным правом игнорировать в целях сохранения переносимости таблицы стилей).
- Его производительность по сравнению с другими Java-процессорами достаточно высока.

Можно отметить два основных потенциальных недостатка продукта: отсутствие какой-либо корпоративной поддержки или гарантий, что затрудняет привлечение серьезного интереса менеджеров проектов, а также тот факт, что он представляет собой только процессор XSLT, а не полный пакет разработчика XML, в который была бы включена поддержка DOM, схем XML, XLink, XPointer и всего прочего.

D

Xalan

Xalan – это процессор XSLT с открытыми исходными текстами, выпускаемый организацией Apache.

Различные продукты Xalan можно найти, следуя по соответствующим ссылкам на странице, по адресу *http://xml.apache.org/*.

> *Xalan ведет свою историю от более раннего продукта, называвшегося LotusXSL, созданного командой разработчиков под предводительством Скотта Боэга (Scott Boag) из дочерней компании IBM, Lotus, и переданного затем Apache Software Foundation. Том LotusXSL, который в настоящий момент доступен на веб-сайте IBM (http://www.alphaworks.ibm.com/), является просто оболочкой для продукта Xalan. Со времени передачи продукта организации Apache он был значительно усовершенствован с помощью разработчиков из ряда других организаций, хотя главным архитектором по-прежнему остался Скотт Боэг. Согласно информации с веб-сайта Apache, Xalan – это имя редкого музыкального инструмента. Если это так, то он и впрямь должен быть очень редким, так как о нем неизвестно даже Оксфордскому словарю английского языка и энциклопедии Британника.*

Xalan доступен как в версии для Java, так и для C++. Я собираюсь посвятить большую часть этого приложения описанию Java-версии, которая представляется более зрелой, более широко используемой и лучше документированной. Были также сообщения о том, что Java-версия быстрее, но я слышал также мнение, что это результат неправильного тестирования. В этой книге я, как правило, воздерживаюсь от сравнения производительности продуктов, отчасти потому, что результаты тестирований очень ненадежны, а отчасти потому, что они не долговременны – незадолго до того, как книга была отдана в печать, было объявлено о выходе Xalan для C++ версии 1.1, в которой были произведены существенные улучшения производительности.

Продукт Xalan-Java привлек широкое внимание пользователей. Недавно продукт был переработан для создания версии Xalan-Java 2, главным нов-

шеством которой стала поддержка API для преобразований TrAX, описанного в приложении F. Эта версия также привела продукт к соответствию спецификациям DOM2 и SAX2. Xalan-Java 2 и будет основным продуктом, описываемым в данном приложении. Версия 1 по-прежнему доступна и, возможно, на момент написания являлась более надежной и устоявшейся, тем не менее, если вы первый раз пользуетесь продуктом, то я бы рекомендовал начать сразу с версии 2.

Рассматриваемое программное обеспечение является свободным и распространяемым на условиях лицензии Apache Software License, которая фактически позволяет любой вид использования, распространения или модификации, но отвергает всякую ответственность. Формальной гарантии или поддержки нет, но существуют активно действующие списки рассылки и процедуры сообщения об ошибках. Вместе с продуктом поставляется его исходный код, включая подробные инструкции и сценарии для самостоятельной сборки продукта в случае изменения исходных файлов.

Xalan использует Apache XML-анализатор Xerces; Java-версия продукта работает с Java-версией Xerces Java, а версия для C++ работает с C++-версией анализатора Xerces. Xalan-Java 2 будет также работать с любым другим анализатором, в котором реализован интерфейс JAXP 1.1. Xalan поддерживает вывод в экземпляр SAX2 `ContentHandler` или в дерево модели DOM2, также как и стандартные методы вывода: XML, HTML или обычный текст.

Утверждается, что Xalan полностью поддерживает обязательные характеристики спецификаций XSLT 1.0 и XPath 1.0. Для XSLT пока не существует независимого тестирования на соответствие стандарту, поэтому, так же как и в случае других поставщиков, остается принимать такие утверждения на веру. Однако Xalan имеет хорошую репутацию, и его соответствие стандарту, по крайней мере, не меньше, чем у остальных продуктов.

Xalan–Java 2

Установка

Следуя ссылкам на странице по адресу *http://xml.apache.org/*, вы сможете загрузить подходящую версию Xalan. Программное обеспечение доступно либо в виде zip-файла (подходит для Windows), либо в формате `.tar.gz`, который лучше подходит для Unix-платформ, тем не менее это одно и то же программное обеспечение.

Вам нужно будет установить виртуальную машину Java Virtual Machine. Xalan явно поддерживает Java-машины от IBM и SUN версий 1.1.8, 1.2.2 или 1.3. В списке поддерживаемых виртуальных машин не значится Microsoft JVM. Я обнаружил, что она также работает с Xalan, но, возможно, существуют ограничения, которых я не обнаружил.

В продукт Xalan входят необходимые компоненты анализатора Xerces, поэтому нет необходимости отдельно устанавливать синтаксический анализатор XML.

Загрузите файл с расширением .zip или .tar.gz для Xalan и распакуйте их в подходящий каталог. Затем перейдите в каталог, содержащий файлы xalan.jar и xerces.jar, и добавьте эти файлы в переменную пути к классам Java. Это все!

В действительности мне представляется удобным копировать все файлы с расширением .jar в один каталог, например c:\apache. Одним из преимуществ такого подхода является то, что отпадает необходимость менять путь к классу каждый раз при установке новой версии: все, что нужно сделать – заменить старые файлы .jar на новые.

Помните, что в переменной пути к классам нужно указать сами файлы, а не содержащие их каталоги. Путь к классам – это список каталогов и файлов .jar, по которым будет осуществляться поиск, когда виртуальной машине Java нужно загрузить какой-либо класс. В Windows 98 это удобнее всего сделать, добавив в файл autoexec.bat строку, подобную следующей:

SET CLASSPATH=.;c:\apache\xerces.jar;c:\apache\xalan.jar

В Windows NT или Windows 2000 вы можете установить эту переменную окружения, выбрав в Панели управления (Control Panel) значок Система (System). Если же вы пользуетесь другой операционной системой, то вы, вероятно, знаете, что нужно сделать!

Запуск Xalan-Java из командной строки

XSLT-процессор Xalan можно запустить из командной строки или с помощью вызова его Java API. Кроме того, существуют оболочки, позволяющие запустить данный процессор из апплета или сервлета.

Типичный вызов из командной строки выглядит следующим образом:

```
java org.apache.xalan.xslt.Process -in a.xml -xsl b.xsl -out c.html
```

Команда java вызывает виртуальную машину Java. Если выдается сообщение, что класс org.apache.xalan.xslt.Process не может быть найден, значит, файл xalan.jar не был корректно записан в переменную пути к классам. Если же сообщение относится к классу org.apache.xerces.jaxp.SAXParserFactoryImpl, это означает, что файл xerces.jar не записан в переменную пути к классам.

Ниже приводится полный список параметров, которые могут быть указаны в командной строке. Эти параметры могут быть набраны как в верхнем, так и в нижнем регистре.

Параметр	Действие
-DIAG	Запрашивает вывод временной диагностики.
-EDUMP	Запрашивает вывод результата трассировки в случае ошибки.
-HTML	Указывает сериализатору выводить данные в формате HTML, в независимости от того, какой формат указан в элементе <xsl:output>.

Параметр	Действие
-IN url	Указывает URL исходного XML-документа. Если это относительный URL, то он воспринимается как имя файла по отношению к текущему каталогу.
-INDENT n	Указывает количество пробелов для выделения каждого уровня при выводе результата с отступами.
-MEDIA	(Недокументированный параметр: может поддерживаться не полностью.) Указывает атрибут media, используемый для выбора инструкции <?xml-stylesheet?> в том случае, когда их несколько. Это значение должно совпадать с псевдоатрибутом media выбираемой инструкции обработки <?xml-stylesheet?>.
-OUT filename	Указывает имя файла, в который будет записываться сериализованный вывод преобразования. Если этот аргумент опущен, то вывод направляется в System.out.

-PARAM name value	Устанавливает значение глобального параметра таблицы стилей. Если URI пространства имен данного параметра имеет определенное значение, то можно применить формат TrAX «{namespace-uri}local-name». Это значение будет рассматриваться как строка. Учтите, что в предыдущих релизах Xalan это значение трактовалось как выражение XPath, поэтому строку нужно было передавать во вложенных кавычках, например:
	`-PARAM город "'Лондон'"`
	Это уже неактуально для Xalan-Java 2; для того чтобы передать строку, нужно просто набрать:
	`-PARAM город Лондон`
	Если нужно передать несколько значений параметров, то ключевое слово -PARAM следует повторить, например:
	`-PARAM город Канзас -PARAM страна США`

-Q	Бесшумный режим: не показывать сообщения о ходе выполнения.
-QC	Отключить сообщения о неоднозначных шаблонных правилах (то есть о ситуациях, когда одному и тому же узлу соответствуют несколько шаблонных правил с одинаковым преимуществом и приоритетом).
-TEXT	Указывает сериализатору выводить данные в формате обычного текста, в независимости от того, какой формат указан в элементе <xsl:output>.
-TG	Указывает отслеживать каждое событие, которое записывает узел в конечное дерево.
-TS	Указывает отслеживать каждое событие, которое выбирает узел из исходного дерева.
-TT	Указывает отслеживать информацию, которая идентифицирует, какое шаблонное правило было применено.

Параметр	Действие
-TTC	Указывает отслеживать каждую инструкцию.
-V	Запрашивает вывод информации о версии Xalan.
-XML	Указывает сериализатору выводить данные в формате XML, в независимости от того, какой формат указан в элементе <xsl:output>.
-XSL url	Указывает URL таблицы стилей. Если это относительный URL, он воспринимается как имя файла по отношению к текущему каталогу. Если этот параметр опущен, то Xalan пытается определить таблицу из инструкции обработки <?xml-stylesheet?> исходного документа.

Например, для того чтобы запустить таблицу стилей с ходами шахматного коня, описанную в главе 10, команда должна выглядеть следующим образом (целиком набранная в одной строке):

```
java org.apache.xalan.xslt.Process -param start e5 -in dummy.xml
       -xsl tour10.xsl -out tour.html
```

Здесь предполагается, что текущим является каталог, содержащий файл таблицы стилей tour10.xsl, который доступен в загружаемом коде примеров для данной книги. Данная таблица стилей работает с любым исходным документом, поэтому единственным требованием для документа dummy.xml будет то, чтобы он существовал и являлся XML-документом. В файле tour10.xsl содержится версия этой таблицы стилей, которая работает с XSLT 1.0 и отличается от версии, описанной в главе 10, которая использует возможности рабочего проекта XSLT 1.1.

На самом деле tour10.xsl можно использовать как исходный файл для обработки, поэтому для удобства в ней содержится инструкция обработки, ссылающаяся на саму таблицу стилей. Это позволяет упростить команду для вызова процессора:

```
java org.apache.xalan.xslt.Process -param start e5 -in tour10.xsl -out tour.html
```

Запуск Xalan из Java-приложений

Xalan также имеет интерфейс Java API, позволяющий вызывать его из приложений, апплетов или сервлетов.

В Xalan-Java 2 используется TrAX API, определенный как часть интерфейса JAXP 1.1. TrAX API полностью описывается в приложении F. Этот интерфейс отличается от собственного API, использовавшегося в Xalan-Java 1. Xalan-Java 2 включает оболочку, обеспечивающую поддержку для большей части Xalan-Java 1 API, но в этом приложении я собираюсь сосредоточить основное внимание на описании более нового интерфейса TrAX.

В Xalan реализован весь javax.xml.transform, включая подпакеты dom, sax и stream как для входных, так и для выходных данных. В нем также реализо-

вана фабрика SAXTransformerFactory, позволяющая осуществлять преобразование как часть конвейера SAX.

Пакет xalan.jar включает версию класса верхнего уровня java.xml.transform.TransformerFactory и сконфигурирован таким образом, что Xalan становится XSLT-процессором, выбираемым системой по умолчанию. Это означает, что в случае, когда Xalan является единственной реализацией TrAX, присутствующей в переменной окружения CLASSPATH, вызов TransformerFactory.newInstance() автоматически приведет к созданию экземпляра Xalan-реализации TrAX. Если же в переменной пути к классам у вас имеется несколько реализаций TrAX, возможно, Saxon и Xalan, то загружается та, которая следует первой. Самым надежным способом в этом случае является явный выбор реализации. Этого можно достичь несколькими способами:

- Вы можете выбрать Xalan, установив системное свойство Java, javax.xml.transform.TransformerFactory равным org.apache.xalan.processor.TransformerFactoryImpl. Используйте параметр -D в строке команды java при вызове своего приложения. Заметьте, что этот параметр указывается перед именем исполняемого класса:

```
java -Djavax.xml.transform.TransformerFactory=
     org.apache.xalan.processor.TransformerFactoryImpl
     com.my-com.appl.Program
```

- Вся команда записывается в одной строке. На практике вы вряд ли захотите каждый раз печатать такую строку: создайте пакетный файл или сценарий оболочки с помощью своего текстового редактора и используйте его.

- Создайте в каталоге $JAVA_HOME/lib (где $JAVA_HOME — это директория установки Java) файл с именем jaxp.properties и включите в этот файл строку вида имя=значение, где имя это название свойства javax.xml.transform.TransformerFactory, а значение — это класс Xalan org.apache.xalan.processor.TransformerFactoryImpl.

- Поместите вызов System.setProperty ("javax.xml.transform.TransformerFactory", "org.apache.xalan.processor.TransformerFactoryImpl") в свое приложение, чтобы исполнить его во время выполнения. Это единственный метод, позволяющий работать с несколькими различными процессорами TrAX из одного приложения, возможно, в целях сравнения результатов или оценки производительности.

Xalan выбирает используемый анализатор SAX2 при помощи механизма, определенного в JAXP 1.1, а именно интерфейса javax.xml.parsers.ParserFactory. В JAXP 1.1 анализатором по умолчанию является Crimson (org.apache.crimson.jaxp.SAXParserFactoryImpl), новый облегченный Java-XML-анализатор от Apache, созданный на основе анализатора Sun Project X. Однако файл META-INF/services/javax.xml.parsers.SAXParserFactory, содержащийся в архиве xalan.jar, переопределяет эту настройку, указывая в качестве анализатора по умолчанию org.apache.xerces.jaxp.SAXParserFactoryImpl. Он-то и бу-

дет загружен при условии, что анализатор Xerces указан в переменной пути к классам. Подходящая версия `xerces.jar` включена в поставку Xalan.

Если вы желаете использовать другой анализатор, этого можно добиться, установив системное свойство `javax.xml.parsers.SAXParserFactory`. Альтернативным способом является явная загрузка анализатора в составе объекта `javax.xml.transform.sax.SAXSource`, который передается в качестве источника для преобразования; это особенно полезно при использовании различных анализаторов для исходного документа и для таблицы стилей, например, один с проверкой на действительность, а другой без нее.

Аналогично в случае необходимости создания документа DOM Xalan также применяет механизм JAXP 1.1: на этот раз используется интерфейс `javax.xml.parsers.DocumentBuilderFactory`. И снова по умолчанию в JAXP 1.1 предполагается новый анализатор `crimson`, но файл `META_INF` из `xalan.jar` назначает используемой по умолчанию хорошо зарекомендовавшую себя модель Xerces DOM, а именно класс `org.apache.xerces.jaxp.DocumentBuiderFactoryImpl`.

Интерфейс TrAX API разбивает преобразование на два этапа: подготовку объекта `Templates` на основе исходной таблицы стилей и использование этого объекта для управления преобразованием. Выгода такой организации состоит в том, что объект `Templates` может быть использован столько раз, сколько потребуется: например, на веб-серверах зачастую одна и та же таблица стилей неоднократно применяется к различным документам. Объект `Templates` может не только применяться несколько раз, но также вследствие своей многопоточности может быть использован для осуществления нескольких преобразований в параллельном режиме. Xalan в полной мере извлекает выгоду из такой архитектуры: объект `Templates` является, в действительности, скомпилированной версией таблицы стилей. Xalan также обеспечивает возможность сериализации объекта `Templates` (в смысле языка Java), что позволяет перемещать его между памятью и диском или реплицировать и перераспределять его между серверами в среде EJB (Enterprise JavaBeans).

Возможность Xalan-Java 1 записывать скомпилированную таблицу стилей из командной строки непосредственно на диск не поддерживается более в Xalan-Java 2, но сама концепция повторного использования остается такой же важной. Для улучшения производительности это является существенным: храните скомпилированную таблицу стилей под рукой, в кэше, а не перекомпилируйте каждый раз.

Xalan и DOM

Xalan может принимать в качестве входных данных любую совместимую с DOM2 реализацию DOM, он также может связывать конечное дерево преобразования с любым DOM2-совместимым документом. Если исходное дерево представлено в модели DOM, то Xalan использует эту модель в качестве внутреннего представления дерева. Если же исходные данные поступают в виде потока SAX или исходного файла XML, то Xalan использует собственное внутреннее представление дерева, называемое STree. Эта модель сменила DTM-дерево, использовавшееся в более ранних версиях. Представление

STree более эффективно, чем универсальная модель DOM, в значительной мере из-за того, что каждый узел содержит последовательное число, которое может быть использовано для быстрой сортировки узлов в порядке следования в документе. Трудность и медлительность процесса определения порядка следования в документе при использовании лишь стандартного интерфейса DOM2 общеизвестна. В модели STree реализованы все необходимые интерфейсы DOM2, но при применении многих методов обновления она вызывает генерацию исключительных ситуаций, поскольку предназначена для добавления узлов только в порядке следования в документе.

Характерной особенностью реализации STree является то, что она позволяет потоку анализатора, который формирует дерево, работать в параллельном режиме с преобразователем, который это дерево считывает. Если потоку преобразователя требуется доступ к еще не проанализированным узлам, то он ждет, пока поток анализатора не обработает эти узлы. В значительном количестве случаев это означает, что пользователь начинает видеть результаты преобразования еще до того, как исходный документ проанализирован целиком. Здесь проявляется стремление разработчиков Xalan, которое они держали в уме для будущего релиза, с помощью анализа исходного кода таблицы стилей обнаружить, когда доступ к конкретному узлу исходного дерева уже не нужен и можно удалить его из памяти; точно так же, сборщик мусора в Java удаляет объекты, на которые больше нет ссылок. Это закладывает основу для обработки больших документов XML, которые слишком велики, чтобы хранить их в памяти, при условии, что процесс преобразования последовательно применяется к документу. Если разработчики найдут способ достижения этой цели, то результат будет аналогичен механизму `<saxon:preview>`, имеющемуся в XSLT-процессоре Saxon (см. приложение C), но только здесь процесс будет полностью автоматизирован.

Применение выражений XPath в Xalan

Используя Java API Xalan, можно применять выражения XPath к представлению исходного документа в виде дерева независимо от таблицы стилей. Процессор XPath написан таким образом, что может работать с любой DOM-моделью: он не зависит от какого-либо конкретного представления дерева или какой-либо части Xalan (хотя в настоящее время он использует некоторые вспомогательные классы совместно с остальными частями продукта Xalan).

Такая ситуация дает два преимущества: она позволяет Java-приложениям, основанным на использовании DOM, перемещаться по дереву с помощью выражений XPath, а также дает возможность другим программным подсистемам, например XPointer, воспользоваться библиотекой XPath.

Большая часть функциональности XPath, которая вам может потребоваться, представлена набором статических методов класса `org.apache.xpath.XPathAPI`. Он обеспечивает оболочку для низкоуровневых операций из других классов пакета.

Эти методы описаны в нижеприведенной таблице:

Метод	Объяснение
`eval(Node contextNode, String xPath) → XObject`	Аргумент `xPath` содержит выражение XPath; `contextNode` указывает на контекстный узел. Как позиция, так и размер контекста равны 1. В качестве префиксов пространств имен могут быть использованы те, которые находятся в пределах видимости контекстного узла. Результат вычисления – объект `XObject`: он соответствует значению XPath, как правило, это `XBoolean`, `XNumber`, `XString`, `XNodeSet` или `XRTreeFrag`, в зависимости от типа данных.
`eval(Node contextNode, String xPath, Node nsNode) → XObject`	Данный метод аналогичен предыдущему, за исключением того, что в качестве префиксов пространства имен могут использоваться те, которые находятся в пределах видимости узла `nsNode`.
`eval(Node contextNode, String xPath, PrefixResolver p) → XObject`	То же самое, что и предыдущий метод, за исключением того, что для определения пространств имен по префиксам используется класс `PrefixResolver`, указанный пользователем.
`selectNodeIterator( Node contextNode, String xPath) → NodeIterator`	Данный метод может быть вызван для вычисления выражения XPath, результатом которого является набор узлов. Как и ранее, аргумент `contextNode` используется как контекстный узел для вычисления выражения и как отправная точка для определения префиксов пространств имен. Результат возвращается в виде объекта DOM2 `NodeIterator` (часть пакета `org.w3c.dom.traversal`). Это означает, что вычисление может быть вставлено в конвейер: не обязательно, чтобы сначала были найдены все узлы и помещены в память, система может считывать каждый узел по мере надобности.
`selectNodeIterator( Node contextNode, String xPath, Node nsNode) → NodeIterator`	Данный метод аналогичен предыдущему, за исключением того, что в качестве префиксов пространства имен могут использоваться те, которые находятся в пределах видимости узла `nsNode`.
`selectNodeList( Node contextNode, String xPath) → NodeList`	Данный метод аналогичен соответствующему методу `selectNodeIterator()`, за исключением того, что результат возвращается в виде объекта DOM `NodeList`. Обычно это означает, что до возврата результата все узлы были найдены.
`selectNodeList( Node contextNode, String xPath, Node nsNode) → NodeIterator`	Данный метод аналогичен предыдущему, за исключением того, что в качестве префиксов пространства имен могут использоваться те, которые находятся в пределах видимости узла `nsNode`.

Метод	Объяснение
`selectSingleNode( Node contextNode, String xPath) → Node`	Данный метод аналогичен соответствующему методу `selectNodeIterator()`, за исключением того, что результат возвращается в виде объекта DOM `Node`. Используйте этот метод только в том случае, когда вы знаете, что выражение XPath возвращает, по крайней мере, один узел (если ни одного узла не найдено, возвращается пустое значение).
`selectSingleNode( Node contextNode, String xPath, Node nsNode) → Node`	Данный метод аналогичен предыдущему, за исключением того, что в качестве префиксов пространства имен могут использоваться те, которые находятся в пределах видимости узла `nsNode`.

Класс `XObject` предоставляет методы, соответствующие большинству распространенных операций XPath над значениями, например, `equals()` для сравнения двух значений в соответствии с правилами XPath, `str()` для преобразования значения в строку и так далее.

С Xalan также поставляется демонстрационное приложение, которое использует возможность исполнять выражения XPath из командной строки. Оно обеспечивает удобный способ узнать про XPath больше и проверить сложные выражения XPath. Вы можете запустить его следующим образом:

```
java ApplyXPath XMLFile XPathExpression
```

где *XMLFile* – это исходный файл XML, а *XPathExpression* – выражение XPath, которое применяется к этому файлу. Если в выражении присутствуют пробелы, используйте кавычки. Например:

```
java ApplyXPath книги.xml "//автор[starts-with(., 'Б')]"
```

При использовании этой утилиты контекстным узлом всегда будет корневой узел. Применяемое выражение XPath должно возвращать набор узлов.

Функционирование Xalan-Java в качестве апплета

Xalan включает в себя класс `org.apache.xalan.client.XSLTProcessorApplet`, в котором сосредоточены все функциональные возможности, необходимые для выполнения преобразований в броузере, в качестве апплета. Продукт также содержит пример HTML-страницы, `samples/AppletXMLtoHTML/appletXMLtoHTML.html`, которая показывает, как встроить эти функциональные возможности в ваши собственные приложения. Ваш броузер при этом, конечно, должен быть настроен для запуска внутри него Java-апплетов. В демонстрационном приложении можно увидеть три параллельно расположенных фрейма, содержащих исходный файл XML, таблицу стилей и сформированную HTML-страницу.

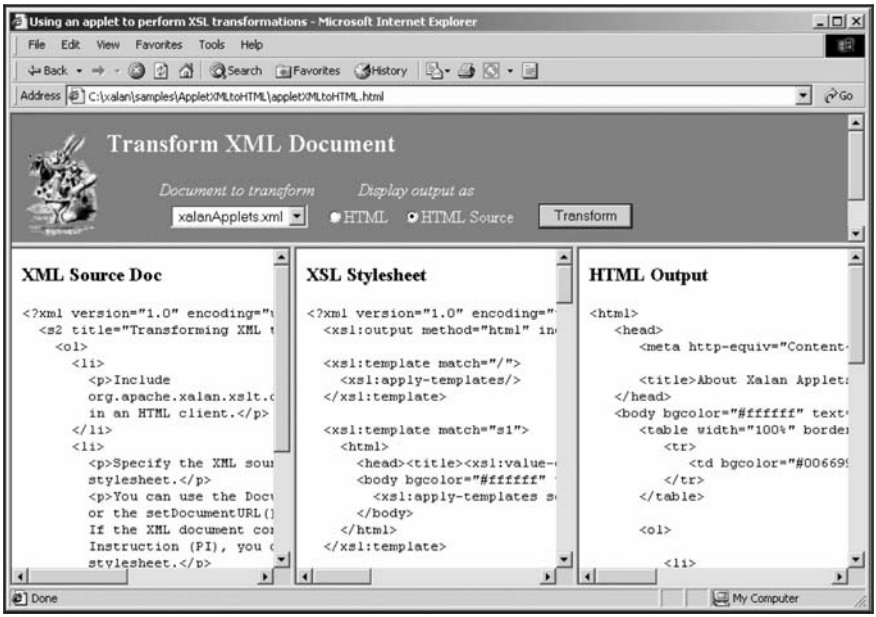


Рис. D.1. *Использование Xalan-Java в качестве апплета внутри броузера*

Вот основная часть кода HTML, которую вам нужно включить в свою страницу:

```
<applet
  name="xslControl"
  code="org.apache.xalan.client.XSLTProcessorApplet.class"
  archive="../../bin/xalan.jar,../../bin/xerces.jar"
  height="0"
  width"0">
  <param name="documentURL" value="default.xml"/>
  <param name="styleURL" value="stylesheet.xsl"/>
</applet>
```

Атрибут `archive` должен указывать на место расположения файлов `xalan.jar` и `xerces.jar` по отношению к расположению самой HTML-страницы: приведенный пример будет работать только в том случае, когда структура каталогов совпадает с использованной в данном приложении. Имена файлов исходного документа и таблицы стилей передаются апплету в качестве параметров. Они, конечно, не обязаны быть фиксированными, можно установить их динамически с помощью следующего кода JavaScript:

```
document.xslControl.setDocumentURL(xmlFileName)
document.xslControl.setStyleURL(xslFileName)
```

Таблицу стилей можно не указывать: если в исходном документе присутствует инструкция обработки `<?xml-stylesheet?>`, то она будет использована для данного документа по умолчанию.

Также можно установить значения параметров таблицы стилей:

```
document.xslControl.setStylesheetParam("start-record", "1")
document.xslControl.setStylesheetParam("last-record", "5")
```

Для выполнения преобразования нужно сделать следующий вызов:

```
var outputHTML = document.xslControl.getHtmlText();
```

Затем можно вставить полученный HTML-код в нужную страницу обычным способом: если страница содержит тег `<div id="resultArea">`, то можно написать:

```
resultArea.innerHTML = outputHTML;
```

Функционирование Xalan в режиме апплета является привлекательной возможностью, если все что нужно находится на вашей локальной машине, но в реальной ситуации нужно учитывать размер кода, который приходится загружать пользовательскому броузеру. Сам Xalan в настоящее время имеет размер около 700 Кбайт, а размер Xerces в целом составляет 1450 Кбайт. Вы можете существенно сэкономить, используя вместо последнего анализатор Crimson, занимающий 185 Кбайт. Но даже с такой экономией загрузка имеет практическую ценность, если все ваши пользователи подключены к высокоскоростной сети и, более того, если они пользуются данным программным обеспечением достаточно часто, чтобы оно оставалось в кэше броузера.

Функционирование Xalan–Java из сервлета

В поставку Xalan входит демонстрационный сервлет, позволяющий осуществлять на стороне сервера преобразования по соответствующим запросам броузера. Код для сервлета содержится в каталоге `samples/servlet`, а основной класс сервлета называется `ApplyXSLT.java`.

Сервлет можно использовать как в его исходном виде, так и адаптировав его для нужд приложения.

Фактически этот сервлет целиком написан при помощи интерфейса TrAX, поэтому его можно применять с любым TrAX-процессором. Единственная зависимость сервлета от Xalan состоит в том, что он использует обработчик ошибок, произведенный на основе обработчика ошибок Xalan, и устанавливает параметры конфигурации, которые не распознаются другими процессорами.

Во-первых, следует установить веб-сервер и процессор сервлетов, а также убедиться в том, что файлы `xalan.jar` и `xerces.jar` занесены в переменную пути к классам, которую использует процессор сервлетов. Удостоверьтесь, что вы используете неустаревшую версию Xerces или другого анализатора, поддерживающего интерфейс SAX2. Процессор сервлетов также должен знать путь к классам и файлам свойств, содержащимся в форме исходного кода в каталоге `samples/servlet` или, в откомпилированном виде, в файле `bin/xalanservlet.jar`. Помните, что процессор сервлетов не обязательно загружает классы из того же места, что и приложения Java, запускаемые из командной строки. Распространенной является ошибка, когда старые версии анализаторов остаются в каталогах, которые используются процессором сервлетов.

Можно скомпилировать поставляемый сервлет в каталог сервлетов своего веб-сервера. После этого его можно вызвать с помощью подобного URL:

```
http://my.company.com/servlet/ApplyXSLT?URL=/source.xml&xslURL=/style.xsl
```

При этом предполагается, что исходный документ XML `source.xml` и таблица стилей `style.xsl` находятся в домашнем каталоге веб-сайта.

Здесь возможны значительные вариации. Исходный файл может быть передан как часть URL, а не как параметр:

```
http://my.company.com/servlet/ApplyXSLT/source.xml?xslURL=/style.xsl
```

Если вы хорошо разбираетесь в конфигурации процессора сервлетов, то можете настроить его для автоматического нахождения файлов с данными, тогда URL можно сократить до:

```
http://my.company.com/servlet/source.xml?xslURL=/style.xsl
```

Если в исходном файле содержится инструкция обработки `<?xml-stylesheet?>`, то параметр `xslURL` может быть опущен (хотя здесь присутствует потенциальная потеря в производительности, поскольку сервер должен осуществлять поиск внутри файла, а не просто посылать его клиенту). Можно даже использовать файл `media.properties` для установки используемых таблиц стилей в зависимости от типа среды клиента, указанного в запросе HTTP, так что в зависимости от возможностей клиентского устройства применяется различная обработка.

В качестве альтернативы можно написать подкласс стандартного сервлета `ApplyXSLT` и определить, какую таблицу стилей использовать в зависимости от ваших знаний структуры приложения. Например, может быть одна таблица стилей для каждого каталога с исходными документами.

Более гибкое решение по управлению XSLT-преобразованием в среде Java-сервера можно найти в близком к Xalan продукте Cocoon, который кратко описан в приложении E. Cocoon представляет собой сложный для понимания продукт, но это следствие сложности решаемой им задачи, выходящей далеко за пределы простого преобразования одного исходного документа с помощью другой документа (таблицы стилей). Cocoon имеет дело со сложностями опубликования данных в Сети, когда имеется множество различных типов документов, представленных в различном виде и предназначенных для различной аудитории. До сегодняшнего момента в продукте Cocoon в качестве процессора преобразований используется Xalan-Java, хотя, возможно, в будущем, в связи с появлением TrAX API, пользователю будет предложен выбор процессора.

Расширяемость

Xalan позволяет определять как функции, так и элементы расширения.

К моменту написания книги в Xalan был реализован собственный механизм реализации расширений, а не тот, который предлагается рабочим проектом спецификации XSLT 1.1. Однако различия не так уж велики.

В модели расширяемости Xalan наборы элементов и функций расширения объединяются в **компоненты**, и каждый компонент идентифицируется при помощи URI пространства имен. Элементы и функции определяются элементом таблицы стилей верхнего уровня <xalan:component>. Но прежде чем рассмотреть элемент <xalan:component>, мы покажем упрощенный синтаксис для вызова методов Java.

Упрощенный синтаксис для вызова методов Java

Для функций расширения, написанных на языке Java, существует упрощенный синтаксис, в котором URI пространства имен функции расширения указывает процессору Xalan на имя Java-класса, содержащего соответствующий метод. Фактически существует три варианта такого упрощенного синтаксиса:

- Использование пространства имен для задания класса. Можно вызвать внешнюю функцию Java, используя следующий код:

```
<xsl:variable name="today" select="Date:new()" xmlns:Date="xalan://java.util.Date"/>
```

- Здесь URI пространства имен непосредственно указывает на требуемый класс. Префикс «xalan://» в идентификаторе пространства имен является условным, в действительности Xalan использует лишь ту часть, которая расположена после второго «/». (Это удобно, поскольку процессоры Saxon и xt используют подобные соглашения, что позволяет создавать код, который будет работать с несколькими процессорами.)
- Использование пространства имен для указания того, что используемым языком является Java. Следуя этой методике, можно включать полное имя класса в локальное имя вызова функции, используя префикс только для того, чтобы показать, что это класс Java. При таких обозначениях можно написать:

```
<xsl:variable name="today" select="java:java.util.Date.new()"
              xmlns:java="http://xml.apache.org/xslt/java"/>
```

- В этом случае URI пространства имен должен быть в точности таким, как показано выше. Преимущество данного подхода заключается в том, что нужно только одно объявление пространства имен для всех вызовов Java-функций расширения: как правило, это объявление дается в элементе <xsl:stylesheet>, а не помещается в вызов функции, как в нашем примере. Этот также означает, что нужно будет исключить из конечного документа только одно пространство имен, используя атрибут exclude-result-prefixes.
- Использование пространства имен для указания пакета Java и локального имени функции для указания класса и метода внутри этого пакета. В таком формате можно написать:

```
<xsl:variable name="today" select="util:Date.new()" xmlns:util="xalan://java.util"/>
```

Такая форма полезна, когда используется много классов из одного пакета Java. Это может быть хорошим выбором, когда пакет содержит ваше собственное приложение.

Элемент <xalan:component>

Элемент <xalan:component> является элементом верхнего уровня, определяющим пакет функций и элементов расширения.

Формат

```
<xalan:component>
    prefix    = префикс пространства имен
    functions = список имен без двоеточий
    elements  = список имен без двоеточий  >
    <xalan:script>...</xalan:script>
</xalan:component>
```

Расположение

Элемент <xalan:component> является элементом верхнего уровня, он должен являться непосредственным потомком элементов <xsl:stylesheet> или <xsl:transform>.

В таблице стилей может быть любое количество элементов <xalan:component>, но в них должны быть реализованы компоненты из различных пространств имен.

Поскольку это элемент верхнего уровня, а не инструкция, стоящая в теле шаблона, то <xalan:component> с технической точки зрения не является элементом расширения, а потому не требует, чтобы пространство имен для него объявлялось при помощи атрибута extension-element-prefixes в элементе <xsl:stylesheet>. Однако если это сделать, то вреда не будет.

Атрибуты

Имя	Значение	Объяснение
prefix *обязателен*	Префикс пространства имен, находящийся в пределах видимости из текущего места пространства стилей	Указывает на пространство имен, использованное для функций и элементов расширения, составляющих данный компонент
functions *не обязателен*	Разделенный пробелами список имен (без префикса пространства имен)	Указывает на функции расширения внутри данного компонента
elements *не обязателен*	Разделенный пробелами список имен (без префикса пространства имен)	Указывает на элементы расширения внутри данного компонента

Содержимое

Элемент <xalan:component> содержит в точности один элемент <xalan:script> и никаких других элементов.

Использование

Компонент определяет пакет связанных функций и элементов расширения: например, у вас может быть пакет для доступа к реляционным базам данных или пакет, отвечающий за работу с датами.

Причина, по которой необходимо наличие двух элементов, `<xalan:component>` и `<xalan:script>`, неочевидна, но, предположительно, это связано с расчетом на будущие улучшения.

Элемент <xalan:script>

Элемент `<xalan:script>` определяет реализацию набора функций и элементов расширения.

Формат

```
<xalan:script
    lang  = "javaclass" | "javascript" | другой язык
    src   = см. ниже
</xalan:script>
```

Расположение

Элемент `<xalan:script>` всегда располагается как непосредственный потомок элемента `<xalan:component>`.

Атрибуты

Имя	Значение	Объяснение
lang *обязателен*	«`javaclass`» (означает Java), «`javascript`» или имя другого языка, поддерживаемого стандартом Bean Scripting Framework[a]	Указывает язык, на котором реализован компонент

src
не обязателен
	Применяется только «`lang="javaclass"`».
	Так же как и в случае упрощенного вызова, имеется три возможных формата:
	`xalan://java.util.Date`
	указывает на конкретный класс
	`http://xml.apache.org/xslt/java`
	указывает, что в самом вызове функции будет указано полное имя класса
	`xalan://java.util`
	указывает пакет Java
		Указывает, где расположен класс или пакет Java

[a] Bean Scripting Framework – архитектура для внедрения сценариев в приложения и апплеты Java. Среди поддерживаемых языков сценариев: Javascript, VBScript, Perl, Tcl, Python и др. – *Примеч. перев.*

Содержимое

В случае функций расширения Java элемент <xalan:script> будет пуст, а в случае функций JavaScript или других языков в нем будет содержаться код реализации. В таких случаях удобно писать код внутри раздела CDATA, во избежание проблем со специальными символами.

Использование: Java

Для функций расширения Java атрибут lang всегда имеет значение «java-class». Атрибут src при этом всегда указывается и может иметь три приведенных выше формата. Вместе с локальным именем самой функции этого достаточно для идентификации как Java-класса, так и самого вызываемого метода.

Когда в атрибуте src используется формат указания класса (например xalan://java.util.Date), вызов функции имеет вид: «prefix:function(args)». Имя функции должно быть одним из перечисленных ниже:

- new в случае конструктора
- именем статического метода
- именем метода экземпляра, в котором нужный экземпляр указывается в виде первого аргумента
- именем метода экземпляра, в котором нужный экземпляр создается особо, с помощью конструктора, определенного по умолчанию

Когда атрибут src имеет значение http://xml.apache.org/xslt/java, имя функции должно быть одним из нижеприведенных, где full.class.Name представляет собой имя Java-класса, уточненное именем пакета:

- full.class.Name.new в случае конструктора
- full.class.Name.method, где method — это имя статического метода класса
- full.class.Name.method, где method — это имя метода экземпляра, в котором нужный экземпляр указывается при помощи первого аргумента.

Наконец, когда атрибут src имеет значение xalan://package.name, имя функции должно быть одним из следующих, где Class — это имя класса без указания имени пакета:

- Class.new в случае конструктора
- Class.method, где method — имя статического метода класса
- Class.method, где method — это имя метода экземпляра, в котором нужный экземпляр указывается при помощи первого аргумента

В Java-классе может быть несколько методов с одинаковыми именами. При выборе метода Xalan пытается найти наилучшее соответствие между типами данных XPath, переданных в вызове метода, и Java-классами, определенными для аргументов в сигнатуре метода. При поиске этого соответствия используются правила, близкие к тем, что определены в спецификации XSLT 1.1, но не идентичные им.

При сравнении аргументов XPath, указанных в вызовах функций, с объектами и значениями Java разрешены соответствия типов данных, приведенные в следующей таблице. Типы приводятся в порядке предпочтения (типы, указанные в квадратных скобках, имеют одинаковую предпочтительность).

Тип XPath	Типы Java
Набор узлов	org.w3c.dom.traversal.NodeIterator, org.w3c.dom.NodeList, org.w3c.dom.Node или его подклассы java.lang.String, java.lang.Object, char, [double, float, long, int, short, byte], boolean
Строковый	java.lang.String, java.lang.Object, char, [double, float, long, int, short, byte], boolean
Логический	boolean, java.lang.Boolean, java.lang.Object, java.lang.String
Числовой	double, java.lang.Double, float, long, int, short, char, byte, boolean, java.lang.String, java.lang.Object
Фрагмент конечного дерева	org.w3c.dom.traversal.NodeIterator, org.w3c.dom.DocumentFragment, org.w3c.dom.Node или его подклассы java.lang.String, java.lang.Object, char, [double, float, long, int, short, byte], boolean
Внешний объект	исходный тип или любой из его надклассов double, float, long, int, short, char, byte, java.lang.String

Если имеется несколько методов с одинаковыми именами и нужным количеством аргументов, Xalan присваивает каждому из них определенный приоритет на основе положения объявленных типов данных в вышеприведенном списке, а затем выбирает метод с наивысшим приоритетом. Если таких методов несколько, то выдается сообщение об ошибке.

Любая функция расширения Java может иметь дополнительный первый параметр типа org.apache.xalan.extensions.ExpressionContext. Он схож с объектом w3c.org.xsl.XSLTContext, определенным в рабочем проекте спецификации XSLT 1.1, хотя детали немного отличаются. Объект ExpressionContext передается автоматически и не нуждается в явном включении его в список аргументов вызываемой функции XPath.

Использование: JavaScript

Функции расширения Xalan могут быть реализованы как на Java или JavaScript, так и на другом языке, поддерживаемым стандартом Bean Scripting Framework, например на Perl или Python. Однако имеющаяся документация для других языков сценариев, кроме JavaScript, достаточно скупа, поэтому при их использовании вам придется продвигаться на ощупь.

Расширения JavaScript всегда встраиваются в саму таблицу стилей, внутрь элемента <xalan:script>, содержащегося в элементе <xalan:component>. В следующем примере показана реализация функции расширения user:datePlus() и элемента расширения <user:trace>.

Пример: Использование функций расширения JavaScript в Xalan

Исходный файл

Данная таблица стилей может быть применена к любому исходному документу.

Таблица стилей

Таблица стилей содержится в файле `xalan-script.xsl`.

В этой таблице стилей создается компонент Xalan, определяемый при помощи URI пространства имен с префиксом «`user`». Этот компонент определяет элемент расширения `<user:trace>` и функцию расширения `user:datePlus()`.

В шаблонном правиле для корневого узла показано, как вызвать эти функцию и элемент расширения.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                      version="1.0"
                      xmlns:xalan="http://xml.apache.org/xslt"
                      xmlns:user="http://any.user.com/xslt/extension1"
                      extension-element-prefixes="user">
<xalan:component prefix="user" elements="trace" functions="datePlus">
   <xalan:script lang="javascript">
      function trace (xslProcessorContext, extensionElement) {
         return "вызвана trace, сообщение=" +
            extensionElement.getAttribute("message");
      }
      function datePlus (days) {
         var d = new Date();
         d.setDate(d.getDate() + parseInt(days));
         return d.toLocaleString();
      }
   </xalan:script>
</xalan:component>
<xsl:template match="/">
    <html><body>
    <p><user:trace message="начинаем"/></p>
    <p>Через 5 дней будет число <xsl:value-of
         select="user:datePlus(5)"/></p>
    <p><user:trace message="закончили"/></p>
    </body></html>
</xsl:template>
</xsl:stylesheet>
```

Результат

```
<html>
<body>
<p>вызвана trace,сообщение=начинаем</p>
<p>Через 5 дней будет число February 25, 2001 11:36:25 AM GMT +00:00</p>
<p>вызвана trace, сообщение=закончили</p>
```

```
    </body>
    </html>
```

Для того чтобы сценарий функции расширения работал правильно, требуется наличие набора нужных программных продуктов. Для данного примера необходимы правильная установка и настройка следующих пяти программных продуктов:

- виртуальной машины Java
- XML-анализатора Xerces
- XSLT-процессора Xalan
- продукта Bean Scripting Framework
- интерпретатора Rhino JavaScript

Как говорилось выше, Xalan можно загрузить по адресу *http://xml.apache.org*; в его состав входит базовая версия Xerces. Bean Scripting Framework также входит в поставку Xalan. А вот Rhino Javascript нужно загрузить отдельно по адресу *http://www.mozilla.org/rhino/*. Xalan-Java 2 предназначен для работы с Rhino версии 1.5. Для запуска примера я использовал следующую команду:

```
java -cp XXX org.apache.xalan.xslt.Process -in dummy.xml -xsl xalan-script.xsl
```

где XXX – это путь к классу Java. Вся команда вводится в одной строке. Действительные имена файлов будут, конечно, зависеть от того, где установлено программное обеспечение.

```
D:\JavaLib\xerces\xerces-j_1_2_3\xerces.jar;
D:\JavaLib\xalan\xalan-j_2_0_0\xalan.jar;
D:\JavaLib\rhino\js.jar;
D:\JavaLib\xalan\xalan-j_2_0_0\bsf.jar
```

Использование: Реализация элементов расширения

Реализация элементов расширения аналогична реализации функций расширения. По сути, это работает следующим образом: если функция расширения реализует конкретную сигнатуру Java, то она может быть использована напрямую как элемент расширения.

Метод Java должен соответствовать локальному имени элемента расширения и иметь два аргумента:

- Первым аргументом является `org.apache.xalan.extensions.XSLProcessorContext`. Этот аргумент предоставляет полный доступ к исходному документу, таблице стилей, текущему узлу и списку узлов, а также к другой внутренней информации.
- Второй аргумент – `org.apache.xalan.templates.ElemExtensionCall`. Этот аргумент представляет собой узел дерева таблицы стилей, соответствующий данному элементу расширения. Он предоставляет служебные методы, которые могут быть вызваны из кода элемента расширения, например развертывание шаблонов значений атрибутов.

(Эта информация верна для Xalan-Java 2. В Xalan-Java 1 имена классов и определения слегка отличаются.)

Внутренняя структура дерева Xalan соответствует спецификации DOM, поэтому для нахождения связанных узлов дерева можно пользоваться стандартными DOM-интерфейсами. Если у функции имеется возвращаемое значение, то оно копируется в конечное дерево.

Расширения

В Xalan имеется полезная библиотека функций и элементов расширения, которая будет описана в следующих разделах. Перед именами этих функций и элементов используется условный префикс xalan:xxx, который объявлен следующим образом:

```
xmlns:xalan="http://xml.apache.org/xalan"
```

Встроенные функции расширения

Ниже приводятся имеющиеся функции расширения:

`difference (ns1, ns2)` Возвращает набор узлов, который является разностью двух указанных наборов узлов; то есть все узлы, которые содержатся в `ns1` и не содержатся в `ns2`.

`distinct(ns1)` Возвращает набор узлов из набора `ns1`, имеющих различные строковые значения. Например

```
<xsl:variable name="города" select="xalan:distinct(//город)"/>
```

создает набор узлов, содержащий по одному элементу для каждого элемента `<город>` из исходного документа, имеющего уникальное имя. Если имеется несколько элементов `<город>` с одинаковыми именами, то выбирается первый в порядке документа, хотя обычно это не имеет значения, так как следующим шагом, как правило, является обработка всех узлов как группы:

```
<xsl:for-each select="$города">
   <h2>Название города: <xsl:value-of select="."/></h2>
      <xsl:for-each select="//город[.=current()]">
         . . .
```

`evaluate (string)` Данная функция позволяет создать выражение XPath из строки во время выполнения и сразу же вычислить его. Это полезно, когда запрос неизвестен заранее, но должен быть построен на основе информации, полученной из параметров или считанной из исходного документа. Например, следующий код создает набор узлов, состоящий из всех элементов `<книга>`, удовлетворяющих предикату, переданному в таблицу стилей в качестве параметра:

```
<xsl:param name="предикат"/>
<xsl:variable name="выбранныеКниги" select=
      "xalan:evaluate(concat('//книга[', $предикат,  ']'))"/>
```

`hasSameNodes (ns1, ns2)`	Возвращает логическое значение, являющееся истиной, только в том случае, когда наборы `ns1` и `ns2` содержат совпадающие множества узлов. Учите, что эта функция отличается от оператора «=», который проверяет, существует ли пара узлов с одинаковыми строковыми значениями. В стандарте XSLT эквивалентная проверка запишется как: `<xsl:if test="count(ns1)=count(ns2) and count(ns1)=count(ns1|ns2)">`

`intersection (ns1,ns2)`	Возвращает набор узлов, который является пересечением двух переданных наборов узлов; то есть в результирующем наборе содержатся все узлы, которые принадлежат как `ns1`, так и `ns2`.
	Эта функция может быть полезна при соединении по ключу, например для того, чтобы перечислить всех программистов из Денвера, нужно написать:
	`<xsl:for-each select="xalan:intersection(`
	`key('специальность', 'программист'),`
	`key('местонахождение', 'Денвер'))">`

`nodeset(tree)`	Данная функция позволяет преобразовывать фрагмент конечного дерева в набор узлов. Это дает возможность обойти ограничения стандарта XSLT 1.0 (снятые в проекте XSLT 1.1) на использование фрагментов конечного дерева в таком контексте, как выражения пути.
`tokenize (string, delimiter?)`	Данная функция формирует новый набор узлов, узлы которого образованы разбиением указанной строки на группы, отделенные друг от друга либо указанным ограничителем, либо пробелами. Функция разработана для таких операций, как замещение символов новой строки в исходном документе элементами ` ` в конечном документе, или поиска заданного слова в текстовом содержимом элемента.

Функции расширения SQL

Xalan предоставляет набор функций расширений для соединения с базами данных через JDBC, выполнения запросов и обработки результатов запроса.

В своих таблицах стилей вы можете использовать три функции расширения. При этом подразумевается использование следующего объявления пространства имен: `xmlns:sql="org.apache.xalan.lib.sql.XConnection"`.

- `sql:new()` устанавливает соединение с базой данных. Возвращаемый объект соединения может быть присвоен переменной. Первым аргументом является имя требуемого драйвера JDBC; второй указывает на URL, идентифицирующий базу данных, в третьем и четвертом аргументах можно передать необязательные имя пользователя и пароль.
- `sql:query(connection, selectStatement)` выполняет запрос. Первый аргумент – объект соединения, возвращаемый функцией `sql:new()`, второй – оператор `SELECT` SQL в виде строки. Функция возвращает корневой узел нового документа, содержащего результаты запроса в некотором фиксированном формате, определенном в Xalan. Если вы хотите увидеть, что это за формат, то простейший способ сделать это – просто вывести его на экран, написав:

```
<xsl:copy-of select="sql:query($connection, 'SELECT * FROM EMP')"/>
```

- `sql:close(connection)` завершает соединение с базой данных.

Мы хотели показать вам пример использования данных возможностей, но, честно говоря, код оказался достаточно ненадежным. Он прекрасно работал с одними JDBC-драйверами, но давал сбой с другими (особенно на перемычке JDBC:ODBC). Несомненно, в будущих релизах данное средство будет работать более стабильно, но в данный момент издательство Wrox предпочло не публиковать примеры, работающие лишь иногда!

Несколько выходных файлов

Единственным элементом расширения, поставляемым с Xalan, является средство для формирования нескольких выходных файлов, элемент `<xalan:redirect>`. Он имеет функциональность, схожую с инструкцией `<xsl:document>`, введенной в рабочем проекте спецификации XSLT 1.1, описанной в главе 4, но отличается в деталях.

Этот элемент расширения позволяет преобразованию XSLT перенаправлять вывод в несколько назначений вывода. Нужно объявить пространство имен для префикса расширения (скажем `xmlns:redirect="org.apache.xalan.xslt.extensions.Redirect"`) как пространство имен расширения (`extension-element-prefixes="redirect"`).

Можно либо использовать элемент `<xalan:write>`, как в приведенном мною ниже примере: в этом случае файл будет открыт, в него будет произведена запись, после чего он сразу закроется; либо поместить свои вызовы на запись между элементами `<xalan:open>` и `<xalan:close>`: в таком случае файл будет открыт в течение нескольких сеансов записи, пока не встретится закрывающий вызов. Вызовы могут быть вложенными. При вызовах могут быть указаны атрибуты `file` и/или `select` для получения имени файла. Если указан атрибут `select`, то он вычисляется для формирования строки, указывающей на имя файла. Если результатом будет пустая строка, то по умолчанию используется атрибут `file`. Этот элемент может также иметь атрибут `mkdirs`, который, будучи установленным в истинное значение, приведет к созданию директорий, если они не существуют.

Лучше всего действие элемента иллюстрируется примером.

Пример: Создание нескольких выходных файлов

В этом примере на входе используется файл, содержащий стихотворение, а в результате преобразования каждая строфа выводится в отдельный файл. Более близким к жизни был бы пример с разбиением книги на главы, но я не хотел использовать большие файлы.

Исходный файл

Исходный документ содержится в `стих.xml`. Он начинается со следующего фрагмента:

```
<стихотворение>
   <автор>Руперт Брук</автор>
```

```
    <дата>1912</дата>
    <заголовок>Song</заголовок>
   <строфа>
      <строка>And suddenly the wind comes soft,</строка>
      <строка>And Spring is here again;</строка>
      <строка>And the hawthorn quickens with buds of green</строка>
      <строка>And my heart with buds of pain.</строка>
    </строфа>
    <строфа>
      <строка>My heart all Winter lay so numb,</строка>
      <строка>The earth so dead and frore,</строка>
. . .
```

Таблица стилей

Таблица стилей содержится в файле `xalan-split.xsl`.

Заметьте, что «xalan» определяется как префикс элементов расширения, поэтому элемент `<xalan:write>` рассматривается как инструкция. Его действие заключается в том, что весь вывод, производимый телом шаблона, разбивается на несколько выходных файлов. Фактически его действие схоже с результатом применения элемента `<xsl:variable>`, который создает дерево, за исключением того, что дерево вместо преобразования во фрагмент основного конечного дерева сериализуется в собственный выходной файл.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xalan="org.apache.xalan.xslt.extensions.Redirect"
                extension-element-prefixes="xalan" version="1.0">
<xsl:template match="poem">
   <стихотворение>
      <xsl:copy-of select="название"/>
      <xsl:copy-of select="автор"/>
      <xsl:copy-of select="дата"/>
      <xsl:apply-templates select="строфа"/>
   </стихотворение>
</xsl:template>
<xsl:template match="строфа">
   <xsl:variable name="файл" select="concat('строфа', position(), '.xml')"/>
   <строфа номер="{position()}" href="{$файл}"/>
   <xalan:write select="$файл">
        <xsl:copy-of select="."/>
   </xalan:write>
</xsl:template>
</xsl:stylesheet>
```

Результат

Основной выходной файл содержит структуру стихотворения (она разбита на строки для удобочитаемости):

```
<?xml version="1.0" encoding="utf-8" ?>
<стихотворение>
```

```
<название>Song</название>
<автор>Руперт Брук</автор>
<дата>1912</дата>
<строфа номер="1" href="строфа1.xml"/>
<строфа номер="2" href="строфа2.xml"/>
<строфа номер="3" href="строфа3.xml"/>
</стихотворение>
```

Три следующих выходных файла: `строфа1.xml`, `строфа2.xml` и `строфа3.xml` создаются в текущем каталоге. Вот содержимое файла `строфа1.xml`:

```
<?xml version="1.0" encoding="utf-8" ?>
<строфа>
<строка>And suddenly the wind comes soft,</строка>
<строка>And Spring is here again;</строка>
<строка>And the hawthorn quickens with buds of green</строка>
<строка>And my heart with buds of pain.</строка>
</строфа>
```

Сериализатор Xalan

Хотя сериализатор Xalan является интегрированной частью продукта, осуществляющего преобразования XSLT, в действительности он спроектирован как независимое программное обеспечение, которое может быть использовано и в других контекстах, например для сериализации документа, хранящегося в DOM. Сериализатор связан с остальными компонентами Xalan посредством интерфейса SAX2 `ContentHandler`.

Xalan поддерживает стандартные методы вывода (xml, html и text), а также все атрибуты элемента `<xsl:output>` согласно спецификации XSLT 1.0.

Один неприятный сюрприз, не являющийся несоответствием в строгом смысле, состоит в том, что атрибут `«indent="yes"»`, по всей видимости, не оказывает должного воздействия. Фактически это приводит к тому, что Xalan вводит новые строки в вывод, но не добавляет каких-либо отступов. Это может быть исправлено установкой свойства `«xalan:indent-amount»` в значение, отличное от нулевого, определенного по умолчанию.

Атрибуты вывода могут быть указаны как в элементе `<xsl:output>`, так и посредством метода `setOutputProperties()` из TrAX API, описанного в приложении F.

Xalan также поддерживает некоторые дополнительные атрибуты и значения атрибутов для более точного управления видом выходных данных. Все они принадлежат пространству имен Xalan (`http://xml.apache.org/xslt`), так что любой другой процессор будет их игнорировать.

Например, можно установить глубину отступов:

```
<xsl:output method="xml" indent="yes" xalan:indent-amount="5"
                    xmlns:xalan="http://xml.apache.org/xslt"/>
```

Вы можете изменить встроенные умолчания для всех свойств вывода, изменив файлы свойств в каталоге src/org/apache/xalan/templates и перекомпоновав файл xalan.jar с помощью предоставляемых средств. Файлы свойств, содержащие определенные по умолчанию свойства вывода, называются output_xxx.properties, где xxx – это html, text или xml.

Ниже приводятся определенные в Xalan свойства вывода:

Название свойства	Применяется к	Объяснение
xalan:indent-amount	XML, HTML	Количество пробелов в отступе. По умолчанию 0, что значит без отступов.
xalan:content-handler	XML, HTML, TEXT	Имя объекта SAX2 ContentHandler, предназначенного для обработки вывода. По умолчанию используется ContentHandler из модуля Xalan.
xalan:entities	XML, HTML	Имя файла ресурсов, в котором содержатся определения сущностей HTML, таких как «é». По умолчанию используется HTMLEntities.res.
xalan:use-url-escaping	HTML	Указывает, нужно ли экранировать URL, содержащиеся в HTML-атрибутах с использованием записи %HH. По умолчанию используется «yes»; установите значение «no», чтобы запретить такое поведение.

Можно использовать свойство xalan:entities для изменения представления символов с помощью именованных сущностей. Например, вы можете указать:

```
<xsl:output xalan:entities="myEntities.txt"/>
```

где файл myEntities.txt выглядит следующим образом:

```
quot 34
amp 38
lt 60
gt 62
nbsp 160
```

В результате как стандартные сущности, такие как «<», так и любой символ неразделяющего строку пробела (значения Unicode #160 или #xA0) будут представлены на выходе как ссылки на сущности « » (которые, в отличие от неразделяющих строку пробелов, видны на выводе).

В сущности Xalan поддерживает любую кодировку символов для выходного файла, которая поддерживается используемой виртуальной машиной Java. Для хорошо известных кодировок, таких как ISO 8859/1, Xalan преобразует ISO-название кодировки (iso-8859-1), которое применяется в XML, в назва-

ние этой кодировки в Java (ISO8859_1). Для каждой кодировки он выводит символы в исходном виде до некоторого порогового значения Unicode, а символы выше этого порогового значения – в виде цифровых ссылок на сущности, например «ࠈ». Пороговое значение отличается для различных кодировок: например, для US-ASCII это 127; для кодировок семейства ISO 8859 – 255; для EBCDIC-кодировок – 255; а для кодировок китайского, японского и корейского языков так же, как и для utf-8 и utf-16, это 65 535.

Xalan–C++

Как и было обещано в начале, большая часть этого приложения была посвящена Java-версии Xalan, однако существует еще и версия для C++, которая также заслуживает внимания, хотя она используется не так широко и имеет более скромную функциональность. Существуют результаты независимого тестирования, которые говорят о том, что эта версия работает медленнее Java-версии, однако, как это часто происходит при тестировании, другие результаты говорят об обратном: общеизвестно, что производительность очень чувствительна к условиям тестирования. В любом случае, кажется, что в последнем релизе (версия 1.1) наблюдается существенное улучшение производительности.

Можно загрузить продукт Xalan-C++ по адресу http://xml.apache.org, в формате, подходящем для вашей платформы, или можно скомпилировать продукт на своей машине. Если у вас Windows-платформа, то продукт устанавливается в виде ряда отдельных DLL и исполняемого файла, позволяющего работать из командной строки. Будет удобно включить каталог, содержащий этот исполняемый файл в переменную окружения PATH, определенную в вашем файле autoexec.bat.

Интерфейс командной строки

Исполняемая программа называется TestXSLT и имеет параметры и аргументы, аналогичные продукту Xalan-Java.

Для элементарного преобразования вызов выглядит следующим образом:

```
TestXSLT -IN source.xml -XSL style.xsl -OUT output.html
```

Полный список параметров и аргументов приведен в следующей таблице. Все параметры не чувствительны к регистру.

Параметр	Значение
-DOM	Формирует выходные данные как DOM-модель, которая затем сериализуется. Таким образом проверяется, что выходные данные являются корректным XML-документом.
-EER	Указывает, нужно ли раскрывать ссылки на сущности в исходном файле – по умолчанию они не раскрываются.

Параметр	Значение
-ESCAPE	Список символов, экранируемых при выводе (по умолчанию «<>&"'\r\n»).
-ESCAPECDATA	Удаляет квадратные скобки вокруг разделов CDATA и экранирует их содержимое.
-HTML	Устанавливает в качестве формата вывода HTML.
-IN	Указывает на URL исходного файла.
-INDENT n	Указывает число пробелов, составляющих отступ в выходных данных (по умолчанию ноль).
-NH	Подавляет вывод объявления XML.
-OUT	Имя файла, в который следует поместить результат преобразования.
-PARAM name expr	Устанавливает значение параметра преобразования. Имя `name` означает имя параметра, а `expr` – выражение XPath, дающее значение параметра. Заметьте, что если это обычная строка, то следует заключить ее в одну пару кавычек, чтобы показать, что это строковый литерал, и в другую, чтобы сообщить интерпретатору командной строки, что внутренние кавычки составляют часть выражения. Например: `-PARAM город "'Лондон'"`
-Q	Подавляет предупреждения и сообщения о ходе выполнения.
-QC	Подавляет предупреждения по поводу неоднозначных совпадений для шаблонных правил.
-STRIPCDATA	Удаляет квадратные скобки вокруг раздела CDATA, но не экранирует содержимое.
-TEXT	Устанавливает в качестве формата вывода TEXT.
-TG	Приводит к трассировке событий, записывающих узлы в конечное дерево.
-TS	Приводит к трассировке событий, выбирающих узлы из исходного дерева.
-TT	Приводит к трассировке шаблонов по мере их вызова.
-TTC	Приводит к трассировке инструкций внутри шаблонов по мере их исполнения.
-V	Выводит информацию о версии.
-VALIDATE	Означает, что исходный файл и таблица стилей должны проверяться на действительность анализатором XML.
-XML	Устанавливает в качестве формата вывода XML.
-XSL	Указывает на URL таблицы стилей.

Поддержка Unicode

В отличие от Java, язык C++ не имеет встроенной поддержки Unicode. Вместо этого Xalan-C++ предоставляет возможность интеграции международных компонентов для Unicode (International Components for Unicode, ICU), доступных на сайте IBM «IBM's DeveloperWorks». Внедрение этих компонентов требует повторной сборки продукта; но без них такие составляющие, как функция `format-number()` и элемент `<xsl:sort>`, не будут поддерживать Unicode.

Аналогично для обеспечения поддержки Unicode в синтаксическом анализаторе Xerces-C++ Xerces также должен быть перекомпонован с этими расширениями.

Расширяемость

Xalan-C++ позволяет писать функции расширения на C или C++. В настоящий момент продукт не позволяет создавать элементы расширения.

Каждая функция расширения должна быть написана как подкласс класса `Function`. Этот подкласс должен, как минимум, предоставлять метод `execute()`, используемый для вычисления функции, и метод `clone()`, который позволяет Xalan создавать новые экземпляры.

Связывание функций с процессором Xalan-C++ является более хитроумным, чем в случае Java. Невозможно объявить функцию в таблице стилей и все время обращаться к ней по имени, вместо этого нужно вызвать из приложения определенные методы при запуске процессора, чтобы объявить функции расширения. В частности, нужно создать объект `XSLTProcessorEnvSupportDefault` и вызывать его методы `installExternalFunctionGlobal()` или `installExtensionFunctionLocal()`. Эти методы принимают в качестве аргументов URI пространства имен и локальное имя функции, как оно встречается в таблице стилей, а также указатель на сам объект `Function`.

Заключение

Xalan – это зрелый, открытый продукт с высоким уровнем промышленной поддержки, осуществляемой организацией Apache. Хотя, полагаясь на распределенную команду разработчиков, далеко не всегда можно ожидать прогресс, данный продукт, по всей видимости, сочетает в себе улучшенную функциональность, высокий уровень новых технических решений, а также надежность и согласованность. В настоящий момент это не самый быстрый доступный процессор, но это не за горами: существует ряд прекрасных архитектурных возможностей, которые говорят о том, что разрыв скоро сузится.

Я бы рекомендовал использовать версию для Java, а не версию для C++, если только у вас нет серьезных оснований для отказа от использования Java. Но в любом случае – это решение, которое вам нужно принять самостоятельно.

E

Другие продукты

В четырех предыдущих приложениях мы подробно рассмотрели четыре процессора XSLT (MSXML3, Saxon, Oracle и Xalan). Мы выбрали именно их, поскольку, вероятно, 90% всех XSLT-преобразований осуществляется с их помощью. Однако существуют и другие продукты, которые заслуживают упоминания – одни из-за своих отличительных характеристик, делающих их достойной альтернативой указанным четырем продуктам, другие по причине их многообещающего потенциала.

Большинство продуктов, приведенных в этом приложении, являются процессорами XSLT, однако здесь также рассматриваются редакторы и среды разработки XSLT. Здесь не рассматриваются XML-инструменты, не имеющие непосредственного отношения к XSLT, просто потому, что их слишком много.

Это быстро меняющаяся область, а потому здесь можно представить лишь состояние на момент написания книги. То, что здесь приведены не все продукты, является неизбежной ситуацией, обусловленной либо тем, что я не знал о существовании этих продуктов, либо тем, что не имел достаточно времени для их достаточного исследования. За исчерпывающей информацией о продуктах XSLT, также как и XML-продуктах, я рекомендую обращаться по адресу *http://www.xmlsoftware.com/*.

В данном приложении рассматриваются следующие продукты, приведенные в алфавитном порядке:

- 4XSLT от FourThought (стр. 890)
- Cocoon от Apache (стр. 892)
- EZ/X от Activated Intelligence (стр. 892)
- iXSLT от Infoteria (стр. 893)

- jd:xslt Йогана Доблера (Johannes Dцbler) (стр. 894)
- Sablotron от Ginger Alliance (стр. 895)
- Stylus Studio от Excelon (стр. 896)
- TransforMiix от Mozilla (стр. 898)
- Unicorn от Unicorn Enterprises (стр. 899)
- XESALT от Inlogix (стр. 903)
- XML Spy от Altova (стр. 903)
- XSL Composer от Whitehill Technologies (стр. 906)
- XSLTC от Sun (стр. 907)
- xt Джеймса Кларка (James Clark) (стр. 908)

4XSLT

Это реализация XSLT с открытыми исходными текстами, написанная на языке Python. Производитель – компания FourThought из Булдера, штат Колорадо (Boulder, Colorado), а ведущим разработчиком является Уче Огбуджи (Uche Ogbuji). Информацию о компании можно найти по адресу *http://www.fourthought.com/*, а загрузить свободно распространяемые инструменты можно с сайта *http://4suite.org/index.epy*.

4XSLT является одним компонентом из набора программного обеспечения XML, которое выделяется из ряда подобных, учитывая небольшой размер фирмы-производителя. Помимо XSLT-процессора в состав продукта входят XML-анализатор с интерфейсами SAX и DOM, реализация стандартов XLink и XPointer, реализация RDF, а также объектная база данных. Основной отличительной чертой всех этих инструментов является то, что все они написаны на Python.

4XSLT является (насколько я могу судить) полной реализацией рекомендаций XSLT 1.0 и XPath 1.0. Средства XPath могут использоваться как автономно, так и совместно с реализацией DOM.

XSLT-процессор может быть запущен из командной строки или из приложения Python. Кроме того, существует возможность написания функций и элементов расширения на Python.

В продукте содержится несколько встроенных элементов расширения. В их числе:

- **ft:apply-templates**
  Разновидность `xsl:apply-templates`, которая позволяет динамически выбирать режим во время выполнения
- **ft:write-file**
  Создает несколько выходных файлов (похож на инструкцию XSLT 1.1 `<xsl:document>`)
- **ft:message-output**
  Определяет назначение вывода для инструкции `<xsl:message>`

4XSLT содержит также полезную библиотеку функций расширения, многие из которых совпадают с функциями расширения других процессоров, но некоторые не имеют аналогов:

- **ft:node-set()**
  Преобразует фрагмент конечного дерева в набор узлов
- **ft:match()**
  Определяет, соответствует ли строка регулярному выражению
- **ft:escape-url()**
  Экранирует специальные символы, содержащиеся в URL
- **ft:iso-time()**
  Возвращает текущие дату и время в формате ISO 8601
- **ft:evaluate()**
  Вычисляет выражение XPath, переданное в виде строки
- **ft:distinct()**
  Удаляет повторяющиеся узлы из набора
- **ft:split()**
  Разбивает строку на набор текстовых узлов, используя указанный ограничитель
- **ft:range()**
  Создает набор текстовых узлов, значения которых являются последовательными числами
- **ft:if()**
  Действует как условное выражение
- **ft:find()**
  Возвращает положение строки внутри содержащей строки

За несколько недель до отправки книги в печать Уче Огбуджи добивался внедрения независящей от поставщиков библиотеки функций расширения, подготавливаемой под руководством Джени Тенисона (Jeni Tennison); см. подробности на сайте *http://www.jenitennison.com/xslt/exslt/common/*. За этой инициативой стоит тот факт, что функции расширения различных поставщиков в значительной мере перекрывают друг друга, а потому имело бы смысл для функций расширения, реализованных несколькими поставщиками, использовать единое пространство имен, так чтобы можно было создавать переносимые таблицы стилей. Представляется вероятным, что продукт 4XSLT будет одним из лидеров в области реализации такой библиотеки, поэтому следите за его развитием.

В действительности я не пытался устанавливать продукт 4XSLT; вероятно, это достаточно просто, если вы знаете Python и знакомы с его соглашениями, однако в противном случае это задача не из легких. При этом Python представляется одним из тех языков, в которые влюбляются сразу после знакомства с ним, поэтому это может быть очень полезным опытом. Подробности см. на сайте *http://www.python.org*.

Cocoon

Cocoon не является XSLT-процессором, хотя он часто упоминается в такой роли. Это каркас для веб-публикаций, основанных на использовании XML, а потому он использует XSLT-процессор как ключевой компонент. Cocoon является одной из открытых технологий XML, которые созданы под покровительством организации Apache, продукт можно найти по адресу *http://xml.apache.org/*. А потому он обычно используется совместно с процессором Xalan XSLT, описанным в приложении D, и, конечно, Xalan распространяется как внутренняя часть Cocoon.

Cocoon предназначен для широкого спектра приложений, касающихся публикации данных. XSLT является технологией, сконцентрированной на одной задаче, в то время как Cocoon пытается охватить целую архитектуру для создания веб-публикаций. Особенно это проявляется в разделении таких задач, как создание информации (разработка XML), ее обработка и преобразование для конечного отображения.

Cocoon предоставляет средства для управления передачей и отображением документов, выбирая преобразования XSLT или другие этапы обработки в зависимости от таких факторов, как типы используемых пользователем устройства или броузера. По сути дела, это технология подключаемых расширений веб-сервера. Cocoon также обеспечивает доступ к базам данных SQL и каталогам LDAP.[1]

Основным средством управления действиями Cocoon служит страница XSP. Расширяемые серверные страницы (eXtensible Server Pages, XSP) являются XML-документами, определяющими проводимую сервером обработку данных. С архитектурной точки зрения они играют роль, схожую со страницами ASP или JSP (как можно предположить по имени), но по форме они менее процедурны. Тем не менее, в них могут быть определены логические процедурные шаги, воплощенные в Java-коде сценария страницы, или путем чтения и изменения свойств Enterprise Java Beans.

Это был лишь краткий обзор продукта Cocoon; это сложный продукт, и я не надеялся отдать ему должное в рамках этого небольшого обзора. Помните, если в ответ на вопрос об используемом XSLT-процессоре вам указали на Cocoon, то, скорее всего, используется Xalan, только этот факт не был замечен.

EZ/X

Этот продукт можно найти по адресу *http://www.activated.com/products/products.html*.

Это еще один Java-продукт, включающий XML-анализатор и XSLT-процессор. В прошлом издании книги я упоминал этот продукт как стоящий вни-

[1] Облегченный протокол доступа к каталогам (Lightweight Directory Access Protocol, LDAP). – *Примеч. перев.*

мания, но, к сожалению, похоже, что за прошедший год никакого прогресса не произошло. Несомненно, он по-прежнему удовлетворяет только рабочему проекту спецификации XSLT октября 1999 года, а не завершенной спецификации XSLT 1.0.

В документации делаются смелые утверждения о производительности, но эта информация представляется неактуальной на данный момент.

Подробные сведения о воплощенных возможностях отсутствуют, однако сообщения от некоторых пользователей позволяют предположить достаточно неполную их реализацию.

iXSLT

Продукт iXSLT распространяется японской компанией Infoteria (*http://www.infoteria.com*).

Данный продукт выглядит как серьезная попытка создать коммерчески качественный XSLT-процессор для Windows. В противоположность большинству продуктов с открытыми исходными текстами веб-сайт компании имеет вполне профессиональный вид, что подтверждается наличием сценариев установки и клиентским интерфейсом под Windows: существенное облегчение по сравнению с набором инструкций в командной строке и ручным редактированием переменных окружения. Можно загрузить бесплатную пробную версию продукта или приобрести коробочную версию, поставляемую с прекрасно упакованным диском CD-ROM и соглашением о технической поддержке. Я тестировал версию 2.0c, относящуюся к ноябрю 2000 года.

Процессор iXSLT реализован как DLL-библиотека Windows, позволяющая управлять преобразованием через программные интерфейсы COM или C++.

Как показано ниже, существует простой Windows-интерфейс пользователя. При особом желании можно работать с продуктом и из командной строки.

Также имеется мастер построения таблиц стилей, который пытается выполнить за вас трудную работу по созданию исходной таблицы стилей. Он предназначен, в первую очередь, для работы с табличными данными, которые вы можете получить, к примеру, из реляционной базы данных.

Вместе с пробной версией продукта также поставляется справочник в формате PDF. В нем есть все, что вам нужно знать для того, чтобы запускать продукт, но очень мало говорится о том, что в действительности можно писать в таблице стилей. Из этого может быть сделан вывод о том, что поддерживаемым языком является XSLT 1.0, не больше и не меньше. Однако некоторые из моих тестовых таблиц стилей были отклонены или неправильно обработаны, то есть, возможно, имеются некоторые недокументированные ограничения.

В файле `readme` дается некоторая информация о реализованных расширениях. В их состав входят элементы расширения для создания нескольких выходных файлов и вызова внешних команд, функции расширения для преоб-

разования фрагмента конечного дерева в набор узлов, а также для вычисления выражений XPath, указанных в виде строки.

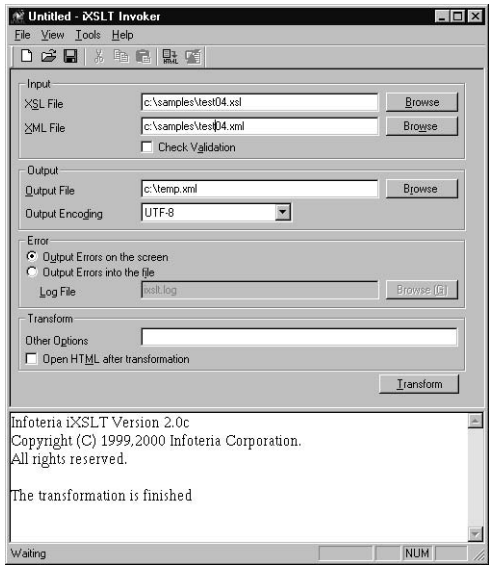


Рис. E.1. *Пользовательский интерфейс iXSLT*

jd:xslt

Данный продукт представляет собой новый Java-процессор XSLT, выпущенный 5 марта 2001 года Йоганом Доблером (Johannes Dцbler) из Мюнхена, Германия, перед самым выходом книги. Это означает, что мы не имели времени, достаточного для полной оценки. В отличие от многих разработчиков продуктов с открытыми исходными текстами, исповедующих философию «выпускай раньше, выпускай чаще» и тем самым наполняющих мир программного обеспечения ненадежными и недоделанными продуктами, Доблер, похоже, решил придержать код до тех пор, пока в нем не будет реализована полная функциональность. Без сомнения, это первый продукт, претендующий на полное соответствие рабочему проекту спецификации XSLT 1.1, что делает его очень полезным для проверки некоторых новых возможностей, описанных в этой книге.

Возможным недостатком такого подхода к выпуску продуктов является отсутствие обратной связи от пользователей, которые бы протестировали продукт в полевых условиях, а потому неудивительно, если начальная версия 1.0 будет содержать некоторое количество ошибок. Однако разработка выглядит достаточно надежной, и я надеюсь, возможные ошибки будут скоро исправлены. Первые впечатления о производительности продукта также достаточно обнадеживают, хотя еще слишком рано выносить какие-либо твердые суждения.

Похоже, что данный продукт является плодом рук одного человека, произведенным без корпоративной поддержки. Продукт можно загрузить по адресу *http://www.aztecrider.com/xslt/jd.zip*. Полный набор исходных кодов доступен на условиях лицензии Mozilla Public License.

jd:xslt разработан для работы с XML-анализатором Crimson от Apache. Продукт имеет собственное внутренне представление дерева, которое согласуется с моделью DOM. В настоящий момент в нем поддерживается собственный интерфейс Java API, а не TrAX API, описанный в приложении F. Функции и элементы расширения не описываются (но благодаря поддержке XSLT 1.1 основные пробелы все-таки заполнены). Пользовательские функции расширения могут реализовываться на Java или JavaScript.

Для сравнения в приведенной ниже таблице приведены размеры исходного кода четырех процессоров XSLT с открытыми исходными текстами, написанными на Java. Из этого совершенно не следует, что чем размер больше, тем продукт лучше, или наоборот; единственная причина, по которой я привожу эту таблицу, – показать, что, несмотря на статус новичка, jd:xslt определенно стоит рассматривать как серьезный продукт.

	Полные исходные коды	Исходные коды без комментариев
jd:xslt	37K	19K
Saxon	63K	31K
Xalan	118K	46K
xt	15K	13K

Sablotron

Sablotron – это XSLT-процессор, произведенный чешской компанией из Праги со странным названием Ginger Alliance[1], занимающейся интеграцией приложений на основе обмена XML-данными. Подробности по адресу *http://www.gingerall.com/*.

Основной продукт Ginger Alliance с не менее странным названием Charlie является представителем распределенного промежуточного программного обеспечения (middleware), позволяющего приложениям взаимодействовать между собой при помощи XML. Sablotron, написанный на C++ и доступный вместе с исходными кодами, появился как вспомогательный продукт: по словам компании, им нужен был быстрый, компактный и переносимый процессор XSLT для использования в составе программного комплекса Charlie, но они не могли найти подходящего среди имеющихся процессоров. Sablotron намеренно создавался как можно более компактным, для использования с карманными компьютерами, телевизионными приставками и т. п. Продукт уже успел заслужить хорошую репутацию благодаря своей производительности.

[1] Ginger Alliance – по всей видимости, «Союз Активистов». – *Примеч. перев.*

К моменту написания Sablotron являлся неполной реализацией XSLT 1.0. К числу недоступных возможностей относятся: наборы атрибутов, элементы `<xsl:import>`, `<xsl:key>`, `<xsl:number>` и `<xsl:fallback>`, оси `preceding`, `following` и `namespace` в XPath, а также функции `format-number()`, `id()` и `unparsed-entity-uri()`.

Sablotron доступен для Windows NT, Linux, Solaris и других разнообразных вариаций Unix. Ведущим разработчиком является Томас Кайзер (Tomas Kaiser). В продукте используется анализатор expat – XML-анализатор с открытыми исходными текстами, написанный Джеймсом Кларком.

Stylus Studio

Корпорация Excelon, расположенная в Берлингтоне (неподалеку от Бостона, США), продвигает на рынке хорошо известную объектную базу данных ObjectStore, а также ряд серверных XML-продуктов, предназначенных для использования в сфере электронной коммерции. Продукты для разработки XML распространяются под маркой Stylus Studio. Подробности можно найти по адресу *http://www.exceloncorp.com/products/excelon_stylus.html*.

Это был один из первых продуктов, которые вышли за пределы простых пакетных интерпретаторов XSLT и предоставили графическую среду разработки. Я был весьма разочарован первыми версиями продукта, но последняя (Stylus 3.0) выглядит очень впечатляюще. Графические средства работают как с XSLT-процессором MSXML3 от Microsoft, так и с собственным XSLT-процессором Excelon. Однако в последнем стандарт XSLT 1.0 воплощен неполностью; это говорит о том, что основной упор компания Excelon делала на среду разработки, а не на сам процессор XSLT.

Stylus Studio является коммерческим продуктом. На веб-сайте предлагается «загрузить пробную версию сегодня», однако я не смог найти кнопку загрузки. Вместо этого нужно подписаться на программу бета-тестирования Stylus 3.0, которая включает довольно сложный процесс регистрации, получения пароля по электронной почте, а затем уже загрузку и установку продукта. И лишь только по окончании этого процесса я узнал, что для работы продукта требуется Windows 2000 или NT – довольно странное ограничение для настольного средства (возможно, это справедливо только для бета-версии продукта – изучите свежую информацию на веб-сайте). Когда же я преодолел все эти препятствия, то оказалось, что объем загружаемых данных весьма внушителен (о времени загрузки не сообщалось), но час спустя 40 Мбайт программного обеспечения были успешно установлены на мой компьютер.

Stylus Studio предлагает обширную коллекцию инструментов для редактирования, просмотра и проверки XML-документов на действительность, а также для создания и отладки таблиц стилей. Здесь я собираюсь сосредоточиться только на инструментах XSLT. Рассматриваемый продукт не пытается снять с вас труд понимания языка XSLT, но он избавляет от утомительного контроля за правильным набором кода и его отладкой.

Существует несколько способов создания таблиц стилей XSLT, и я нашел, что все они нормально работают. Более того, различные способы создания таблиц стилей можно легко совместить.

Одним из подходов является использование Stylus в качестве редактора, поддерживающего синтаксис XML. Вероятно, это вам и нужно, когда определена основная структура таблицы стилей. Редактор предлагает обычный набор возможностей ускоренного набора, и после того, как вы привыкнете к ним, это значительно сократит количество нажатий клавиш, а также устранит большинство ошибок, совершаемых из-за невнимательности.

Для начального создания таблицы стилей может быть полезным представление Templates. На основе примера XML-документа Stylus создает модель его структуры в формате DTD или XML-схемы. После чего вы можете работать с этой структурой, определяя свои шаблонные правила для каждого типа элемента. Так как многие шаблонные правила очень просты, и Stylus предлагает по умолчанию некое стартовое правило, то этот процесс может быть очень быстрым.

Другой полезной методикой является импорт HTML-файла, который является типичным представлением формируемых вами файлов. Stylus вставит структуру этого HTML-файла в корневой шаблон таблицы стилей, который можно редактировать далее для указания тех мест, в которых нужно произвести выборку данных из исходного документа, или можно перенести части вставленного HTML-кода в другие шаблонные правила. Если логика требует написание сложных выражений XPath, то в Stylus имеется инструмент запросов XPath, в который можно вводить запросы XPath и просматривать их результаты; добившись правильного результата, можно скопировать нужное выражение в таблицу стилей.

Еще одним способом является создание начальной таблицы стилей с помощью инструмента установления соответствия Mapper, продемонстрированного ниже. Для его использования нужно загрузить экземпляры исходного и формируемого документов, а затем определить связь между ними, проводя линии от одной структуры к другой. Далее можно установить различные свойства соответствия, щелкнув правой клавишей мыши на линии. Я подозреваю, что такой подход эффективен только в тех случаях, когда исходный и формируемый документы тесно связаны, но когда это так, процесс первоначального создания определенно ускоряется. Я нашел, однако, что сформированная таблица стилей представляет собой не совсем то, что мне было нужно; в ней слишком часто использовались вложенные инструкции `<xsl:for-each>` вместо шаблонных правил.

Каким бы образом не была сформирована начальная таблица стилей, всегда можно вернуться к ручному редактированию для наведения последних штрихов. На любом этапе, нажав клавишу <F5>, можно применить таблицу стилей к исходному документу и увидеть результат либо непосредственно в текстовом виде, либо в виде XML-дерева, либо так, как он будет выглядеть в броузере HTML. Это еще не все, щелкнув на фрагменте сформированного документа, вы увидите выделенную часть таблицы стилей, которая показывает шаблонное правило, отвечающее за данный фрагмент вывода.

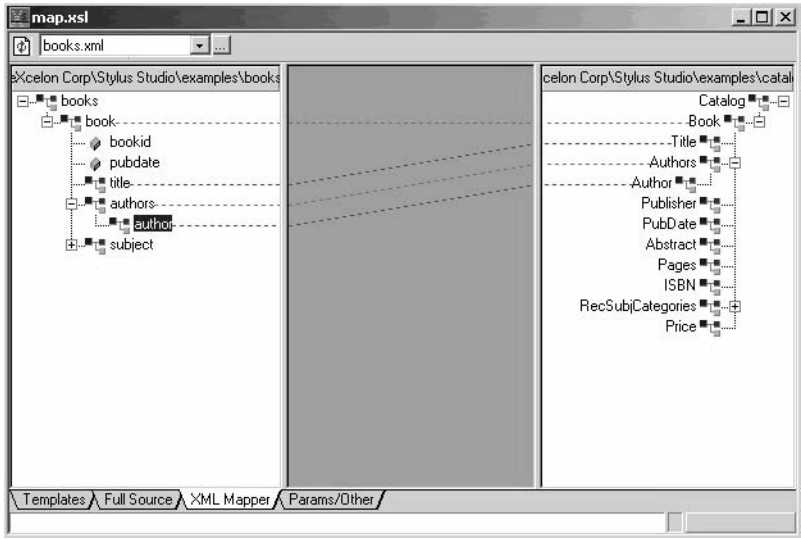


Рис. Е.2. *Создание таблицы стилей с помощью инструмента Mapper из Stylus Studio*

Я не дошел до того, чтобы в гневе использовать инструменты отладки таблицы стилей, но они выглядят достаточно полноценными. Вы можете устанавливать точки останова, отслеживать значения выбранных переменных и просматривать содержимое стека. Более того, все это можно осуществлять не только для инструкций таблицы стилей, но и для ваших собственных функций расширения на Java.

Самой слабой частью продукта Stylus Studio является сам XSLT-процессор. Несмотря на утверждения документации о соответствии стандарту XSLT 1.0, можно обнаружить, что это не так: наиболее существенным упущением является отсутствие поддержки пространств имен. Кроме того, при обработке документов любого размера (например, 250 Кбайт) процессор работает достаточно медленно.

И в заключение: как средство разработки Stylus Studio – лучший инструмент из виденных мною, но как процессор XSLT он оставляет желать лучшего.

TransforMiix

TransforMiix представляет собой открытый XSLT-процессор, разработанный под крышей организации Mozilla. Его предназначение – выполнять ту же функцию в броузере Mozilla, какую играет процессор MSXML3 для Internet Explorer. Подробности можно узнать по адресу *http://www.mozilla.org/projects/xslt/*. Ведущими разработчиками являются Кейт Вискоу (Keith Visco) и Петер ван дер Бекен (Peter van der Becken).

В настоящее время вход на веб-сайт TransforMiix создает ощущение, что вы залезли на стройку. Не входите на него без каски! Вы увидите, что сайт полон разговоров о невоплощенных возможностях, ограничениях, утечках памяти и множестве неисправностей. Вы будете приятно удивлены тем, что можете загрузить бинарную версию для Windows или Linux, а также тем, что существует форма, автоматически определяющая версию, необходимую для вашей платформы, однако вы быстро спуститесь на землю, узнав, что продукт предназначен для работы «с вашими последними версиями еженощных сборок Mozilla».

Если вы поклонник продуктов Netscape/Mozilla и желаете присоединиться к активному сообществу единомышленников, то TransforMiix – ваш верный выбор. Если же вы просто хотите работать с XSLT-процессором, который можно применять в готовом виде, то я бы пока предостерег вас от использования TransforMiix. Тем не менее, работоспособные бинарные версии для Linux и Mozilla уже появляются, так что ситуация очень скоро может измениться.

Unicorn

Набор инструментов Unicorn XML Toolkit базируется на основе XSLT-процессора, написанного на C++. Данный продукт работает только в операционной системе Microsoft Windows. Его выпускает компания Unicorn Enterprises, расположенная в Швице (Schwyz), Швейцария. Компания занимается разработкой программного обеспечения и предоставлением консультационных услуг в сфере обработки документов. Подробности можно узнать по адресу *http://www.unicorn-enterprises.com/*.

Продукт распространяется на условиях коммерческого лицензионного соглашения, в эти условия входит поставка исходного кода, а также стандартный договор о поддержке, однако некоторые компоненты, в числе которых и бинарный XSLT-процессор, можно загрузить бесплатно. Мне без труда удалось загрузить и установить программу, не проходя утомляющих процедур регистрации. Просто распакуйте файл `.zip` в подходящий каталог, скажем `c:\unicorn`, и запустите процессор из этого каталога:

```
uxt  demo\demo.xml  demo\demo.xsl  c:\output.html
```

Коммерческий продукт скомпонован так, что можно выбрать один из трех вариантов:

- Стандартное издание включает версию XSLT-процессора, который может работать из командной строки. Оно содержит библиотеку функций расширения, поддерживающих входные данные различных текстовых форматов, а также возможности создания отчетов.
- Издание с поддержкой баз данных содержит дополнительно библиотеку функций расширения, поддерживающих доступ к реляционным базам данных.
- Профессиональное издание содержит объектные расширения, обеспечивающие интеграцию с помощью архитектуры Microsoft ActiveX.

В продукт входит XML-анализатор, не осуществляющий проверку на действительность, соответствующий C++-версии SAX2, реализация DOM1, автономная реализация XPath и, конечно, процессор XSLT 1.0. Также имеются интерпретатор ECMAScript и библиотека регулярных выражений. Наиболее необычно то, что имеется реализация форматирующих объектов XSL.

Поставщик заявляет о полном соответствии продукта стандартам XSLT 1.0 и XPath 1.0. Я не смог выполнить некоторые из своих тестовых примеров (в том числе и примеры из этой книги), но такие случаи соответствовали действительно нечетким областям спецификации.

Помимо командной строки, процессор преобразований можно запускать из кода C++ или ECMAScript (то есть JavaScript для нас с вами). Функции расширения могут быть написаны с помощью ECMAScript, а возможности объектной интеграции в профессиональном издании дают также доступ к произвольным COM-объектам.

Доступ к базам данных

В состав Unicorn входит полный и хорошо документированный модуль функций и элементов расширений, обеспечивающих доступ к реляционным базам данных, как правило, через ODBC. Вероятно, будет лучше продемонстрировать эту возможность на паре примеров.

Следующая таблица стилей берет данные из исходного документа и заносит их в базу данных. На выходе преобразования будет пустое дерево.

```
<xsl:template match='/'>
  <root>
    <sql:connect
        source="{$source}"
        user="{$user}"
        authentication="{$authentication}"
        type="odbc"
        connection-id="dbc1"/>
    <sql:prepare
        connection-id="dbc1"
        sql="INSERT INTO PRODUCT (CODE, CATEGORY, NAME)
                 VALUES (?, ?, ?)"
        statement-id="stmt1">
      <sql:param type="CHAR" precision="64"/>
      <sql:param type="CHAR" precision="64"/>
      <sql:param type="CHAR" precision="64"/>
    </sql:prepare>
    <xsl:apply-templates/>
  </root>
</xsl:template>
<xsl:template match="product">
  <sql:execute statement-id="stmt1">
    <sql:with-param value="{code}"/>
    <sql:with-param value="{category}"/>
```

```
      <sql:with-param value="{name}"/>
    </sql:execute>
  </xsl:template>
```

Таблица стилей может также считывать данные из базы. Следующая таблица стилей выбирает все записи из таблицы PRODUCT, относящиеся к определенной категории (указанной в качестве параметра таблицы стилей), а затем заносит XML-представление информации о продуктах в конечное дерево. В данном случае содержимое исходного документа игнорируется.

```
<xsl:param name="category" select="'Desktop'"/>
<xsl:template match='/'>
  <root>
    <sql:connect
        source="{$source}"
        user="{$user}"
        authentication="{$authentication}"
        type="odbc"
        connection-id="dbc1"/>
    <sql:prepare
        connection-id="dbc1"
        sql="SELECT CODE, CATEGORY, NAME FROM PRODUCT
                 WHERE CATEGORY=? ORDER BY CODE"
        statement-id="stmt1">
      <sql:param type="CHAR" precision="50"/>
    </sql:prepare>
    <sql:for-each statement-id="stmt1">
      <sql:with-param value="{$category}"/>
      <product>
        <code><xsl:value-of select="sql:fetch('CODE')"/></code>
        <category><xsl:value-of select="sql:fetch('CATEGORY')"/></category>
        <name><xsl:value-of select="sql:fetch('NAME')"/></name>
      </product>
    </sql:for-each>
  </root>
</xsl:template>
```

Формирование отчетов

Расширения Unicorn XSLT, которые предназначены для формирования отчетов, можно, в сущности, разделить на две категории: поддержку именованных счетчиков и поддержку группировки.

Именованные счетчики могут быть объявлены, инициализированы, инкрементированы и считаны при помощи различных форм элемента <rpt:counter>. Подобно инструкции Saxon <saxon:assign>, данный элемент являет собой вопиющее несоответствие философии XSLT «без побочных эффектов» и позволяет вернуться к более традиционному процедурному стилю программирования.

Группировка осуществляется с помощью элемента <rpt:group>. Внешний элемент <rpt:group> применяется для определения итераций, которые исполняются для каждой группы узлов, имеющих одно и то же значение для некоторого выражения группировки. Вложенная инструкция <rpt:group> применяется для итераций по узлам внутри каждой группы. Например:

```
<rpt:group select="город" key="@страна">
  <страна название="{@страна}">
    <rpt:group>
      <xsl:copy-of select="город"/>
    </rpt:group>
  </страна>
</rpt:group>
```

Сортировку групп можно произвести с помощью элемента <rpt:sort>, синтаксис которого похож на синтаксис элемента <xsl:sort>.

Считывание форматированных текстовых файлов

Unicorn предоставляет пакет элементов и функций расширения, предназначенных для считывания содержимого текстовых файлов, форматированных либо колонками с фиксированной шириной, либо разделителями, например файлы значений, разделенных запятой.

Элемент <txt:format>, содержащий элементы <txt:field>, определяет формат входных записей. Это может быть либо формат с фиксированной шириной колонок:

```
<txt:format name="TEXT01">
      <txt:field name="КОД" width="10"/>
      <txt:field name="КАТЕГОРИЯ" width="15"/>
      <txt:field name="НАЗВАНИЕ" width="50"/>
</txt:format>
```

либо формат с разделителями:

```
<txt:format name="TEXT02" separator=",">
      <txt:field name="КОД"/>
      <txt:field name="КАТЕГОРИЯ"/>
      <txt:field name="НАЗВАНИЕ"/>
</txt:format>
```

Записи считываются из исходного файла при помощи инструкции <txt:read>; отдельные поля можно получить при помощи функции расширения txt:field-value():

```
<txt:read href="text02.txt" format="TEXT02">
   <изделие>
      <код><xsl:value-of select="txt:field-value('КОД')"/></код>
      <категория><xsl:value-of select="txt:field-value('КАТЕГОРИЯ')"/></категория>
      <название><xsl:value-of select="txt:field-value('НАЗВАНИЕ')"/></название>
   </изделие>
</txt:read>
```

XESALT

XESALT – это относительно малоизвестный процессор XSLT от немецкой компании Inlogix, основанной в Лейпциге, Германия. Данный процессор является единственным продуктом компании. Подробности можно узнать по адресу *http://www.inlogix.de/xesalt.html*.

Программа сконфигурирована для работы в трех различных средах:

- как процессор, запускаемый из командной строки
- как модуль веб-сервера
- как средство, встраиваемое в Netscape Communicator 4.X для осуществления преобразований на стороне клиента

В настоящий момент процессор описывается как предварительный выпуск продукта, который «вероятно, имеет ряд недостатков». Следовательно, вы знаете, по крайней мере, что имеете дело с честной компанией. Веб-сайт предлагает бесплатно скачать продукт для оценки, но я не стал этого делать, так как, во-первых (в феврале 2001 года), говорилось, что срок действия бесплатной лицензии истекает в сентябре 2000 года; и во-вторых, потому что не смог найти кнопку для загрузки.

Пока я отнес бы этот продукт к категории тех, за которыми стоит следить, а не тех, на которые стоит потратить время уже сейчас.

XML Spy

Продукт XML Spy выпускается компанией Altova GmbH из Вены, Австрия. Компания была основана в 1992 году, и недавно сменила свое предыдущее название Icon Information Systems. Но, кажется, новое имя не так уж и любимо компанией: поиск как по старому, так и по новому имени приводит на веб-сайт XML Spy, который расположен по адресу *http://www.xmlspy.com/*. Коммерческая стоимость продукта составляет 200 долларов США за индивидуальную пользовательскую лицензию. Вы можете загрузить 30-дневную версию для оценки.

XML Spy не является XSLT-процессором как таковым, скорее, это интерактивный редактор XML и среда разработки. Одна из многих задач, которые вы можете выполнить с его помощью, – это редактирование таблиц стилей и проверка их работы.

XML Spy отображает иерархическую структуру документа в виде вложенных таблиц, такое изображение называется сеточным представлением. Ниже показан пример такого представления. Им можно пользоваться для непосредственного редактирования документа, выбирая конкретный элемент с помощью правой клавиши мыши, и затем предпринимая одно из следующих действий: вставка (Insert), присоединение (Append) или добавление непосредственного потомка (Add Child) для создания дополнительных узлов, связанных с выбранным узлом. В любом случае на ваш выбор предоставля-

ются только те действия, которые не нарушают корректную структуру, принимая во внимание ограничения, накладываемые определением DTD или схемой, если таковые имеются.

Ниже показан рисунок, демонстрирующий, как выглядит в сеточном представлении XML-версия шекспировской трагедии «Отелло»:

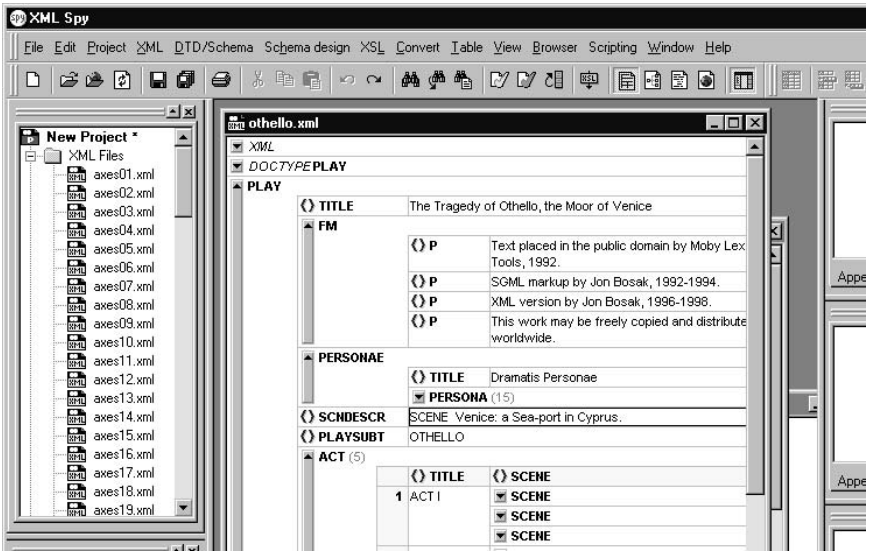


Рис. Е.3. *Сеточное представление XML-документа в виде вложенных таблиц*

Тем же образом можно использовать XML Spy для редактирования таблиц стилей, просматривая их в сеточном или текстовом представлении, как показано ниже:

Рис. Е.4. *Таблица стилей XSLT в сеточном представлении XML Spy*

А вот та же таблица стилей в текстовом представлении:

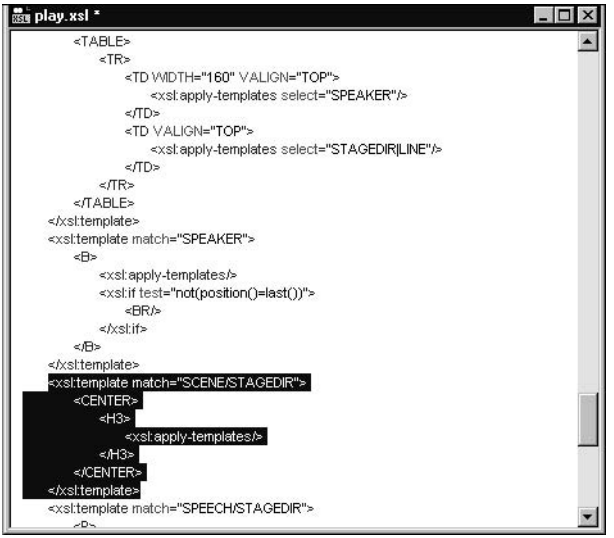


Рис. E.5. *Таблица стилей XSLT в текстовом представлении XML Spy*

XML Spy позволяет связывать таблицу стилей с определенным исходным документом XML, а затем выполнять преобразование. Результат преобразования можно также просматривать в различном виде. В том числе можно отображать выходной документ XML или HTML непосредственно в броузере.

Одним из упущений является то, что во время осуществления преобразования нет прямого способа задания параметров таблицы стилей. Можно задавать опции для XSLT-процессора, но только при помощи редактирования командной строки, вызывающей процессор. Другой недостаток состоит в том, что для преобразования данного исходного документа с помощью таблицы стилей нужно изменить инструкцию обработки `<?xml-stylesheet?>` документа, что необязательно совпадает с вашим желанием.

Продукт может работать с рядом различных процессоров XSLT: на выбор предлагаются MSXML3, Saxon и Infoteria iXSLT, кроме того, можно выбрать другой процессор, отредактировав командную строку, вызывающую процессор. Если не удается выполнить преобразование, то сообщение об ошибке, выдаваемое процессором XSLT, отображается во всплывающем окне, и затем XML Spy позволяет перейти к той строчке таблицы стилей, в которой была обнаружена ошибка.

Помимо обработки таблиц стилей, в XML Spy имеются возможности для создания и редактирования определений DTD и схем XML (имеющихся в нескольких вариациях, в связи с медленной разработкой стандарта). В продукте имеется несколько удобных средств для формирования DTD и схем из экземпляров документов, а также для преобразования между различными видами схем. Однако эти возможности выходят за границы нашего рассмотрения.

XSL Composer

Это коммерческий продукт, доступный на веб-сайте фирмы Whitehill Technologies по адресу *http://www.whitehill.com/*. Это не XSLT-процессор и даже не средство, предназначенное в первую очередь для создания таблиц стилей XSLT. Я охарактеризовал бы данный продукт скорее как средство для определения способа отображения ваших XML-документов в HTML-виде, которое использует XSLT для преобразований.

Это инструмент для веб-дизайнеров и разработчиков HTML, а не для разработчиков XSLT. Если вы уже знакомы с XSLT, то, скорее всего, разочаруетесь в данном инструменте, потому что с его помощью можно задавать только внешний вид результата, но не логику требуемого преобразования.

Главный *modus operandi*[1] данного средства заключается в выборе отдельных элементов из XML-документа и описании правил того, как они должны выглядеть в HTML. Эти правила можно разделить на две категории: стиль параграфа или стиль таблицы.

Стиль параграфа достаточно прост: в вашем распоряжении имеется набор форм с закладками, которые позволяют выбрать все возможные вариации цвета, размера и выравнивания, а также все остальное, что вы можете применить к тексту.

Создание таблиц показалось мне более трудной задачей. Когда XML-данные представляли собой достаточно простую структуру из строк и столбцов, эта задача удавалась мне достаточно успешно, но в случае более сложной структуры мне было трудно управлять происходящим. Так, к примеру, я не нашел способа создать HTML-таблицы для отображения рядом персонажей и их реплик в пьесе. У меня есть сильное подозрение, что это возможно, но в этом случае вам потребуются особые курсы, чтобы научиться это делать.

Ограничением данного подхода к разработке является то, что оно начинается с XML-документа, а не схемы или DTD. Поэтому нет способа указать инструменту, что факт нахождения в элементе `<годовые-продажи>` двенадцати элементов `<месячные-продажи>` является закономерностью для всей таблицы стилей, в противном случае исходный документ не корректен.

В продукт входит визуальное средство XPath Builder, которое может быть полезным, если вы не знакомы с основами XPath.

В целом же я полагаю, что большинство программистов, особенно знающих XSLT, сочтет этот продукт бесполезным. Он создан для непрограммистов и тех, кто считает создание таблиц стилей XSLT черной магией. Количество таких людей, конечно, значительно превосходит количество читателей этой книги. Но не мне судить, представляет ли данный продукт интерес для этой категории пользователей.

---

[1] Образ действий, принцип работы (лат.). – *Примеч. перев.*

XSLTC

Подробности о продукте, который в настоящее время описывается как альфа-релиз, можно найти по адресу *http://www.sun.com/xml/developers/xsltc/*.

XSLTC от Sun не похож на все остальные процессоры XSLT, описанные в данной книге, тем, что он представляет собой компилятор, который, имея на входе таблицу стилей, генерирует исполняемый код. В действительности это не машинный код, а байт-код Java. Этот байт-код точно такой же, что и на выходе компилятора Java; он выводится в файлы с расширением `.class`, которые могут исполняться виртуальной машиной Java. Файл `.class`, формируемый компилятором XSLTC, известен (по моему мнению, без особой необходимости) как «транслет» (translet), который может исполняться виртуальной машиной Java точно так же, как и скомпилированный класс Java.

Утверждается, что транслеты являются компактными и быстрыми, что делает их идеальными для осуществления преобразований на устройствах «тонкого клиента», таких как переносные компьютеры или мобильные телефоны.

Утверждение о малом размере файлов определенно подтверждается: когда я скомпилировал таблицу стилей, состоящую из 750 строк (ту, что используется для форматирования спецификации XML, описанной в главе 10), то в результате получил набор файлов для девяти классов, занимающих в целом 30 Кбайт. Когда я указал компилятору осуществлять вывод в файл `.jar`, то размер сократился до 13 Кбайт, что значительно меньше размера исходной таблицы стилей. При этом, безусловно, нужно загрузить еще и библиотеку времени выполнения, состоящую из двух файлов: `xml.jar` (анализатор XML) размером 127 Кбайт и `xsltrc.jar` (библиотека времени выполнения для осуществления XSLT-преобразования) размером 109 Кбайт. Это существенно меньше, чем Saxon (555 Кбайт) или Xalan (687 Кбайт), хотя и ненамного меньше, чем xt (345 Кбайт).

Что касается скорости, то, проведя простое тестирование, я нашел, что преобразование имеет приблизительно такую же производительность, что и в xt или Saxon. При условии, что данная версия является альфа-релизом, можно ожидать, что скорость будет значительно увеличена, однако это дело будущего. Для простых преобразований производительность определяется скорее временем анализа входных данных, построения дерева и форматирования выходных данных, а поскольку эти задачи по большей части не зависят от таблицы стилей, то нет причин полагать, что формирование скомпилированного кода ускорит процесс преобразования. Это также не ускорит выполнение таких расходующих время задач, как сортировка.

Для меня выглядит некоторым парадоксом, что компания Sun занимается продвижением подобной технологии, ведь в сфере производительности Java именно они показали, что хороший интерпретатор, использующий такие методики, как оперативная (just-in-time) компиляция (например, HotSpot), может быть, по крайней мере, таким же быстрым, как и традиционный скомпилированный код. Это может объяснить, почему Sun в своих утверждениях делает акцент именно на размере, а не на скорости.

Но существуют и другие потенциальные преимущества компиляции таблицы стилей в исполняемый код. Например, компиляция скрывает исходный код, что может быть полезно, когда существенная часть вашего коммерческого приложения написана на XSLT. Кроме того, скомпилированный код можно легко переносить; многие продукты, такие как Saxon, формирующие «скомпилированную» таблицу стилей в памяти, не позволяют переносить скомпилированную таблицу стилей между машинами или загрузить ее для исполнения в броузере. Даже такой продукт, как Xalan, который осуществляет сериализацию таблицы стилей, все-таки существенно снижает производительность за счет сериализации и затем десериализации. Поэтому рассматриваемая технология определенно имеет право на существование наряду с традиционным подходом интерпретации.

К моменту написания текущая версия XSLTC не полностью поддерживала спецификацию XSLT 1.0, но ограничения были незначительны. Я нашел несколько ошибок как в логике преобразования, так и в самой среде (например, в версии для Windows при выполнении сценария командной строки подавлялся вывод сообщений об ошибках). Но для продукта, заявленного как альфа-релиз, проблемы были минимальны.

xt

Продукт **xt** является реализацией XSLT с открытыми исходными текстами, созданной редактором спецификации Джеймсом Кларком. Это была первая доступная реализация и до сих пор одна из самых быстрых, однако стандарт XSLT 1.0 в ней реализован неполностью. Кроме того, продукт больше не разрабатывается (именно поэтому я и не стал выделять для него отдельное приложение). Последняя версия, упоминаемая как версия 19991105, по-прежнему описывается как бета-релиз.

Подобно Saxon, xt – плод усилий одного человека, а не корпоративная разработка. Продукт доступен на очень гибких условиях, позволяющих, по сути дела, делать с ним все что угодно, но ограждая автора от каких-либо претензий. Поддержка и гарантии отсутствуют. Продукт написан на Java и поставляется с исходным кодом.

xt можно найти по адресу *http://www.jclark.com/xml/xt.html*.

В продукте не поддерживаются следующие элементы рекомендаций по XSLT 1.0 и XPath 1.0:

- Механизм элементов расширения (атрибуты `extension-element-prefixes` и `xsl:extension-element-prefixes`, элемент `<xsl:fallback>` и функция `element-available()`)
- Ключи (элемент `<xsl:key>` и функция `key()`)
- Элемент `<xsl:decimal-format>` и необязательный третий аргумент функции `format-number()`, который ссылается на элемент `<xsl:decimal-format>`
- Ось namespace

- Обработка в режиме совместимости с последующими версиями
- Атрибут `xsl:exclude-result-prefixes` конечных литеральных элементов (но атрибут `exclude-result-prefixes` элемента `<xsl:stylesheet>` реализован)

Другим ограничением является то, что не во всех случаях выдается сообщение об ошибках, а это значит, что неправильная таблица стилей может показаться правильной в xt, в то время как соответствующий стандарту XSLT-процессор забракует ее. Например, xt позволяет объявить локальную переменную, когда в области видимости имеется другая переменная с тем же именем.

Запуск процессора xt

Процессор xt может быть запущен из командной строки, через Java API или как сервлет.

Для полной версии продукта типичная командная строка выглядит следующим образом:

```
java com.jclark.xsl.sax.Driver source.xml style.xsl out.html param1=value1...
```

Продукт также распространяется в виде исполняемого файла Windows, в этом случае можно набрать:

```
xt source.xml style.xsl out.html param1=value1...
```

Для работы под Windows данная версия в установке и использовании представляет собой воплощение простоты.

Как следует из имени использованного Java-класса, xt устроен как фильтр SAX, принимая на входе поток событий SAX от XML-анализатора, и возвращая конечному процессу форматирования или другому потоку SAX другой поток SAX-событий.

xt работает с любым SAX1-совместимым анализатором XML, который может быть указан с помощью системного свойства. Однако рекомендуемым является анализатор Джеймса Кларка xp, который обладает интерфейсом SAX1, расширенным таким образом, чтобы можно было передавать в приложение комментарии.

Параметры `param1=value1` указывают значения для любых глобальных параметров; значения должны быть строкового типа.

Процессор xt имеет также интерфейс Java API, позволяющий запускать его из сервлета. К сожалению, со скоростью работы кода Джеймса Кларка может соперничать только краткость документации к нему. Поэтому, если вы хотите пользоваться этим Java API, то будьте готовы к глубокому исследованию структуры продукта. Есть, правда, несколько исходных файлов примеров (внутри архива `xt.jar`, который распаковывается с помощью WinZip или любого другого распаковщика ZIP-файлов), в которых показано, как нужно использовать интерфейсы SAX и DOM, предоставляемые xt, а также как запускать процессор из сервлета.

Расширяемость

Продукт xt позволяет создавать пользовательские функции расширения на языке Java.

Класс Java определяется непосредственно по URI пространства имен, использованному при вызове функции. Вы можете осуществить следующий вызов:

```
<xsl:variable name="сегодня" select="Date:new()"
              xmlns:Date="http://www.jclark.com/xt/java.util.Date"/>
```

без объявления внешнего класса. Класс берется из URI пространства имен как та часть, которая следует после «`http://www.jclark.com/xt/`».

Таким образом в xt можно вызывать как статические методы, так и конструкторы. Для вызова метода экземпляра нужно передать дополнительный первый аргумент: внешний объект Java, который был получен ранее как результат вызова статического метода или конструктора.

Подробности соответствия типов данных XPath и Java можно найти в документации, поставляемой с продуктом.

xt в настоящий момент не поддерживает элементы расширения, хотя в нем реализовано синтаксическое подмножество для использования в собственных элементах расширения, описанных ниже.

Расширения

xt поддерживает несколько расширений, включая вывод нескольких документов, вывод не в XML-формате, пользовательские обработчики вывода, а также ряд функций расширения. Эти расширения описываются в следующих разделах.

Вывод нескольких документов

Процессор xt поддерживает элемент расширения `<xt:document>`, с помощью которого таблица стилей может создавать несколько выходных документов. Префикс `xt` (или другой выбранный вами префикс) должен указывать на следующий URI пространства имен: `http://www.jclark.com/xt`. Действие элемента `<xt:document>` очень похоже на действие инструкции `<xsl:document>`, введенной в рабочем проекте спецификации XSLT 1.1, которая описана в главе 4.

Элемент `<xt:document>` имеет обязательный атрибут `href`, который должен быть относительным URI. Значение атрибута `href` рассматривается как шаблон значения атрибута. Содержимое элемента `<xt:document>` – это тело шаблона для конечного дерева, которое должно быть сохранено там, куда указывает атрибут `href`. Базовый URL для определения относительного URL – это URL родительского выходного документа: то есть либо URL основного выходного документа, либо URL документа, в котором хранится родительский элемент `<xt:document>`. Таким образом, тот же относительный URL, указанный в атрибуте `href`, может использоваться в родительском документе для ссылок на документ, созданный с помощью элемента `<xt:document>`.

Элемент <xt:document> может также иметь все те же атрибуты, что и элемент <xsl:output>. Эти атрибуты объединяются с атрибутами, указанными в элементах <xsl:output> верхнего уровня, для определения метода вывода документа. Атрибуты элемента <xt:document> имеют приоритет над атрибутами, указанными в элементах <xsl:output> верхнего уровня.

Вывод в формате, отличающемся от XML

В XSLT имеется возможность формировать текстовый вывод при помощи <xsl:output method="text">; однако здесь есть ограничения. Например, невозможно вывести управляющие символы, не входящие в набор символов XML.

Поэтому xt предоставляет альтернативный способ текстового вывода с помощью метода вывода «xt:nxml», где префикс xt связан с URI пространства имен http://www.jclark.com/xt. Такой метод формирует дерево XML, определенные элементы которого имеют особое значение:

- Элемент <char> позволяет выводить символы, которые не разрешены в XML, например большинство управляющих символов с кодовым значением меньше #x20.
- Элемент <data> содержит данные. Внутри элемента <data> специальные символы экранируются.
- Элемент <escape> определяет специальный символ и устанавливает, каким образом он будет экранироваться.
- Элемент <control> содержит символы, которые следует выводить в исходном виде, без экранирования.

Пример: Формирование текстового вывода в xt

Исходный документ

Данная таблица стилей может применяться к любому исходному документу.

Таблица стилей

Таблица стилей содержится в файле xt-text.xsl.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xt:nxml" xmlns:xt="http://www.jclark.com/xt"/>
<xsl:template match="/">
   <nxml>
      <escape char="\">\\</escape>
      <data>&amp;&lt;&gt;\</data>
      <control>&amp;&lt;&gt;\</control>
   </nxml>
</xsl:template>
</xsl:stylesheet>
```

Результат

```
&<>\\&<>\
```

Пользовательские обработчики вывода

Атрибут method элементов <xsl:output> и <xt:document> может принимать форму «java:package.name.class», где префикс «java» связан с URI пространства имен http://www.jclark.com/xt/java, а «package.name.class» – полное имя Java-класса, реализующего интерфейс com.jclark.xsl.sax.OutputDocumentHandler (являющийся расширением интерфейса SAX1 org.xml.sax.DocumentHandler).

В результате узлы конечного дерева будут направлены пользовательскому обработчику документа.

Функции расширения

xt предлагает следующие функции расширения. Для них используется следующий URI пространства имен: http://www.jclark.com/xt.

xt:node-set (frag)	Преобразует фрагмент конечного дерева к эквивалентному набору узлов. Аргумент должен быть набором узлов или фрагментом конечного дерева. Результат представляет собой набор узлов.
xt:intersection (ns1, ns2)	Возвращает пересечение двух наборов узлов (узлы, присутствующие в обоих указанных наборах).
xt:difference (ns1, ns2)	Возвращает разность двух наборов узлов (узлы из первого набора, отсутствующие во втором наборе).

Заключение

В этом приложении я дал краткий обзор более чем дюжины продуктов, имеющих отношение к XSLT. В их число входят надежные корпоративные продукты, которые можно использовать уже сейчас; развивающиеся продукты, чьи технологии может ждать большое будущее; а также некоторые продукты, разработка которых, по всей видимости, приостановилась.

Общее заключение таково, что данная область весьма подвижна. Очень многое меняется, и я не надеюсь, что смог охватить все. Тем более я не смею надеяться, что в нескольких параграфах смог дать справедливую оценку четырнадцати очень различающимся программным продуктам, каждый из которых стоил нескольких лет усердной работы талантливейших разработчиков. Убедительно прошу учесть, что я описывал продукты в начале 2001 года, и спустя шесть или двенадцать месяцев это описание может не соответствовать действительности. Помните также, что мой подход к оценке сильных и слабых сторон продукта может не совпадать с вашим. Я лишь надеюсь, что вы найдете мой анализ полезным в качестве руководства для вашего дальнейшего исследования и тестирования продуктов на предмет удовлетворения вашим собственным требованиям.

F

TrAX: API для XML-преобразований

TrAX (Transformation API for XML) – это Java API для управления XSLT-преобразованиями.

Одним из достижений XSLT стало то, что уже через несколько месяцев после опубликования первой версии стандарта XSLT 1.0 появился ряд продуктов, в которых был реализован высокий уровень соответствия этому стандарту. Некоторые из этих продуктов были написаны на Java. Поэтому пользователи хотели создавать приложения, в которых можно было бы запускать любой из этих XSLT-процессоров, что позволило бы им оценить производительность и защитить свои приложения от ошибок в том или ином процессоре.

Однако, к сожалению, каждый из этих процессоров предлагал различные API-интерфейсы Java, поэтому для использования какого-либо процессора в приложении нужно было писать отдельный модуль драйвера.

Тогда производители организовали неформальную встречу под предводительством Скотта Боэга (Scott Boag) из Lotus, чтобы оценить возможность использования общего API, так появился **TrAX**, API для XML-преобразований. Поначалу дело двигалось медленно, но, в конце концов, фирма Sun подхватила инициативу и решила опубликовать интерфейс, основанный на предложениях TrAX, как часть интерфейса Java API для обработки XML (Java API for XML Processing, JAXP). Окончательный релиз JAXP версии 1.1 был опубликован 6 февраля 2001 года. Подробности можно узнать по адресу:

http://java.sun.com/xml/download.html

Информация, приводимая в данном приложении, относится к последней версии JAXP 1.1. Теперь это уже достаточно стабильная версия, хотя нельзя исключать вероятность обнаружения незначительного количества ошибок.

К моменту написания книги интерфейс TrAX был достаточно полно реализован в продуктах Saxon и Xalan. Можно ожидать, что и другие продукты последуют их примеру.

Обозначения

Для экономии места в приложении использованы сокращенные обозначения для исключений, генерируемых в каждом методе. Список типа «[SAXE, IOE]» обозначает генерируемые исключения, используя следующие аббревиатуры:

FCE	javax.xml.parsers.FactoryConfigurationError
IAE	java.lang.IllegalArgumentException
IOE	java.io.IOException
PCE	javax.xml.parsers.ParserConfigurationException
SAXE	org.xml.sax.SAXException
SAXNRE	org.xml.sax.SAXNotRecognizedException
SAXNSE	org.xml.sax.SAXNotSupportedException
TCE	javax.xml.transform.TransformerConfigurationException
TE	javax.xml.transform.TransformerException
TFCE	javax.xml.transform.TransformerFactoryConfigurationError

API анализатора JAXP 1.1

JAXP фактически определяет два набора интерфейсов: интерфейсы для синтаксического анализа XML в пакете javax.xml.parsers и интерфейсы для XML-преобразований (то есть TrAX) в пакете javax.xml.transform и вложенных пакетах. Рассмотрение API для анализаторов выходит за рамки данной книги, но поскольку приложения зачастую используют оба набора интерфейсов, то мы сделаем краткий обзор API для анализаторов SAX и DOM.

Эти интерфейсы не заменяют интерфейсы SAX и DOM, которые описываются во многих справочниках по XML. Скорее, они дополняют их возможностями, отсутствующими в SAX и DOM, а именно возможностью выбирать, какой из анализаторов – SAX или DOM – использовать, а также устанавливать такие опции, как проверка на действительность и обработка пространств имен.

Мы посмотрим на эти две части, SAX и DOM, по отдельности.

Поддержка SAX в JAXP

JAXP 1.0 поддерживал интерфейс SAX1; JAXP 1.1 поддерживает SAX2, но остается совместимым с JAXP 1.0. Для краткости черты, относящиеся только к SAX1, здесь не приводятся.

javax.xml.parsers.SAXParserFactory

Первое, что должно сделать приложение – это получить доступ к объекту SAXParserFactory, чего можно достичь вызовом статического метода SAXParserFactory.newInstance(). Различные поставщики SAX-анализаторов реализуют свои собственные подклассы SAXParserFactory, вызов которых и определяет, анализатором какого поставщика будет пользоваться ваше приложение. Если имеется несколько доступных анализаторов, то выбор осуществляется на основании следующих действий:

- Используется значение свойства javax.xml.parsers.SAXParserFactory, если оно доступно. Как правило, системные свойства устанавливаются с помощью опции -D в командной строке java или с помощью вызова System.setProperty() из приложения.
- Просматривается файл свойств $JAVA_HOME/lib/jaxp.properties, и внутри этого файла ищется значение свойства с именем javax.xml.parsers.SAXParserFactory.
- Используются сервисные API, являющиеся частью спецификации JAR.

Скорее всего, в процессе установки конкретного анализатора файл, находящийся внутри архива .jar, устанавливает данный анализатор в качестве используемого по умолчанию. Поэтому, если ничего не предпринять для выбора анализатора, то автоматически выбранный анализатор будет зависеть от порядка файлов и директорий в переменной окружения CLASSPATH.

После того как получен доступ к SAXParserFactory, можно применить ряд методов для его конфигурации. И, наконец, можно вызвать метод newSAXParser() для получения экземпляра SAXParser. Ниже приводятся доступные методы. Аббревиатуры в квадратных скобках означают генерируемые исключения в соответствии с таблицей, приведенной в начале приложения.

Метод	Описание
boolean getFeature(String) [SAXNRE, SAXNSE, PCE]	Определяет, сконфигурирована ли фабрика анализатора для поддержки указанного свойства. Имена свойств соответствуют определенным в SAX2 для класса XMLReader.
boolean isNamespaceAware()	Определяет, поддерживают ли анализаторы, полученные с использованием данной фабрики, пространства имен.
boolean isValidating()	Определяет, осуществляют ли анализаторы, полученные с использованием данной фабрики, проверку XML на действительность.
static SAXParserFactory newInstance() [FCE]	Создает SAXParserFactory для конкретного анализатора, выбранного в соответствии с вышеуказанными правилами.

Метод	Описание
`SAXParser newSAXParser()` `[PCE]`	Возвращает экземпляр `SAXParser`, который может быть использован для осуществления синтаксического анализа. Этот объект является оболочкой для объекта SAX2 `XMLReader`.
`void setFeature(String, boolean)` `[SAXNRE, SAXNSE, PCE]`	Включает или отключает указанную возможность, название которой соответствует определенному в SAX2 для класса XMLReader.
`void setNamespaceAware(boolean)`	Указывает, должны ли анализаторы, полученные с помощью данной фабрики, распознавать пространства имен.
`void setValidating(boolean)`	Указывает, должны ли анализаторы, полученные с помощью данной фабрики, осуществлять проверку XML на действительность.

javax.xml.parsers.SAXParser

Объект `SAXParser` получают с помощью метода `newSAXParser()` фабрики `SAXParserFactory`. `SAXParser` является оболочкой для объекта SAX2 `XMLReader`. Сам же объект `XMLReader` может быть оболочкой для объекта SAX1 `Parser`. Объект `XMLReader` можно получить с помощью метода `getXMLReader()`, но в обычных случаях такой необходимости нет, поскольку можно выполнить анализ и назначить обработчик для всех событий, происходящих в ходе анализа, используя только сам класс `SAXParser`.

Ниже приведены методы, имеющие отношение к SAX2-анализаторам:

Метод	Описание
`Object getProperty(String)` `[SAXNRE, SAXNSE]`	Получает значение свойства SAX2 `XMLReader`.
`XMLReader getXMLReader()` `[SAXE]`	Возвращает объект SAX2 `XMLReader`.
`boolean isNamespaceAware()`	Определяет, распознает ли SAX2 `XMLReader` пространства имен.
`boolean isValidating()`	Определяет, выполняет ли соответствующий SAX2 `XMLReader` проверку на действительность XML.
`void parse(File,` `DefaultHandler)` `[IOE, IAE, SAXE]`	Осуществляет синтаксический анализ содержимого файла `File`, передавая все события, возникающие в процессе анализа указанному обработчику. Это, как правило, не `DefaultHandler`, определенный в SAX2, а определенный пользователем подкласс `DefaultHandler`, написанный для выборочной обработки событий.
`void parse(InputSource,` `DefaultHandler)` `[IOE, IAE, SAXE]`	Анализирует содержимое указанного SAX `InputSource`, передавая все события, возникающие в процессе анализа, указанному обработчику.

Метод	Описание
`void parse(InputStream, DefaultHandler) [IOE, IAE, SAXE]`	Анализирует содержимое указанного `InputStream`, передавая все события, возникающие в процессе анализа, указанному обработчику. Учтите, что в этом случае системный идентификатор входного потока неизвестен, а потому анализатор не имеет возможности определить полные URI на основе относительных URI, имеющихся в исходных XML-данных.
`void parse(InputStream, DefaultHandler, String) [IOE, IAE, SAXE]`	Анализирует содержимое указанного `InputStream`, передавая все события, возникающие в процессе анализа, указанному обработчику. В третьем аргументе передается системный идентификатор, который будет использоваться для разрешения относительных URI.
`void parse(String, DefaultHandler) [IOE, IAE, SAXE]`	Анализирует содержимое XML-документа, указанного с помощью URI в первом аргументе, передавая все события, возникающие в процессе анализа, указанному обработчику.
`void setProperty(String, Object) [SAXNSE, SAXNRE]`	Устанавливает значение указанного свойства SAX2 `XMLReader`.

Поддержка DOM в JAXP

Если JAXP 1.0 поддерживал первоначальную спецификацию DOM первого уровня, то JAXP 1.1 ориентируется на DOM второго уровня. В DOM второго уровня появился ряд возможностей, облегчающих навигацию по документу, но основным достижением (и источником сложностей) стала поддержка пространств имен XML.

Сама по себе DOM определяет методы для построения дерева документа программным путем, а также методы для навигации по этому дереву, но она не определяет какие-либо способы построения дерева DOM путем синтаксического анализа исходного XML-документа. JAXP 1.1 призван заполнить этот пробел.

Архитектура интерфейсов в DOM и SAX имеет много сходств:

- Сначала вызывается статический метод `DocumentBuilderFactory.newInstance()` для получения доступа к фабрике `DocumentBuilderFactory`, представляющей конкретную реализацию DOM.
- Затем вызывается метод `newDocumentBuilder()` этой фабрики, в результате чего создается экземпляр `DocumentBuilder`.
- И наконец, вызов одного из многочисленных методов `parse()` этого `DocumentBuilder` приводит к созданию объекта DOM `Document`.

javax.xml.parsers.DocumentBuilderFactory

Первое, что должно сделать приложение – это получить доступ к объекту DocumentBuilderFactory, чего можно достичь с помощью вызова статического метода DocumentBuilderFactory.newInstance(). Различные поставщики DOM-анализаторов реализуют свои собственные подклассы SAXParserFactory, вызов которых и определяет, какой реализацией будет пользоваться ваше приложение. Если имеется несколько доступных анализаторов, то выбор осуществляется на основании следующих действий:

- Используется значение системного свойства javax.xml.parsers.DocumentBuilderFactory, если к нему есть доступ. Как правило, системные свойства устанавливаются с помощью опции -D в командной строке java или с помощью вызова System.setProperty() из приложения.
- Просматривается файл свойств $JAVA_HOME/lib/jaxp.properties и внутри этого файла ищется значение свойства с именем javax.xml.parsers.DocumentBuilderFactory.
- Используются сервисные API, являющиеся частью спецификации JAR.

Скорее всего, в процессе установки конкретной реализации DOM файл, находящийся внутри архива .jar, устанавливает данный анализатор в качестве используемого по умолчанию. Поэтому если ничего не предпринять для выбора анализатора, то автоматически выбранный анализатор будет зависеть от порядка файлов и директорий в переменной окружения CLASSPATH.

После того как получен доступ к DocumentBuilderFactory, можно применить ряд методов для его конфигурации и вызвать метод newDocumentBuilder() для получения экземпляра DocumentBuilder.

Ниже приводятся имеющиеся методы:

Метод	Описание
Object getAttribute(String) [IAE]	Возвращает информацию о свойствах соответствующей реализации.
boolean isCoalescing()	Определяет, будет ли полученный DocumentBuilder объединять узлы CDATA с соседними текстовыми узлами.
boolean isExpandEntityReferences()	Определяет, будет ли полученный DocumentBuilder раскрывать ссылки на сущности и объединять их содержимое с соседними текстовыми узлами.
boolean isIgnoringComments()	Определяет, будет ли полученный DocumentBuilder игнорировать комментарии в исходном XML-файле.
boolean isIgnoringElementContentWhitespace()	Определяет, будет ли полученный DocumentBuilder игнорировать пробельные символы в содержимом элементов.

Метод	Описание
`boolean isNamespaceAware()`	Определяет, будет ли полученный `DocumentBuilder` распознавать пространства имен.
`boolean isValidating()`	Определяет, будет ли полученный `DocumentBuilder` проверять исходные XML-данные на действительность.
`DocumentBuilder newDocumentBuilder()` `[PCE]`	Возвращает новый объект `DocumentBuilder`, сконфигурированный в соответствии с предыдущими вызовами.
`static DocumentBuilderFactory newInstance()` `[FCE]`	Возвращает `DocumentBuilderFactory` в реализации конкретного поставщика, выбранного в соответствии с вышеуказанными правилами.
`setAttribute(String, Object)` `[IAE]`	Устанавливает свойства для конкретной реализации.
`void setCoalescing(boolean)`	Определяет, будет ли полученный `DocumentBuilder` объединять узлы CDATA с соседними текстовыми узлами.
`void setExpandEntityReferences (boolean)`	Определяет, будет ли полученный `DocumentBuilder` раскрывать ссылки на сущности и объединять их содержимое с соседними текстовыми узлами.
`void setIgnoringComments(boolean)`	Определяет, будет ли полученный `DocumentBuilder` игнорировать комментарии в исходном XML-файле.
`void setIgnoringElementContent Whitespace(boolean)`	Определяет, будет ли полученный `DocumentBuilder` игнорировать пробельные символы в содержимом элементов.
`void setNamespaceAware(boolean)`	Определяет, будет ли полученный `DocumentBuilder` распознавать пространства имен.
`void setValidating(boolean)`	Определяет, будет ли полученный `DocumentBuilder` проверять исходные XML-данные на действительность.

javax.xml.parsers.DocumentBuilder

Экземпляр `DocumentBuilder` всегда создается с помощью вызова метода `newDocumentBuilder()` фабрики `DocumentBuilderFactory`.

`DocumentBuilder` выполняет задачу по синтаксическому анализу исходного XML-документа и помещению полученного экземпляра `org.w3.dom.Document`, содержащего корневой узел древовидного представления документа в память.

Исходный документ указывается так же, как и входные данные для SAX-анализатора. Это не означает, что `DocumentBuilder` должен использовать SAX-

анализатор для синтаксического разбора: некоторые будут работать таким образом, а некоторые нет. Такой подход используется для того, чтобы избежать излишних разногласий в подходах SAX- и DOM-моделей.

> *Возможно, вы знаете, что в реализации DOM от Microsoft класс* `Document` *имеет метод* `load()`*, который анализирует исходный XML-файл и конструирует объект* `Document`*. Данный метод является расширением Microsoft и не имеет аналога в спецификации W3C DOM. Данный класс* `DocumentBuilder` *заполняет этот пробел.*

Ниже приведены доступные методы:

Метод	Описание
`boolean isNamespaceAware()`	Показывает, распознает ли анализатор пространства имен XML.
`boolean isValidating()`	Показывает, осуществляет ли анализатор проверку исходного XML-файла на действительность.
`Document newDocument()`	Возвращает новый объект `Document` без содержимого. Заполнить полученный объект можно с помощью методов DOM, таких как `createElement()`.
`Document parse(File)` `[IOE, SAXE, IAE]`	Анализирует XML-данные в указанном файле и возвращает получившийся объект `Document`.
`Document parse(InputSource)` `[IOE, SAXE, IAE]`	Анализирует XML-данные из указанного SAX-источника `InputSource` и возвращает получившийся объект `Document`.
`Document parse(InputStream)` `[IOE, SAXE, IAE]`	Анализирует XML-данные из указанного `InputStream` и возвращает получившийся объект `Document`. Учтите, что в этом случае системный идентификатор входного документа неизвестен, а потому анализатор не имеет возможности определить полные URI на основе относительных URI, имеющихся в документе.
`Document parse(InputStream, String)` `[IOE, SAXE, IAE]`	Анализирует XML-данные из указанного `InputStream` и возвращает получившийся объект `Document`. Во втором аргументе передается системный идентификатор, на основе которого можно разрешать относительные URI.
`Document parse(String)` `[IOE, SAXE, IAE]`	Анализирует XML-данные, указанные с помощью URI, и возвращает получившийся объект `Document`.
`void setEntityResolver(EntityResolver)`	Указывает анализатору использовать переданный SAX `EntityResolver`.
`void setErrorHandler(ErrorHandler)`	Указывает анализатору использовать переданный SAX `ErrorHandler`.

API для преобразований JAXP 1.1

В предыдущих разделах был сделан обзор классов и методов, определенных в JAXP 1.1 для управления процессом синтаксического анализа XML. Давайте теперь взглянем на классы и методы, используемые для управления XSLT-преобразованием, набор которых известен как TrAX.

В действительности эти классы создавались так, чтобы их можно было использовать и с другими механизмами преобразований, отличными от XSLT, например, их можно применять для преобразования, определенного спецификацией W3C XQuery. Однако XSLT-преобразования были отправным пунктом, и им мы посвятим основное внимание.

Тем не менее, существует еще один тип преобразований, о котором стоит упомянуть, – это идентичные преобразования, результатом которых является создание копии входного документа. TrAX обеспечивает явную поддержку идентичных преобразований. Такие преобразования полезнее, чем может показаться на первый взгляд, поскольку TrAX определяет три способа передачи исходного документа (SAX, DOM или последовательный XML) и три способа выдачи полученного документа (SAX, DOM или последовательный XML), таким образом, идентичные преобразования могут быть использованы для конвертирования из одного формата в другой. Например, входные данные в модели SAX можно преобразовать в последовательный XML-файл или преобразовать входные данные DOM в выходной поток событий SAX.

Интерфейс TrAX также предназначен для управления составными преобразованиями, состоящими из последовательности единичных преобразований, каждое из которых определяется своей таблицей стилей XSLT. Для этого в него заложено понятие объекта SAX2 `XMLFilter`, который рассматривает входной документ как поток SAX-событий и формирует на выходе другой поток SAX-событий. Для определения составного преобразования можно организовать конвейер из произвольного количества таких фильтров.

Так же как интерфейсы JAXP – `SAXParser` и `DocumentBuilder`, TrAX позволяет выбрать конкретную реализацию XSLT с помощью объекта `TransformerFactory`. Обычно каждый поставщик XSLT предоставляет свой собственный подкласс `TransformerFactory`.

Для повышения производительности в рассматриваемом API разделены процессы компиляции таблицы стилей и ее исполнения. Таблица стилей может быть скомпилирована один раз, а затем неоднократно применяться к различным исходным документам, возможно, параллельно в различных потоках. Скомпилированная таблица стилей, следуя номенклатуре Microsoft MSXML3, называется объектом `Templates`. Однако для простоты существуют также методы, которые объединяют процесс компиляции и исполнения в один вызов.

Классы, определенные в модуле `javax.xml.transform`, можно разбить на несколько категорий:

Категория	Класс или интерфейс	Описание
Основные классы	`TransformerFactory`	Выбирает и настраивает конкретную реализацию.
	`Templates`	Соответствует скомпилированной таблице стилей в памяти.
	`Transformer`	Соответствует единичному применению таблицы стилей к исходному документу для получения преобразования.
	`SAXTransformerFactory`	Позволяет упаковать преобразование в SAX `XMLFilter`.
	`Source`	Соответствует входным данным преобразования.
	`Result`	Соответствует результату преобразования.
Исходные классы	`SAXSource`	Входные данные преобразования в виде потока SAX-событий.
	`DOMSource`	Входные данные преобразования в виде DOM `Document`.
	`StreamSource`	Входные данные преобразования в виде последовательного XML-документа.
Конечные классы	`SAXResult`	Выходные данные в виде потока SAX-событий.
	`DOMResult`	Выходные данные в виде DOM `Document`.
	`StreamResult`	Выходные данные в виде последовательного XML-документа (или HTML, или обычного текстового файла).
Вспомогательные классы	`URIResolver`	Указываемый пользователем объект, который на основе переданного ему URI, содержащегося в таблице стилей (например, в функции `document()`), извлекает соответствующий документ в виде объекта `Source`.
	`ErrorListener`	Указываемый пользователем объект, которому передаются сообщения о предупреждениях и ошибках. Объект `ErrorListener` извещает об этих условиях пользователя и решает, можно ли продолжать обработку.
	`SourceLocator`	Используется в основном для указания места в таблице стилей, где произошла ошибка.
	`DOMLocator`	Подкласс `SourceLocator`, используемый, когда исходные данные представлены в виде DOM.
	`OutputKeys`	Набор констант, определяющих имена свойств для выводимых последовательных файлов.

Категория	Класс или интерфейс	Описание
Классы ошибок	`Transformer-Configuration-Exception`	Обычно обозначает ошибку в таблице стилей, которая обнаружена во время компиляции.
	`Transformer-Exception`	Сбой, произошедший во время исполнения преобразования.
	`Transformer-FactoryConfigurationError`	Ошибка при конфигурации `Transformer`.

В следующих разделах приведены описания этих классов в алфавитном порядке по имени классов (без учета имени модуля).

javax.xml.transform.dom.DOMLocator

Класс `DOMLocator` используется для установления местонахождения ошибки, когда документ передается в форме DOM. Этот объект, как правило, создается процессором при возникновении ошибки, а доступ к нему можно получить при помощи метода `getLocator()` соответствующего объекта `Exception`. Он унаследован от класса `SourceLocator` и предоставляет один дополнительный метод:

Метод	Описание
`org.w3c.dom.Node getOriginatingNode()`	Возвращает узел, на котором произошла ошибка или другое событие.

javax.xml.transform.dom.DOMResult

Использование объекта `DOMResult` в качестве результата преобразования означает, что выходные данные должны быть помещены в память в виде DOM. Обычно этот объект создается приложением и передается процессору во втором аргументе метода `Transformer.transform()`.

`DOMResult` указывает на объект `Node` (который, как правило, является, объектом `Document` или `Element`, а также, возможно, `DocumentFragment`), предназначенный для хранения результата преобразования. Непосредственные потомки корня конечного дерева будут непосредственными потомками этого объекта `Node` в представлении DOM. Если приложение не указало никакого объекта `Node`, то система сама создаст узел `Document`, который можно извлечь с помощью метода `getNode()`.

Многие процессоры XSLT будут поддерживать в будущем вывод в форме DOM, но это не обязательно. Если процессор поддерживает такой вывод, то метод `getFeature(DOMResult.FEATURE)` возвратит истину.

Если таблица стилей XSLT выводит текст с опцией `«disable-output-escaping="yes"»`, то этому тексту в дереве будет предшествовать инструкция обработки, название которой соответствует значению константы `Result.PI_DISABLE_OUTPUT_ESCAPING`, а за ним будет следовать инструкция обработки, названная в соответствии со значением константы `Result.PI_ENABLE_OUTPUT_ESCAPING`.

Объект DOMResult хранит ссылку на целевой объект Node, а также строку String, в которой содержится системный идентификатор.

Этот класс имеет два конструктора:

```
DOMResult()
DOMResult(org.w3c.xml.Node)
```

а также следующие методы:

Метод	Описание
org.w3c.dom.Node getNode()	Возвращает узел, в котором будет содержаться конечное дерево.
String getSystemId()	Возвращает системный идентификатор.
void setNode(org.w3c.dom.Node)	Возвращает узел, в котором будет содержаться конечное дерево. Это должен быть Document, Element или DocumentFragment.
void setSystemId(String)	Устанавливает системный идентификатор.

javax.xml.transform.dom.DOMSource

DOMSource содержит DOM Document в качестве объекта Source, так что он может быть передан на вход преобразования.

Объект DOMSource обычно создается приложением и передается процессору как первый аргумент метода Transformer.transform(). Также он может быть использован для указания таблицы стилей; в этом случае он передается в качестве параметра в метод TransformerFactory.newTemplates().

Желательно вызывать метод setSystemId() для передачи URI документа, так чтобы можно было разрешать относительные URI (например, используемые в функции document()). В самом документе DOM эта информация не хранится, поэтому ее нужно получать извне.

Объект DOMSource может обозначать любой узел в модели DOM, он не обязан быть узлом Document. Например, если используется DOMSource для обозначения таблицы стилей, то это может быть таблица стилей, вложенная в другой документ; в этом случае объект DOMSource будет представлять собой узел элемента <xsl:stylesheet>. Когда в качестве входного параметра метода transform() передается узел, отличный от Document, то результат не определен однозначно, но, возможно, действие метода будет похоже на применение метода transformNode() в MSXML3. Это означает, что входные данные для преобразования определяются всем документом, но процесс преобразования начнется с поиска шаблонного правила для указанного узла, а не для корневого узла.

Следует заметить, что существует два способа, с помощью которых XSLT-процессоры могут обработать полученный документ. Они могут создать дерево в модели XPath как представление или оболочку для дерева DOM или же они могут создать его копию. Разница между этими двумя подходами становится видна, когда в таблице стилей осуществляется вызов внешних

функций Java, которые пытаются работать с деревом как с формой DOM. Это, очевидно, также влияет на производительность. В общем случае предпочтительнее передавать входные данные как поток или источник SAX, если имеется такая возможность. Не нужно специально создавать модель DOM в качестве исходных данных для преобразования, как это обычно делают люди, привыкшие к использованию интерфейса Microsoft MSXML API.

Не каждый процессор XSLT поддерживает ввод данных в форме DOM. Если же такой ввод поддерживается, то метод `getFeature(DOMSource.FEATURE)` возвратит истину.

В этом классе есть три конструктора:

```
DOMSource()
DOMSource(Node)
DOMSource(Node, String)
```

Ниже приведены методы класса:

Метод	Описание
`Node getNode()`	Возвращает начальный узел данного `DOMSource`
`String getSystemId()`	Возвращает системный идентификатор для разрешения относительных URI
`void setNode(Node)`	Устанавливает начальный узел данного `DOMSource`
`setSystemId(String)`	Устанавливает системный идентификатор для разрешения относительных URI

javax.xml.transform.ErrorListener

`ErrorListener` является интерфейсом; если вы хотите самостоятельно обрабатывать ошибки, то можно написать класс, реализующий этот интерфейс, и указать его в методах `setErrorListener()` классов `TransformerFactory` или `Transformer`. Объект `ErrorListener` будет получать сообщения как об ошибках компиляции, так и ошибках исполнения.

Этот класс основан на SAX-интерфейсе `ErrorHandler`, он делит ошибки на три категории: предупреждения, ошибки и фатальные ошибки. После предупреждения процесс преобразования может быть доведен до успешного завершения; после ошибки процессор может продолжить работу, но лишь для того, чтобы обнаружить дальнейшие ошибки и, в конце концов, аварийно завершить работу; и, наконец, после фатальной ошибки процессор немедленно прекращает работу.

Каждый метод может сгенерировать исключение, которое приведет к немедленному прерыванию обработки, даже тогда, когда процессор мог бы продолжить свою работу.

Если никакого объекта `ErrorListener` не указано, то ошибки передаются на стандартный вывод `System.err`.

Сами методы приведены ниже. Аббревиатура «[TE]» после каждого метода означает, что каждый из них может сгенерировать исключение TransformerException, чтобы завершить обработку.

Метод	Описание
void error(TransformerException) [TE]	Обрабатывает ошибку
void fatalError(TransformerException) [TE]	Обрабатывает фатальную ошибку
void warning(TransformerException) [TE]	Обрабатывает предупреждение

javax.xml.transform.OutputKeys

В этом классе определяется набор строковых констант, используемых для представления свойств стандартного вывода, которые задаются в элементах <xsl:output> и <xsl:document>.

Вот названия этих констант:

```
CDATA_SECTION_ELEMENTS
DOCTYPE_PUBLIC
DOCTYPE_SYSTEM
ENCODING
INDENT
MEDIA_TYPE
METHOD
OMIT_XML_DECLARATION
STANDALONE
VERSION
```

Они очевидным образом соответствуют атрибутам элемента <xsl:output>.

Эти константы полезны при вызове таких методов, как getOutputProperty() и setOutputProperty() объекта Transformer.

javax.xml.transform.Result

Result является интерфейсом; он представляет собой абстракцию трех классов SAXResult, DOMResult и StreamResult, которые суть различные способы представления вывода XML-данных. Данный интерфейс позволяет передавать экземпляры этих классов во второй аргумент метода Transformer.transform().

В этом классе определены две статических константы, PI_DISABLE_OUTPUT_ESCAPING и PI_ENABLE_OUTPUT_ESCAPING, имена которых соответствуют инструкциям обработки, сформированным в результате установки опции «disable-output-escaping="yes"» в инструкциях <xsl:text> или <xsl:value-of> таблицы стилей.

Данный интерфейс определяет два метода, позволяющих иметь системный идентификатор любому типу экземпляра Result:

Метод	Описание
`String getSystemId()`	Возвращает системный идентификатор
`void setSystemId()`	Устанавливает системный идентификатор

javax.xml.transform.sax.SAXResult

Указание объекта `SAXResult` в качестве вывода для преобразования приводит к тому, что конечное дерево преобразования поступает в переданный пользователем объект SAX2 `ContentHandler` в виде последовательности событий SAX так, как если бы они исходили от самого XML-анализатора. Объект `SAXResult` содержит ссылку на этот `ContentHandler`, а также строку, в которой содержится системный идентификатор. (Упомянутый системный идентификатор можно сделать доступным в коде `ContentHandler` как часть объекта `Locator`, хотя в спецификации нет четкого указания на то, что так должно быть.)

Многие процессоры XSLT будут поддерживать в будущем вывод в форме SAX, но это не обязательно. Если процессор поддерживает такой вывод, то метод `getFeature(SAXResult.FEATURE)` возвратит истину.

Существует несколько потенциальных трудностей, возникающих при передаче выходных данных XSLT в `ContentHandler`:

- Что происходит с опцией `disable-output-escaping`? Спецификация TrAX решает этот вопрос, говоря, что любой вывод текста с использованием опции `disable-output-escaping="yes"` будет предваряться инструкцией обработки, название которой соответствует значению константы `Result.PI_DISABLE_OUTPUT_ESCAPING`, а за ним будет следовать инструкция, название которой соответствует значению константы `Result.PI_ENABLE_OUTPUT_ESCAPING`.
- Что происходит с комментариями в конечном дереве? Спецификация TrAX разрешает эту проблему, позволяя хранить в объекте `SAXResult` как `LexicalHandler`, так и `ContentHandler`. Поэтому комментарии могут быть переданы в `LexicalHandler`.
- Что происходит, если конечное дерево не является правильно построенным документом? SAX-интерфейс `ContentHandler` предназначен для получения потока событий, относящихся к правильно построенному документу, и во многих случаях передача чего-нибудь другого в `ContentHandler` приведет к ошибке. Но, как мы видели в главе 2, результат XSLT-преобразования должен быть только сбалансированным. К сожалению, в спецификации TrAX нет ответа на этот вопрос. (Решение, реализованное в Saxon, состоит в том, чтобы посылать обработчику содержимого особую инструкцию обработки, указывающую на то, что вывод будет обрезан, поскольку он не является правильно построенным. И только в том случае, если `ContentHandler` **отвергнет** эту инструкцию обработки, сгенерировав исключение `SAXException`, будут посланы последующие события.

У данного класса имеется два конструктора:

```
SAXResult()
SAXResult(org.xml.sax.ContentHandler)
```

и следующие методы:

Метод	Описание
`org.xml.sax.ContentHandler getHandler()`	Возвращает `ContentHandler`
`org.xml.sax.ext.LexicalHandler getLexicalHandler()`	Возвращает `LexicalHandler`
`String getSystemId()`	Возвращает системный идентификатор
`void setHandler(org.xml.sax.ContentHandler)`	Устанавливает `ContentHandler`, чтобы получать события, относящиеся к конечному дереву
`void setHandler(org.xml.sax.ext.LexicalHandler)`	Устанавливает `LexicalHandler`, чтобы получать лексические события (особенно комментарии), относящиеся к конечному дереву
`void setSystemId()`	Устанавливает системный идентификатор

javax.xml.transform.sax.SAXSource

`SAXSource` является подклассом `Source`, поэтому экземпляр этого класса можно передавать в метод `Transformer.transform()` (когда он соответствует исходному XML-документу) или в метод `TransformerFactory.newTemplates()` (когда он соответствует таблице стилей).

В сущности, объект `SAXSource` является комбинацией SAX-анализатора (`XMLReader`) и объекта SAX `InputSource`, который может быть как идентификатором URL, так и входным потоком двоичных или символьных данных. Из объекта `SAXSource` исходный документ выходит в виде потока событий SAX. Обычно это достигается за счет анализа XML-данных, хранящихся в файле или памяти; однако, определяя свои собственные реализации `XMLReader` и/или класса `InputSource`, можно получить события SAX из любого источника, например, можно сформировать их из запроса SQL или из результатов поиска в каталоге LDAP.

Если никакого объекта `XMLReader` не указано, то система использует анализатор по умолчанию. Выбор последнего может быть сделан с применением правил для класса `javax.xml.parsers.SAXParserFactory`, описанного выше в данном приложении, хотя это и не гарантируется.

Не в каждом процессоре XSLT поддерживается ввод данных в форме SAX. Если же такой ввод поддерживается, то метод `getFeature(SAXSource.FEATURE)` возвратит истину.

У данного класса есть три конструктора:

```
SAXSource()
SAXSource(InputSource)
SAXSource(XMLReader, InputSource)
```

а также следующие методы:

Метод	Описание
`InputSource getInputSource()`	Возвращает SAX `InputSource`.
`String getSystemId()`	Возвращает системный идентификатор, применяемый для разрешения относительных URI.
`XMLReader getXMLReader()`	Возвращает `XMLReader` (анализатор), если таковой был установлен.
`void setInputSource(org.xml.sax.InputSource)`	Устанавливает SAX `InputSource`.
`void setSystemId(String)`	Устанавливает системный идентификатор, применяемый для разрешения относительных URI.
`void setXMLReader(XMLReader)`	Устанавливает применяемый `XMLReader` (анализатор).
`static org.xml.sax.InputSource sourceToInputSource(Source source)`	Этот статический метод пытается создать SAX `InputSource` из объекта TrAX `Source` любого типа. Если это невозможно, метод возвращает null.

javax.xml.transform.sax.SAXTransformerFactory

Данный класс является подклассом `TransformerFactory`, который предоставляет три дополнительных возможности:

- Возможность формировать объект SAX `ContentHandler` (называемый `TemplatesHandler`), который будет принимать поток событий SAX, соответствующих таблице стилей, и по завершении возвращать объект `Templates` для этой таблицы стилей.
- Возможность формировать объект SAX `ContentHandler` (называемый `TemplatesHandler`), который будет принимать поток событий SAX, соответствующих таблице стилей, и по завершении автоматически применять данную таблицу стилей к исходному документу.
- Возможность формировать объект SAX `XMLFilter`, основанный на конкретной таблице стилей: этот `XMLFilter` осуществляет то же самое преобразование из SAX в SAX, что и эквивалентный объект `Transformer`, но с применением интерфейсов, определенных для `XMLFilter`. Таким образом, появляется возможность вставить данный фильтр преобразования в конвейер фильтров.

Далее приводятся дополнительные возможности, которые являются необязательными, так что не каждый процессор TrAX поддерживает их.

- Если `getFeature(SAXTransformerFactory.FEATURE)` возвращает истину, то данная реализация `TransformerFactory` является `SAXTransformerFactory`.
- Если `getFeature(SAXTransformerFactory.FEATURE_XMLFILTER)` возвращает истину, то можно использовать оба метода `newXMLFilter()`.

Если доступен `SAXTransformerFactory`, то объект именно этого класса будет возвращаться в результате вызова `TransformerFactory.newInstance()`.

Помимо методов класса TransformerFactory в данном методе имеются следующие методы:

Метод	Описание
TemplatesHandler newTemplatesHandler() [TCE]	Создает и возвращает TemplatesHandler. Объект TemplatesHandler может быть передан вместе с потоком событий SAX, соответствующих содержимому таблицы стилей.
TransformerHandler newTransformerHandler() [TCE]	Создает и возвращает TransformerHandler. Объект TransformerHandler будет выполнять идентичное преобразование исходного документа XML, который передается в виде потока событий SAX.
TransformerHandler newTransformerHandler (Source) [TCE]	Создает и возвращает TransformerHandler. Аргумент Source указывает на документ, содержащий таблицу стилей. Объект TransformerHandler будет выполнять преобразование, определяемое этой таблицей стилей, к исходному документу XML, который передается в виде потока событий SAX.
TransformerHandler newTransformerHandler (Templates) [TCE]	Создает и возвращает TransformerHandler. Аргумент Templates указывает на скомпилированную таблицу стилей. Объект TransformerHandler будет выполнять преобразование, определяемое этой таблицей стилей, к исходному документу XML, который передается в виде потока событий SAX.
org.sax.xml.XMLFilter newXMLFilter(Source) [TCE]	Создает и возвращает XMLFilter. Аргумент Source указывает на документ, содержащий таблицу стилей. Полученный XMLFilter будет выполнять преобразование, определяемое этой таблицей стилей.
org.sax.xml.XMLFilter newXMLFilter(Templates) [TCE]	Создает и возвращает XMLFilter. Аргумент Templates указывает на скомпилированную таблицу стилей. Полученный XMLFilter будет выполнять преобразование, определяемое этой таблицей стилей.

javax.xml.transform.Source

Source является интерфейсом; он существует как абстракция трех классов: SAXSource, DOMSource и StreamSource, которые соответствуют различным способам представления XML-документа. С его помощью можно использовать любой объект любого из перечисленных типов в качестве исходного документа для метода Transformer.transform() или в качестве таблицы стилей для метода TransformerFactory.newTemplates().

Данный интерфейс определяет два метода, которые позволяют объекту Source иметь системный идентификатор:

Метод	Описание
String getSystemId()	Возвращает системный идентификатор
void setSystemId()	Устанавливает системный идентификатор

javax.xml.transform.SourceLocator

SourceLocator – это интерфейс, основанный на базе интерфейса SAX Locator. Объект SourceLocator применяется для указания места в таблице стилей, где произошла ошибка. Обычно экземпляр SourceLocator формируется XSLT-процессором, а доступ к нему приложения осуществляется с помощью метода TransformerException.getLocator().

Ниже приведены методы интерфейса:

Метод	Описание
int getColumnNumber()	Возвращает номер столбца с ошибкой, если известно ее местонахождение, или -1 в противном случае
int getLineNumber()	Возвращает номер строки с ошибкой, если известно ее местонахождение, или -1 в противном случае
String getPublicId()	Возвращает открытый идентификатор документа, если он доступен, и null в противном случае
String getSystemId()	Возвращает системный идентификатор документа, если он доступен, и null в противном случае

javax.xml.transform.stream.StreamSource

Класс StreamSource представляет входные данные XML в виде потока символов или байтов. Он сделан по образцу класса SAX InputSource; единственная причина, по которой необходимо существование класса StreamSource, это то, что в классе InputSource не реализован интерфейс Source, то есть экземпляр InputSource не может быть непосредственно передан в методы типа Transformer.transform(Source, Result).

Большинство XSLT-процессоров будут поддерживать ввод в форме потока, тем не менее, это не обязательно. Если процессор поддерживает ввод в такой форме, то метод getFeature(StreamSource.FEATURE) возвратит истину.

Если ввод производится из потока байтов (InputStream) или символов (Reader), то будет правильным вызвать метод setSystemId() для указания URI документа, чтобы можно было разрешать относительные URI (например, те, которые используются в функции document()). Сам поток не содержит этой информации, поэтому ее нужно передавать извне.

Ниже приведены конструкторы класса:

```
StreamSource()
StreamSource(java.io.File)
StreamSource(java.io.InputStream)
StreamSource(java.io.InputStream, String)
StreamSource(java.io.Reader)
StreamSource(java.io.Reader, String)
StreamSource(String)
```

В каждом случае действие конструктора будет таким же, как если бы использовался конструктор по умолчанию, после которого применяется соответствующий метод `setXXX()`. Аргумент `String` во всех случаях является системным идентификатором документа.

Далее следуют методы класса:

Метод	Описание
`java.io.InputStream getInputStream()`	Возвращает переданный объект `InputStream`
`String getPublicId()`	Возвращает установленный открытый идентификатор
`java.io.Reader getReader()`	Возвращает переданный объект `Reader`
`String getSystemId()`	Возвращает системный идентификатор
`void setInputStream(java.io.InputStream)`	Передает объект `InputStream`
`void setPublicId(String)`	Устанавливает открытый идентификатор
`void setReader(java.io.Reader)`	Передает объект `Reader`
`void setSystemId(java.io.File)`	Передает файл `File`, из которого можно извлечь системный идентификатор
`void setSystemId(String)`	Устанавливает системный идентификатор (URL)

javax.xml.transform.stream.StreamResult

Если требуется получить последовательный вывод конечного дерева, то нужно в качестве результата преобразования указать объект класса `StreamResult`. Форматом конечного файла может быть XML, HTML или простой текст, в зависимости от метода вывода, определенного при помощи элемента `<xsl:output>`, или методов `setOutputProperty()` и `setOutputProperties()` класса `Transformer`.

Класс `StreamResult` определяется по аналогии с классом `StreamSource`, который, в свою очередь, основан на классе SAX `InputSource`. Экземпляр `StreamResult` может быть либо файлом (заданным при помощи URL или Java-объекта `File`), либо символьным потоком (`Writer`), либо потоком байтов (`OutputStream`).

Большинство XSLT-процессоров будут поддерживать потоковый вывод, но это не обязательно. Если же процессор поддерживает такой формат вывода, то метод `getFeature(StreamResult.FEATURE)` возвратит истину.

Хотя место для вывода результата преобразования можно задать с помощью URI, это должно быть место назначения с возможностью записи. На практике это, как правило, означает, что используется URI с префиксом `«file:»`, но потенциально вывод может осуществляться по FTP или WebDAV[1]. Если про-

[1] Протокол для распределенной разработки веб-сайта и контроля версий (Web Distributed Authoring and Versioning, WebDAV). – *Примеч. перев.*

цессор запускается из апплета, то вывод в файл, вероятно, невозможен. В спецификации не указывается, что произойдет, если задан относительный URI; но, поскольку относительные URI нельзя использовать с классом SAX `InputSource`, на основе которого образован данный класс, то лучше избегать их применения. Некоторые процессоры могут рассматривать относительные URI по отношению к текущей директории.

В классе предусмотрены конструкторы для каждого из возможных форматов вывода. Конструктор, принимающий строку `String`, ожидает получить в качестве аргумента системный идентификатор (URL).

```
StreamResult()
StreamResult(File)
StreamResult(java.io.OutputStream)
StreamResult(String)
StreamResult(java.io.Writer)
```

Далее следуют методы класса:

Метод	Описание
`java.io.OutputStream getOutputStream()`	Возвращает двоичный поток выходных данных
`String getSystemId()`	Возвращает системный идентификатор
`java.io.Writer getWriter()`	Возвращает объект `Writer` (выходной текстовый поток данных)
`void setOutputStream(java.io.OutputStream)`	Устанавливает двоичный поток выходных данных
`void setSystemId(java.io.File)`	Устанавливает вывод в указанный файл, устанавливая системный идентификатор равным URL этого файла
`void setSystemId(String)`	Устанавливает системный идентификатор, переданный в виде URL
`void setWriter(java.io.Writer)`	Указывает объект `Writer` (выходной текстовый поток данных), в который передается вывод

javax.xml.transform.Templates

Объект `Templates` представляет собой скомпилированную таблицу стилей. Скомпилированные таблицы стилей не могут быть сохранены на диск, тем не менее, они хранятся в памяти и могут быть использованы столько раз, сколько потребуется. Чтобы использовать объект `Templates` для выполнения преобразования, нужно сначала создать объект `Transformer`, вызвав метод `newTransformer()`, а затем, настроив нужным образом этот объект (например, установив значения параметров и выходных свойств), запустить преобразование с помощью метода `Transformer.transform()`.

Ниже приводятся методы объекта Templates:

Метод	Описание
java.util.Properties getOutputProperties()	Возвращает объект Properties, соответствующий именам и значениям выходных свойств, определенных в таблице стилей с помощью элементов <xsl:output>. Ключи этих свойств будут строками, определенными в классе OutputKeys; значения определяются значениями, указанными в таблице стилей. Заметьте, что выходные свойства, которые определены динамически, этим методом не возвращаются: например, если атрибут метода вывода опущен в элементе <xsl:output>, то во время компиляции система не знает, в каком формате будут представлены выходные данные: XML или HTML.
Transformer newTransformer() [TCE]	Создает объект Transformer, который может использоваться для воздействия на преобразование, определяемое таблицей стилей.

javax.xml.transform.sax.TemplatesHandler

Класс TemplatesHandler – это тот же SAX ContentHandler, в котором поток событий SAX рассматривается как содержимое таблицы стилей. Когда документ передается целиком, то таблица стилей компилируется, и результат можно получить с помощью метода getTemplates().

Данный способ является альтернативой вызову метода TransformerFactory.newTemplates() и передачи в качестве источника таблицы стилей объекта SAXSource. Применение TemplatesHandler становится актуальным, когда события SAX исходят не от объекта SAX XMLReader; например, когда таблица стилей является результатом другого преобразования, тогда источником SAX-событий является TrAX Transformer. В подобной ситуации объект TemplatesHandler может быть «спрятан» внутрь объекта SAXResult и передан в качестве объекта Result в предыдущее преобразование.

Объект TemplatesHandler всегда создается при помощи метода newTemplatesHandler(), принадлежащего классу SAXTransformerFactory. Помимо методов, определенных в интерфейсе SAX ContentHandler, данный класс предлагает следующие методы:

Метод	Описание
String getSystemId()	Возвращает системный идентификатор таблицы стилей.
Templates getTemplates()	Возвращает объект Templates, полученный в результате компиляции указанного документа в качестве таблицы стилей.
void setSystemId()	Устанавливает системный идентификатор таблицы стилей. Системный идентификатор требуется для разрешения относительных URI (например, в элементах <xsl:include> или <xsl:import>).

javax.xml.transform.TransformerFactoryConfigurationError

Класс TransformerFactoryConfigurationError (те, кто писал спецификацию, должно быть, прекрасно владели машинописью) представляет ошибку в конфигурации процессора XSLT, в отличие от ошибки в самой таблице стилей.

Заметьте, что это скорее ошибка (Error), нежели исключение (Exception), то есть не ожидается, что приложение предпримет какие-либо действия по восстановлению.

Вот все имеющиеся методы:

Метод	Описание
String getMessage()	Возвращает сообщение об ошибке
Exception getException()	Возвращает любое вложенное исключение

javax.xml.transform.Transformer

В объекте Transformer собраны ресурсы, требующиеся для осуществления преобразования из объекта Source в объект Result. Сюда входят скомпилированная таблица стилей, значения параметров и выходных свойств, а также такие детали, как используемые объекты ErrorListener и URIResolver.

Интерфейс этого класса аналогичен интерфейсу класса IXSLProcessor из MSXML3 API от Microsoft.

Объект Transformer всегда создается при помощи вызова метода newTransformer() объекта Templates или TransformerFactory.

Объект Transformer может быть использован для выполнения более одного преобразования, но он не поддерживает многопоточность: не следует запускать следующее преобразование, пока не завершилось предыдущее. При необходимости осуществлять преобразования в параллельном режиме следует создавать несколько объектов Transformer из одного объекта Templates.

Главным методом класса является transform(), который принимает два аргумента, соответствующих исходному (Source) и конечному (Result) документам. Определено несколько типов объектов Source и несколько типов объекта Result. Все они описываются в данном приложении.

Ниже приводится полный набор методов класса. Коды исключений [IAE] и [TE] относятся соответственно к IllegalArgumentException и TransformerException.

Метод	Описание
void clearParameters()	Очищает весь набор параметров с использованием setParameter().
ErrorListener getErrorListener()	Возвращает ErrorListener для данного преобразования.

Метод	Описание
`java.util.Properties getOutputProperties() {IAE}`	Возвращает выходные свойства, определенные для данного преобразования. Сюда входят свойства, определенные в таблице стилей, а также с помощью методов `setOutputProperty()` и `setOutputProperties()`.
`String getOutputProperty(String) [IAE]`	Возвращает указанное выходное свойство, определенное для данного преобразования. В качестве аргумента должна передаваться одна из констант, определенных в классе `OutputKeys`, или свойство из конкретной реализации.
`Object getParameter(String)`	Возвращает значение параметра, определенного для данного преобразования.
`URIResolver getURIResolver()`	Возвращает объект `URIResolver`, применяемый в данном преобразовании, или пустое значение, если таковой не был указан.
`void setErrorListener(ErrorListener) [IAE]`	Указывает объект `ErrorListener`, применяемый для обработки ошибок, возникающих в ходе преобразования.
`void setOutputProperties(java.util.Properties) [IAE]`	Устанавливает выходные свойства для результата преобразования. Эти свойства переопределяют любые значения, установленные в таблице стилей с помощью `<xsl:output>`. Имена свойств – это, как правило, константы, определенные в классе `OutputKeys`, или имена свойств конкретной реализации, но это также могут быть свойства, определенные пользователями, при условии, что они используют корректные пространства имен. Для этого нужно применять обозначения вида `«{uri}локальное имя»`, так же, как и для параметров.
`void setOutputProperty(String, String) [IAE]`	Устанавливает указанное выходное свойство для результата данного преобразования.
`void setParameter(String, Object)`	Устанавливает параметр преобразования. Первый аргумент соответствует имени параметра, определенного в глобальном элементе таблицы стилей `<xsl:param>`; если данный параметр определяется с использованием пространств имен, то его следует записывать в форме `«{uri}локальное имя»`. Второй аргумент – значение параметра. Вероятнее всего (хотя и необязательно), процессоры будут использовать для преобразования объектов Java в значения XPath те же правила, что и для возвращаемых значений внешних функций Java. Эти правила описаны в главе 8.

Метод	Описание
`void setURIResolver( URIResolver)`	Указывает объект `URIResolver`, который применяется для разрешения относительных URI, встречающихся в преобразовании, особенно при вычислении функции `document()`.
`void transform( Source, Result) [TE]`	Осуществляет преобразование. Аргументы `Source` и `Result` являются интерфейсами, позволяющими указать объекты `Source` и `Result` различных типов.

javax.xml.transform.TransformerConfigurationException

Данный класс определяет ошибку компиляции, в общем случае находящуюся в таблице стилей. Этот класс является подклассом от `TransformerException` и имеет те же методы, что и родительский класс.

Для этого класса определено несколько конструкторов, но по той причине, что объект данного класса, как правило, создается самим XSLT-процессором, они здесь не приводятся.

javax.xml.transform.TransformerException

Объект `TransformerException` представляет собой исключение, которое может быть обнаружено как во время компиляции, так и во время выполнения. В нем может содержаться следующая информация:

- Сообщение, поясняющее ошибку.
- Вложенное исключение, как правило, содержащее дополнительную информацию об ошибке. (В действительности, несмотря на название метода `getException()`, возвращаемым значением не всегда будет объект класса `Exception`: это может быть любой объект класса `Throwable`, то есть как `Exception`, так и `Error`.)
- Объект `SourceLocator`, указывающий местонахождение ошибки в таблице стилей.
- Причина. Честно говоря, я не понимаю причины, побудившей разработчиков этого интерфейса разделить вложенное исключение и причину. Но, так или иначе, было сделано так, что информация о причине находится тут.

Для данного класса определено несколько конструкторов, но, по той причине, что объекты класса создаются самим XSLT-процессором, они здесь не приводятся.

Ниже перечислены методы класса:

Метод	Описание
`Throwable getCause()`	Возвращает причину исключения, если таковая имеется
`Throwable getException()`	Возвращает вложенное исключение, если таковое имеется

Метод	Описание
`String getLocationAsString()`	Формирует строку, описывающую местонахождение ошибки
`SourceLocator getLocator()`	Возвращает объект `SourceLocator`, если таковой имеется, идентифицирующий местонахождение произошедшей ошибки
`String getMessageAndLocation()`	Формирует строку, в которой приводится информация об ошибке и ее местонахождении
`void initCause(Throwable)`	Устанавливает причину исключения
`void setLocator(SourceLocator)`	Устанавливает объект `SourceLocator`, идентифицирующий местонахождение ошибки

Методы `printStackTrace()`, унаследованные от стандартного класса `Exception`, переопределены таким образом, что они показывают не только стек, соответствующий произошедшей ошибке `TransformerException`, но также и стек, соответствующий любому вложенному исключению `Exception`.

javax.xml.transform.TransformerFactory

Подобно фабрикам `SAXParserFactory` и `DocumentBuilderFactory` из модуля `javax.xml.parsers`, описанного в первой части этого приложения, данная фабрика классов позволяет выбирать реализацию XSLT от конкретного поставщика.

Первое, что нужно сделать в приложении – это получить доступ к объекту `TransformerFactory`, вызвав статический метод `TransformerFactory.newInstance()`. Различные поставщики XSLT-процессоров реализуют свои собственные подклассы `TransformerFactory`, и вызов данного метода определяет, процессором какого поставщика будет пользоваться ваше приложение. Если имеется несколько доступных процессоров, то выбор осуществляется на основании следующих действий:

- Используется значение свойства `javax.xml.parsers.TransformerFactory`, если к нему есть доступ. Как правило, системные свойства устанавливаются с помощью опции `-D` в командной строке java или с помощью вызова `System.setProperty()` из приложения.
- Просматривается файл свойств `$JAVA_HOME/lib/jaxp.properties` и внутри этого файла ищется значение свойства с именем `javax.xml.parsers.TransformerFactory`.
- Используются сервисные API, являющиеся частью спецификации JAR.

Скорее всего, в процессе установки конкретного XSLT-процессора файл, находящийся внутри архива `.jar`, устанавливает данный процессор в качестве используемого по умолчанию. Поэтому, если ничего не предпринять для выбора, то автоматически выбранный процессор будет зависеть от порядка файлов и каталогов в переменной окружения `CLASSPATH`.

После того как получен доступ к TransformerFactory, можно применить ряд методов для его конфигурации. И, наконец, можно вызвать метод newTemplates() для компиляции таблицы стилей или метод newTransformer() для непосредственного получения объекта Transformer (если скомпилированная таблица будет использоваться только один раз) экземпляра SAXParser.

Ниже приводятся доступные методы. Как и в случае других методов, генерируемые исключения обозначены аббревиатурами, значения которых объясняются в таблице на стр. 914.

Метод	Описание
Source getAssociatedStyleSheet(Source, String media, String title, String charset) [TCE]	Ищет внутри исходного XML-документа, указанного с помощью аргумента Source, инструкцию обработки <?xml-stylesheet?> с атрибутами, соответствующими параметрам media, title и charset (любой из которых может быть значением null), и возвращает объект Source, соответствующий таблице стилей.
Object getAttribute(String) [IAE]	Возвращает свойство конфигурации, зависящее от поставщика.
ErrorListener getErrorListener()	Возвращает объект ErrorListener, который используется в преобразовании по умолчанию. Если таковой не был указан, то используется ErrorListener поставщика.
boolean getFeature(String)	Возвращает информацию о возможностях, поддерживаемых в данной реализации. Возможности определяются константами из других классов, например, getFeature(SAXResult.FEATURE) возвращает true, если процессор поддерживает вывод в SAX ContentHandler.
URIResolver getURIResolver()	Возвращает URIResolver, который используется для преобразований по умолчанию.
static TransformerFactory newInstance() [TFCE]	Возвращает экземпляр TransformerFactory в конкретной реализации, выбранной в соответствии с вышеуказанными правилами.
Templates newTemplates(Source) [TCE]	Компилирует таблицу стилей, переданную в объекте Source, возвращая объект Templates, являющийся представлением скомпилированной таблицы стилей.
Transformer newTransformer() [TCE]	Создает объект Transformer, который осуществляет идентичное преобразование.
Transformer newTransformer(Source) [TCE]	Метод, эквивалентный вызову newTemplates(Source).newTransformer().
void setAttribute(String, Object) [IAE]	Указывает свойство конфигурации, характерное для поставщика.

Метод	Описание
`void setErrorListener(ErrorListener) [IAE]`	Определяет объект `ErrorListener` для обработки ошибок.
`void setURIResolver(URIResolver)`	Определяет `URIResolver` для разрешения URI, содержащихся в таблице стилей или исходном документе.

javax.xml.transform.sax.TransformerHandler

Объект `TransformerHandler` получает события SAX, соответствующие исходному документу. Он выполняет преобразование исходного документа и записывает результат в переданный объект `Result`.

Интерфейс `TransformerHandler` расширяет три интерфейса, обрабатывающие события SAX, а именно `ContentHandler`, `LexicalHandler` и `DTDHandler`. Он должен действовать как `LexicalHandler`, чтобы можно было обрабатывать комментарии в исходном документе, в то же время он должен действовать как `DTDHandler`, для того чтобы можно было игнорировать комментарии в DTD, а также получить полную информацию об объявлениях в DTD неанализируемых сущностей.

Применение объекта `TransformerHandler` является альтернативой созданию объекта `Transformer` и использованию `SAXSource` для определения входного документа. Такой подход особенно полезен в том случае, когда источником событий SAX является объект, отличный от SAX `XMLReader`. Например, таким источником может быть другое преобразование TrAX, или же другая программа, которая позволяет объекту `ContentHandler` получать результаты преобразования.

Экземпляр `TransformerHandler` всегда создается при помощи метода `newTransformerHandler()` фабрики `SAXTransformerFactory`.

Помимо методов, определенных в SAX-интерфейсах `ContentHandler`, `LexicalHandler` и `DTDHandler`, класс `TransformerHandler` предоставляет следующие методы:

Метод	Описание
`String getSystemId()`	Возвращает системный идентификатор, определенный для исходного документа.
`Transformer getTransformer()`	Возвращает соответствующий объект `Transformer`, который может быть использован для установки значений параметров и выходных свойств преобразования.
`void setResult(Result) [IAE]`	Устанавливает объект, в который будет передан результат преобразования.
`void setSystemId(String)`	Устанавливает системный идентификатор для исходного документа. Он может использоваться в качестве базового URI для разрешения относительных URI, имеющихся в документе.

javax.xml.transform.URIResolver

URIResolver является интерфейсом: можно создать класс, реализующий этот интерфейс, и передать объект этого класса в метод setURIResolver() классов TransformerFactory или Transformer. Когда процессору XSLT нужно найти XML-документ с помощью URI, указанного в элементах <xsl:include>, <xsl:import> или в функции document(), он передает эту работу объекту URIResolver. Объект URIResolver может рассматривать URI удобным ему способом, а затем возвращает затребованный документ в виде объекта Source: как правило, это SAXSource, DOMSource или StreamSource.

Если, к примеру, в таблице стилей происходит следующий вызов: «document('db:employee=517541')», то объект URIResolver может интерпретировать данный URI как запрос к базе данных и вернуть XML-документ, представляющий собой набор записей.

В данном интерфейсе определяется только один метод:

Метод	Описание
Source resolve(String, String) [TE]	В первом аргументе содержится относительный URI, а во втором базовый, по отношению к которому будет рассматриваться относительный URI. Метод возвращает объект Source, содержащий затребованный документ; если документ не удалось найти, то генерируется исключение TransformerException. Метод может также вернуть значение null, указывая, что нужно использовать URIResolver, определенный по умолчанию.

Примечание: существует практическая проблема, которую не решает интерфейс TrAX, а именно: в XSLT есть правило, которое гласит, что если функция document() вызывается два раза для извлечения одного и того же абсолютного URI, то каждый раз должен возвращаться тот же самый документ. Поскольку в классе URIResolver относительный и базовый URI рассматриваются как два отдельных аргумента, а абсолютный URI не возвращается, то не совсем понятно, на ком лежит ответственность за обеспечение вышеуказанного правила: на процессоре или на объекте URIResolver, особенно в тех случаях, когда URI передается в таком формате, который не распознается XSLT-процессором, как в приведенном выше примере.

Заметьте, что аргументы передаются в обратном порядке по сравнению с Java-классом java.net.URL.

Примеры

В этом разделе приводится несколько простых примеров приложений, различным образом использующих API TrAX для управления преобразованием.

Пример 1: Преобразование с использованием файлов

В этом примере (FileTransform.java) выполняется единичное преобразование; исходный документ и таблица стилей берутся из файлов, а результат записывается в другой файл.

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.File;
public class FileTransform {
    public static void main(String[] args) throws Exception {
        File source = new File(args[0]);
        File style = new File(args[1]);
        File out = new File(args[2]);

        TransformerFactory factory = TransformerFactory.newInstance();
        Transformer t = factory.newTransformer(new StreamSource(style));
        t.transform(new StreamSource(source), new StreamResult(out));
    }
}
```

Так выглядит минимальное приложение, использующее TrAX.

Примеры на языке Java, приведенные в этом разделе, можно загрузить с веб-сайта издательства Wrox в виде исходных кодов. Это сделано специально, поскольку вы научитесь большему, если будете сами компилировать программы. В частности, вам потребуется XSLT-процессор, поддерживающий интерфейс TrAX, например Saxon или Xalan. Переменная окружения CLASSPATH должна быть установлена нужным образом. Если у вас что-то установлено неправильно, то сообщения об ошибках, которые вы получите во время компиляции, будут гораздо яснее, нежели сообщения об ошибках, полученные во время запуска.

Полагая, что у вас установлен пакет Sun Java JDK, вы можете скомпилировать данную программу с помощью следующей команды:

```
javac  FileTransform.java
```

Здесь предполагается, что текущим каталогом является тот, который содержит файл с исходным кодом. Скомпилировав код, вы можете запустить его из командной строки следующим образом:

```
java  FileTransform  source.xml  style.xsl  out.html
```

Безусловно, данное приложение не является профессионально написанным. Если указаны некорректные аргументы, если входные файлы не существуют или в ходе преобразования произошла ошибка, то произойдет сбой приложения с выводом стека вызовов. Но это только начало. Данный пример призван продемонстрировать, как работают основные классы TrAX, а не научить вас профессиональному программированию на Java.

Поскольку некоторые предпочитают набирать примеры так, как они написаны в книге, то в каталоге с Java-приложениями также содержатся образцы файлов XML и XSL, их имена `source.xml` и `style.xsl`. Поэтому, если этот каталог является вашей текущей директорией, то вы можете ввести приведенную выше команду именно в таком виде. Но, безусловно, это Java-приложение обработает любой исходный файл и таблицу стилей.

> *В своих примерах я указывал в качестве аргументов относительные URI, такие как* `source.xml`. *Во многих примерах из спецификации JAXP 1.1 делается то же самое, и как Saxon, так и Xalan принимают такие аргументы. Но, по всей вероятности, это будет работать только в тех случаях, когда анализатор SAX рассматривает относительный URI как системный идентификатор, передаваемый объекту* `InputSource`, *несмотря на тот факт, что в спецификации SAX явно утверждается обязательность передачи абсолютного URI.*

Пример 2: Передача параметров и выходных свойств

Данное приложение, `Parameters.java`, является усовершенствованием предыдущего примера:

- Оно позволяет передавать из командной строки параметр таблицы стилей
- В нем изменяются выходные свойства, определенные в таблице стилей
- Выход направляется в `System.out` вместо файла.

Метод `main()` этого улучшенного приложения выглядит следующим образом:

```
public static void main(String[] args) throws Exception {
    File source = new File(args[0]);
    File style = new File(args[1]);
    String title = args[2];

    TransformerFactory factory = TransformerFactory.newInstance();
    Transformer t = factory.newTransformer(new StreamSource(style));
    t.setParameter("title", title);
    t.setOutputProperty(OutputKeys.INDENT, "no");
    t.transform(new StreamSource(source), new StreamResult(System.out));
}
```

Данная версия приложения может быть запущена с помощью приведенной ниже команды. В качестве третьего параметра командной строки вместо выходного файла теперь указывается значение параметра для таблицы стилей (поскольку это значение содержит пробелы, оно заключено в кавычки).

```
java Parameters source.xml style.xsl "New organization structure"
```

Сравните результат с предыдущим примером, и вы заметите, что HTML-код теперь не обрамляется отступами, а содержимое элементов `<title>` и `<h1>` изменилось. И, конечно, результат преобразования в данном примере поступает на консоль.

Пример 3: Хранение документов в памяти

В этом примере мы будем хранить в памяти как таблицу стилей, так и исходный документ, чтобы их можно было использовать неоднократно. В принципе, это могло бы нам позволить применить одну таблицу стилей к нескольким исходным документам, но для сохранения простоты примера мы будем использовать один и тот же исходный документ и таблицу стилей несколько раз, меняя только параметры преобразования. Для каждого из параметров, указанных в командной строке, мы выполним по одному преобразованию.

Для хранения в памяти исходного документа мы создадим его DOM-модель. Естественно было бы сделать это посредством использования класса `DocumentBuilder`, описанного в первой части данного приложения, но поскольку я предпочитаю использовать в этих примерах интерфейс TrAX, то мы поступим другим образом. TrAX легко позволяет выполнить идентичное преобразование и построить DOM-модель: все, что нам нужно сделать – это выполнить идентичное преобразование из последовательного файла в объект DOM.

Таблицу стилей мы могли бы также хранить в памяти как DOM, но это означало бы проверку на действительность и компиляцию таблицы стилей при каждом использовании. Будет лучше, если мы будем хранить скомпилированную таблицу стилей, то есть объект `Templates`.

Код для данного примера (`Repeat.java`) выглядит следующим образом:

```
public static void main(String[] args) throws Exception {
    File source = new File(args[0]);
    File style = new File(args[1]);

    TransformerFactory factory = TransformerFactory.newInstance();

    // формируем документ DOM с помощью идентичного
    // преобразования
    Transformer builder = factory.newTransformer();
    DOMResult result = new DOMResult();
    builder.transform(new StreamSource(source), result);
    Document doc = (Document)result.getNode();

    // Компилируем таблицу стилей
    Templates templates = factory.newTemplates(new StreamSource(style));

    // выполняем по одному преобразованию для каждого
    // переданного параметра
    for (int i=2; i<args.length; i++) {
        Transformer t = templates.newTransformer();
        System.out.println("======= TITLE = " + args[i] + "=======");
        t.setParameter("title", args[i]);
        t.transform(new StreamSource(source), new StreamResult(System.out));
    }
}
```

Запустить данное приложение можно с помощью следующей команды:

```
java  Repeat  source.xml  style.xsl  one  two  three  four
```

В результате запуска этой команды преобразование будет выполнено четыре раза, и в выходном HTML-коде название страницы будет по очереди принимать значения «one», «two», «three» и «four».

Приведенное приложение вряд ли является реалистичным, однако сам принцип хранения таблицы стилей и исходного документа в памяти может часто использоваться для достижения выигрыша в производительности, когда применяются сервлеты.

Пример 4: Использование инструкции обработки <?xml-stylesheet?>

Во всех предыдущих примерах исходный документ и таблица стилей указывались отдельно. Однако, как можно было видеть в главе 3, существует возможность указания предпочтительной таблицы стилей в начале исходного XML-файла с помощью инструкции обработки <?xml-stylesheet?>. В данном примере показано, как извлечь соответствующую таблицу стилей с помощью интерфейса TrAX API, а именно с помощью метода getAssociatedStylesheet(), имеющегося у объекта TransformerFactory.

Ниже приведен код метода main() для данного примера (Associated.java):

```
public static void main(String[] args) throws Exception {
    File input = new File(args[0]);
    StreamSource source = new StreamSource(input);

    TransformerFactory factory = TransformerFactory.newInstance();

    // Получаем таблицу стилей, связанную с исходным документом

    Source style =
           factory.getAssociatedStylesheet(source, null, null, null);

    // Используем ее в преобразовании


    Transformer t = factory.newTransformer(style);
    t.transform(source, new StreamResult(System.out));
}
```

Указав значения null для аргументов media (тип содержимого), title (название) и charset (набор символов) в методе getAssociatedStylesheet(), мы тем самым выбираем таблицу стилей, используемую по умолчанию. Если в документе указано несколько инструкций обработки <?xml-stylesheet?>, то указание этих аргументов позволяет выбрать конкретную таблицу стилей.

Запустить данный пример можно с помощью команды:

```
java  Associated  source.xml
```

Пример 5: Конвейер SAX

Зачастую бывает полезно поместить XSLT-преобразование внутрь конвейера SAX. Конвейер состоит из ряда каскадов, каждый из которых реализует ин-

терфейс SAX2 `XMLFilter`. Эти фильтры соединяются между собой так, что каждый из них подобен объекту SAX2 `ContentHandler` (получатель событий SAX) для следующего каскада. Некоторые из этих фильтров могут быть фильтрами XSLT, другие же могут быть написаны на Java или реализованы с помощью других средств.

В данном примере (`Pipeline.java`) используется трехкаскадный конвейер. На первом этапе написанный на Java фильтр переводит имена всех элементов в документе в верхний регистр, записывая первоначальное имя в атрибут. На втором этапе XSLT-преобразование копирует некоторые элементы без изменения, а другие удаляет на основании значения другого атрибута. И на конечной стадии другой XML-фильтр восстанавливает имена элементов к первоначальному виду.

Этот пример разработан в демонстрационных целях, но в нем присутствует и практический смысл. В XML различаются имена в верхнем и нижнем регистре, то есть имена `<b>` и `<B>` означают различные элементы. Но наследие HTML приводит к тому, что иногда вам нужно будет преобразовывать элементы `<b>` и `<B>` одинаковым образом. В XSLT этого добиться не так-то просто, поэтому мы делаем предварительную и конечную обработку с помощью Java.

Начнем с двух XML-фильтров, написанных на Java. Они реализованы как подклассы вспомогательного класса SAX `XMLFilterImpl`. Нам нужно только написать методы `startElement()` и `endElement()`; другие методы попросту пропускают через себя события без изменений.

Предварительный фильтр выглядит следующим образом. Он приводит имя элемента к нижнему регистру и сохраняет переданное локальное имя и полное имя как дополнительные атрибуты.

```
private class PreFilter extends XMLFilterImpl {

        public void startElement (String uri, String localName, String qName,
                    Attributes atts)
         throws SAXException {
             String newLocalName = localName.toLowerCase();
             String newQName = qName.toUpperCase();
             AttributesImpl newAtts =
                 (atts.getLength()>0 ?
                     new AttributesImpl(atts) :
                     new AttributesImpl());
             newAtts.addAttribute("", "old-local-name",
                        "old-local-name", "CDATA", localName);
             newAtts.addAttribute("", "old-qname",
                        "old-qname", "CDATA", qName);
             super.startElement(uri, newLocalName, newQName, newAtts);
         }

         public void endElement (String uri, String localName, String qName)
         throws SAXException {
             String newLocalName = localName.toLowerCase();
```

```
            String newQName = qName.toUpperCase();
            super.endElement(uri, newLocalName, newQName);
        }
    }
```

Конечный фильтр очень похож на предварительный фильтр; единственное отличие состоит в том, что в коде метода `startElement()` **первоначальные имена элементов (которые берутся из списка атрибутов) сохраняются в стеке, чтобы метод** `endElement` **мог достать их оттуда позже.**

```
    private class PostFilter extends XMLFilterImpl {

        public Stack stack;

        public void startDocument() throws SAXException {
            stack = new Stack();
            super.startDocument();
        }

        public void startElement (String uri, String localName, String qName,
                    Attributes atts)
         throws SAXException {

            String originalLocalName = localName;
            String originalQName = qName;
            AttributesImpl newAtts = new AttributesImpl();
            for (int i=0; i<atts.getLength(); i++) {
                String name = atts.getQName(i);
                String val = atts.getValue(i);
                if (name.equals("old-local-name")) {
                    originalLocalName = val;
                } else if (name.equals("old-qname")) {
                    originalQName = val;
                } else {
                    newAtts.addAttribute(
                                atts.getURI(i),
                                atts.getLocalName(i),
                                name,
                                atts.getType(i),
                                val);
                }
            }
            super.startElement(uri, originalLocalName, originalQName, newAtts);
            stack.push(originalLocalName);
            stack.push(originalQName);
        }

        public void endElement (String uri, String localName, String qName)
        throws SAXException {
            String originalQName = (String)stack.pop();
            String originalLocalName = (String)stack.pop();
            super.endElement(uri, originalLocalName, originalQName);
        }
    }
```

Теперь мы можем составить конвейер, который, в действительности, имеет пять стадий:

- Сам синтаксический анализатор XML, который мы можем получить с помощью механизма ParserFactory, описанного в начале данного приложения.
- Предварительный фильтр.
- Преобразование XSLT, осуществляемое с помощью таблицы стилей, хранящейся в файле filter.xsl.
- Конечный фильтр.
- Сериализатор, получаемый из TransformerFactory, который фактически является объектом TransformerHandler, выполняющим идентичное преобразование в форму StreamResult.

Как в любом конвейере SAX2, первая стадия – это объект XMLReader, последняя – ContentHandler, а каждая из промежуточных стадий является объектом XMLFilter. Каждая фаза связана с предыдущей методом setParent(), за исключением того, что объект ContentHandler связан с последним объектом XMLFilter с помощью вызова метода setContentHandler(). И, наконец, конвейер активизируется после применения метода parse() к последнему объекту XMLFilter, который в нашем случае является конечным фильтром.

Ниже приводится код, конструирующий конвейер и осуществляющий прохождение по нему исходного файла:

```
public void run(String input) throws Exception {
    StreamSource source = new StreamSource(new File(input));
    File style = new File("filter.xsl");

    TransformerFactory factory = TransformerFactory.newInstance();
    if (!factory.getFeature(SAXTransformerFactory.FEATURE_XMLFILTER)) {
        System.err.println("SAX Filters are not supported");
    } else {
        SAXTransformerFactory saxFactory = (SAXTransformerFactory)factory;
        XMLFilter pre = new PreFilter();
        // подставьте сюда выбранный вами анализатор SAX2
        // или используйте SAXParserFactory для его получения
        pre.setParent(new com.icl.saxon.aelfred.SAXDriver());
        XMLFilter filter = saxFactory.newXMLFilter(new StreamSource(style));
        filter.setParent(pre);
        XMLFilter post = new PostFilter();
        post.setParent(filter);
        TransformerHandler serializer = saxFactory.newTransformerHandler();
        serializer.setResult(new StreamResult(System.out));
        Transformer trans = serializer.getTransformer();
        trans.setOutputProperty(OutputKeys.METHOD, "xml");
        trans.setOutputProperty(OutputKeys.INDENT, "yes");
        post.setContentHandler(serializer);
        post.parse(source.getSystemId());
    }
}
```

Мы используем класс в нашем примере с помощью следующей тривиальной программы `main()`:

```
public static void main(String[] args) throws Exception {
    new Pipeline().run(args[0]);
}
```

Исполнить эту программу можно с помощью команды:

```
java  Pipeline  mixed-up.xml
```

Результаты посылаются на стандартный вывод.

Заключение

В этом приложении был дан обзор интерфейсов JAXP 1.1.

Ради полноты изложения мы начали с небольшого обзора возможностей JAXP по управлению анализаторами SAX и DOM, которые содержатся в модуле `javax.xml.parsers`.

Затем было приведено подробное описание классов и методов из модуля `javax.xml.transform` и его вспомогательных модулей, в совокупности образующих API для XML-преобразований или TrAX.

И в завершение были даны простые примеры, позволяющие показать реальные возможности TrAX в очень простых приложениях.

Выгода применения TrAX состоит в том, что он позволяет создавать Java-приложения, которые осуществляют XSLT-преобразования и при этом не привязаны к конкретному XSLT-процессору. К моменту написания книги интерфейсы TrAX уже поддерживались в продуктах Saxon и Xalan, хотя спецификация находилась еще в стадии рабочего проекта. Можно ожидать, что в будущем появятся и другие реализации.

Глоссарий

В этом глоссарии собраны некоторые из наиболее общих технических терминов, использованных в данной книге. Не все приведенные термины определяются в спецификациях XSLT и XPath – некоторые позаимствованы из спецификации XML и других смежных стандартов, а другие были введены в данной книге. Поэтому рядом с каждым термином приводится информация о месте его происхождения.

Однако сами определения сформулированы автором, и в некоторых случаях спецификации дают более формальные определения, хотя иногда они на удивление расплывчаты.

ID — XML

Атрибут типа ID имеет значение, уникальное в пределах документа (то есть отличное от других ID-атрибутов). Он является ID в силу того, что объявлен таковым в определении типа документа. Уникальность идентификатора ID гарантируется только в том случае, когда документ является действительным (valid) (для обработки с помощью XSLT действительность документа не обязательна). Доступ к элементам по их ID может быть осуществлен посредством функции `id()`. Атрибуты типа IDREF не имеют специального предназначения в XSLT.

URI — Стандарты Internet

Унифицированный идентификатор ресурса (Uniform Resource Identifier) – обобщение унифицированного указателя ресурса URL (Uniform Resource Locators), используемое для уникальной адресации таких ресурсов, как вебстраницы в Интернете.

URI пространства имен (Namespace URI) — XML Namespaces

URI, используемый для идентификации пространства имен. Такие URI отличаются от остальных тем, что им не соответствуют какие-то реальные ресурсы; URI – это просто уникальный идентификатор. На самом деле в качестве URI пространства имен может использоваться любая строка, однако URL типа «`http://`» используются чаще, предоставляя тем самым перспективу уникальности.

Атрибут (Attribute) XML

Пара имя=значение, присутствующая в открывающем теге элемента, например «category="grocery"».

Базовый URI (Base URI) XSLT

Для каждого узла есть связанный с ним базовый URI. Для элемента таким URI будет URI внешней XML-сущности, содержащей открывающий и закрывающий теги элемента (чаще всего, этой сущностью, конечно, является документ). Для узлов других типов базовый URI определяется по связанному узлу элемента, как правило, являющегося родителем. Можно также явно установить базовый URI при помощи атрибута xml:base. Базовый URI используется при раскрытии относительного URI, определенного в данном узле, например, относительный URI href рассматривается по отношению к URI родительского элемента.

Вложенная таблица стилей (Embedded Stylesheet) XSLT

Физическая таблица стилей, которая не представляет сама по себе XML-документа, но является вложенной внутрь некоторого большего документа XML (или не XML) в виде элемента <xsl:stylesheet>.

Временное дерево (Temporary Tree) Данная книга

Дерево, построенное в ходе применения таблицы стилей путем обработки непустого элемента <xsl:variable>. В спецификации XSLT 1.0 этот объект (и его тип) известен как фрагмент конечного дерева. В рабочем проекте XSLT 1.1 этот тип данных становится просто набором узлов, а самому объекту не присваивается никакого особого имени, поэтому мы придумали его сами.

Встроенное шаблонное правило (Built-in Template Rule) XSLT

Шаблонное правило, которое не определено явно в таблице стилей, однако может применяться неявно для обработки узла, если ему не соответствует никакого явно заданного правила.

Выражение (Expression) XPath

Конструкция языка XPath, вычисление которой дает строку, число, логическое значение, набор узлов или фрагмент конечного дерева. Используется во многих местах, например в атрибуте select элементов <xsl:for-each>, <xsl:value-of> и <xsl:variable>, а также в атрибуте test элементов <xsl:if> и <xsl:when>. Кроме того, выражения используются в фигурных скобках шаблонов значений атрибутов.

Выражение пути (Path Expression) XPath

Выражение пути – это выражение, с помощью которого выбирается набор узлов из исходного дерева. В нем определяются начальный набор узлов, с которого начинается выборка, а также последовательность шагов, которые

определяют путь следования от начальных узлов к последующим. Конечным результатом является набор узлов, полученный после выполнения по очереди всех шагов. Например, выражение пути «./parent::*» содержит в качестве начального набора узлов контекстный узел «.» и один шаг, который ведет к родительскому элементу этого узла.

Глобальная переменная (Global Variable) XSLT

Переменная, определенная в элементе <xsl:variable> верхнего уровня. Глобальная переменная доступна из любого места логической таблицы стилей, если не подменяется локальной переменной с тем же именем или другой глобальной переменной с тем же именем и более высоким приоритетом импортирования.

Дерево (Tree) XPath

Абстрактная структура данных, представляющая информационное содержимое XML-документа. Дерево всегда имеет один корень (который, в отличие от ботанического аналога всегда расположен сверху). Структура узлов дерева не обязана соответствовать правилам для корректных XML-документов, так, например, корень может иметь несколько дочерних элементов.

Документ (Document) XML

Анализируемая сущность, которая удовлетворяет синтаксису XML Document, называется корректным (well-formed) документом, а документ, кроме того, удовлетворяющий еще и правилам, заданным в определении типа документа, называется действительным (valid).

Именованный шаблон (Named Template) XSLT

Элемент <xsl:template>, для которого указан атрибут name. Именованный шаблон можно вызвать с помощью инструкции <xsl:call-template>.

Инструкция (Instruction) XSLT

Один из элементов XSLT, которые могут находиться непосредственно в теле шаблона, например <xsl:variable>, <xsl:choose> или <xsl:message>. Не все элементы XSLT являются инструкциями, так, элементы <xsl:param> и <xsl:when> не являются инструкциями, поскольку могут быть использованы в определенном контексте.

Инструкция обработки (Processing Instruction) XML

Конструкция в XML-документе, используемая для передачи инструкций программе, которая обрабатывает данный документ. Заключается между ограничителями «<?» и «?>». Заметьте, однако, что объявление XML в начале документа и текстовое объявление в начале внешней анализируемой сущности не являются инструкциями обработки, хотя и используют те же ограничители.

Исходный документ (Source Document) XPath

Основной исходный документ – это XML-документ, к которому применяется таблица стилей. С помощью функции `document()` могут быть загружены вторичные исходные документы.

Комментарий (Comment) XML

Фрагмент XML-документа, который предназначен для предоставления дополнительной информации, которая не относится собственно к документу. Записывается между ограничителями «<!--» и «-->».

Конечное дерево (Result Tree) XSLT

Выходные данные таблицы стилей. Таблица стилей определяет преобразование из исходного дерева в конечное. Некоторые продукты имеют расширения, которые позволяют создавать на выходе несколько деревьев. Конечная стадия обработки – это, как правило, сериализация конечного дерева в поток символов или байтов в зависимости от выбранного метода вывода.

Конечный литеральный элемент (Literal Result Element) XSLT

Конечный литеральный элемент – это элемент, находящийся внутри тела шаблона в таблице стилей, который не является инструкцией XSLT или элементом расширения. Когда тело шаблона подвергается обработке, то конечный литеральный элемент копируется в текущее назначение вывода, а затем, в свою очередь, обработке подвергается его содержимое (которое также является телом шаблона).

Контекстный узел (Context Node) XPath

Для выражения XPath, содержащегося непосредственно в таблице стилей (например, выражение в атрибуте `select` элемента `<xsl:value-of>` или выражение, заключенное в фигурные скобки шаблона значения атрибута), контекстный узел означает то же, что и текущий узел. Для выражения XPath, используемого в качестве предиката, текущим будет тот узел, для которого проверяется предикат. Получить контекстный узел можно при помощи выражения «.».

Корневой узел (Root Node) XPath

Самый верхний узел в дереве. Если дерево представляет правильно построенный XML-документ, то корневой узел будет иметь только один дочерний узел элемента, представляющий собой элемент документа и ни одного дочернего текстового узла. В других случаях (например, для фрагмента конечного дерева) у него может быть как ноль, так и более дочерних узлов элементов, а также ноль или более дочерних текстовых узлов: я говорю, что такие документы являются **сбалансированными (well-balanced)**. В обоих случаях корневой узел может иметь дочерние узлы комментариев и узлы инструкций обработки.

Корректный (Well-formed) XML

Документ является корректным, если он удовлетворяет правилам синтаксиса, определенным в спецификации XML. Эти правила включают требование, чтобы в документе был один самый внешний элемент, который включает в себя все остальные. Результат XSLT-преобразования не обязательно должен быть корректным, он должен быть сбалансированным.

Логический тип (Boolean) XPath

Один из допустимых в выражениях XPath типов данных. Принимает значения истина или ложь.

Локальная переменная (Local Variable) XSLT

Переменная, определенная внутри тела шаблона. Локальная переменная доступна только тем элементам, которые следуют за элементом <xsl:variable>, определяющим данную переменную, и которые имеют того же родителя, а также их потомкам. Это аналогично обычным правилам в языках программирования с блочной структурой.

Метод вывода (Output Method) XSLT

В XSLT определено три метода вывода: xml, html и text. Метод вывода определяет способ, с помощью которого из конечного дерева формируются выходные данные, как поток символов или байтов.

Набор атрибутов (Attribute Set) XSLT

Именованное множество инструкций <xsl:attribute>, которое, будучи вызванным либо при помощи атрибута use-attribute-sets элемента <xsl:element> или <xsl:copy>, либо при помощи атрибута xsl:use-attribute-sets конечного литерального элемента, формирует набор узлов атрибутов для добавления их текущему выводимому элементу.

Набор узлов (Node-set) XPath

Набор узлов – это неупорядоченное множество различных узлов из одного или нескольких деревьев. Он может быть пустым и гетерогенным, в том смысле, что в нем могут находиться узлы различного типа.

Неанализируемая сущность (Unparsed Entity) XML

Неанализируемая сущность – сущность, объявленная в определении типа документа со связанной нотацией. Такие сущности остаются неанализируемыми, так как обычно содержат двоичные данные, такие как изображения, а не XML-данные. При помощи функции unparsed-entity-uri() в XSLT можно получить неанализируемые сущности, связанные с исходным документом.

Не-число (NaN) XPath

Не-число (Not-a-Number, NaN). Это одно из возможных значений, принимаемых переменной, имеющей числовой тип. Оно возникает в результате операций, результат которых не число, например «number('яблоко')».

Образец (Pattern) XSLT

Конструкция, определяющая условие, которому каждый узел либо удовлетворяет, либо нет. Синтаксис для образцов является подмножеством синтаксиса выражений XPath. Образцы используются только в трех элементах XSLT: <xsl:template>, <xsl:key> и <xsl:number>.

Объявление пространства имен (Namespace Declaration) XML Namespaces

Конструкция, содержащаяся в XML-документе, которая говорит о том, что в определенной части документа для ссылки на пространство имен, заданное указанным URI, будет использоваться данный префикс. Существуют две формы объявления пространства имен: xmlns="uri" – для объявления пространства имен, используемого по умолчанию (с незаданным префиксом), и xmlns:prefix="uri" – для пространства имен с указанным префиксом. Обе формы указываются в виде XML-атрибутов и применяются к элементам, внутри которых объявлены, а также, если не переопределены, и к их потомкам.

Объявление пространства имен по умолчанию (Default Namespace Declaration) XML

Объявляется в виде атрибута XML как xmlns="uri". Тем самым декларируется, что для любого элемента, находящегося в области видимости и не имеющего явного префикса, будет использоваться данный URI пространства имен. Пространство имен по умолчанию применимо только к элементам – для других объектов, не имеющих префикса (например, атрибутов), URI пространства имен будет пустым.

Определение типа документа (Document Type Definition, DTD) XML

Определение структуры XML-документа или набора XML-документов. Может разделяться на внешнее подмножество, хранящееся в отдельном файле, и внутреннее подмножество, встроенное в сам документ.

Основной тип узла (Principal Node Type) XPath

Для каждой оси существует основной тип узла. Для большинства осей основные узлы – это элементы. Для оси attribute основным типом узла является атрибут, а для оси namespace таковым является пространство имен. Основной тип узла определяет тип узлов, выбираемых критерием узла «*»: например, выражение «following-siblings::*» выбирает элементы, в то время как выражение «namespace::*» выбирает узлы пространств имен.

Ось (Axis) XPath

Ось – это направление перемещения по дереву. Начинаясь от некоторого контекстного узла, ось задает тот набор узлов, к которым можно попасть из данного места. Например, ось `ancestor` возвращает родителя данного узла, затем его родителя и так вплоть до корневого узла, в то время как ось `following-sibling` возвращает все узлы, расположенные после контекстного узла и имеющие общего родителя.

Ось ancestor (Ancestor Axis) XPath

По оси `ancestor` выбираются: непосредственный предок контекстного узла, его предок и так далее вплоть до корневого узла включительно. Направление этой оси обратно порядку документа.

Ось ancestor-or-self (Ancestor-or-Self Axis) XPath

По оси `ancestor-or-self` выбираются контекстный узел и далее все узлы по оси `ancestor`. Направление этой оси обратно порядку документа.

Ось attribute (Attribute Axis) XPath

По оси `attribute` выбираются все атрибуты контекстного узла. Если контекстный узел не является элементом, то по оси ничего не будет выбрано.

Ось child (Child Axis) XPath

По оси `child` выбираются все непосредственные потомки контекстного узла. Это могут быть элементы, текстовые узлы, комментарии или инструкции обработки, но не узлы атрибутов или пространств имен.

Ось descendant (Descendant Axis) XPath

По оси `descendant` выбираются все непосредственные потомки данного узла, затем их потомки и так далее в порядке следования в документе.

Ось descendant-or-self (Descendant-or-Self Axis) XPath

По оси `descendant-or-self` выбираются контекстный узел, а за ним все узлы по оси `descendant`.

Ось following (Following Axis) XPath

По оси `following` выбираются все узлы, которые идут за контекстным, кроме узлов атрибутов и узлов пространств имен, а также потомков контекстного узла. Направление оси совпадает с порядком следования в документе.

Ось following-sibling (Following-Sibling Axis) XPath

По оси `following-sibling` выбираются все узлы, которые следуют за контекстным и имеют того же родителя. Направление оси совпадает с порядком документа.

Ось namespace (Namespace Axis) XPath

По оси `namespace` выбираются все узлы пространств имен, относящихся к контекстному узлу. Если контекстный узел не является элементом, то эта ось будет пуста. Что же касается элементов, то для каждого пространства имен, находящегося в области видимости данного элемента, всегда существует один узел пространства имен, относится ли оно к объявлению пространства имен из данного элемента или из элемента, содержащего его.

Ось preceding (Preceding Axis) XPath

По оси `preceding` выбираются все узлы, которые предшествуют контекстному узлу, в порядке, обратном порядку следования в документе, за исключением узлов атрибутов и пространств имен, а также предков контекстного узла.

Ось preceding-sibling (Preceding-Sibling Axis) XPath

По оси `preceding-sibling` выбираются все узлы, которые предшествуют контекстному узлу, а также имеют одинакового с ним родителя. Направление оси обратно порядку следования в документе.

Параметр (Parameter) XSLT

Переменная, значение которой передается вызывающей конструкцией. Глобальный параметр – это глобальная переменная, значение которой может быть установлено (определенным поставщиком способом) при применении таблицы стилей. Локальный параметр определяется внутри элемента `<xsl:template>`, а его значение может быть установлено при вызове шаблона инструкциями `<xsl:apply-templates>` или `<xsl:call-template>`.

Подвергать обработке (Instantiate) XSLT

Про инструкции и тела шаблонов в XSLT не говорят, что они исполняются, отрабатываются или активизируются, а что они подвергаются обработке (instantiate). Такой термин выбран для того, чтобы не возникло путаницы с тем, что исполнение должно быть обязательно последовательным.

Полное имя (QName) XML Namespaces

Полное имя (qualified name). Это либо имя без двоеточия (`NCName`), либо имя, перед которым стоят префикс и двоеточие.

Положение в контексте (Context Position) XPath

Для выражения XPath, содержащегося непосредственно в таблице стилей, положение в контексте – это позиция текущего узла в текущем списке узлов, где позиция отсчитывается от единицы. Для выражения XPath, используемого в качестве предиката на некотором шаге, положение в контексте – это позиция контекстного узла среди других узлов, выбранных на этом шаге в выражении пути в направлении оси шага. Положение в контексте оп-

ределяет значение, возвращаемое функцией position(), а также используется для вычисления численных предикатов, таких как «[1]».

Порядок следования в документе (Document Order) XPath

Узлы в наборе всегда отсортированы в порядке следования в документе. Для элементов из одного документа порядок следования в документе – это то же, что и порядок, в котором следуют их открывающие теги. В терминах древовидной структуры узел следует за предшествующими одноуровневыми узлами, порожденными от того же узла, а те, в свою очередь, за родительским узлом. Порядок узлов атрибутов и пространств имен, а также узлов из различных исходных документов, не определен.

Предикат (Predicate) XPath

Выражение, используемое в качестве условия выбора узлов на конкретном шаге выражения пути, или для выбора подмножества узлов из набора. Логическое выражение выбирает те узлы, для которых значение предиката есть истина, а численные выражения выбирают узлы, чья позиция совпадает со значением выражения, например предикат «[1]» выбирает первый узел.

Преимущество (Precedence) XSLT

См. Преимущество импортирования.

Преимущество импортирования (Import Precedence) XSLT

Таблица стилей, загруженная с использованием <xsl:import>, имеет меньшее преимущество импортирования по сравнению с импортирующей таблицей стилей. Преимущество импортирования оказывает влияние на все элементы верхнего уровня данной таблицы стилей при решении вопроса о том, какой из элементов верхнего уровня использовать. Например, если две глобальных переменных имеют одинаковые имена, то использована будет та, которая имеет большее преимущество импортирования.

Префикс (Prefix) XML Namespaces

См. Префикс пространства имен.

Префикс пространства имен (Namespace Prefix) XML Namespaces

Короткое имя, используемое для идентификации пространства имен внутри определенной области в таблице стилей и называемое так, потому что оно чаще всего используется как префикс в полном имени (часть полного имени, стоящая перед двоеточием). Для одного и того же пространства имен могут быть использованы разные префиксы, а в различных контекстах один и тот же префикс может использоваться для разных пространств имен.

Приоритет (Priority) XSLT

Каждое шаблонное правило имеет приоритет. Приоритет выражается числом с плавающей запятой. Приоритет можно задать явно при помощи атрибута `priority` элемента `<xsl:template>`; если же он опущен, то на основе образца присваивается приоритет по умолчанию. Приоритет используется при выборе шаблона для обработки узла, когда ему подходят сразу несколько шаблонных правил: используется то правило, которое имеет численно более высокий приоритет по сравнению с другими правилами.

Пробельные символы (Whitespace) XML

Пробельные символы – это любая непрерывная последовательность знаков табуляции, возврата каретки, перевода строки или пробелов. Пробельный узел – это текстовый узел, строковое значение которого состоит только из пробельных символов.

Пространство имен (Namespace) XML Namespaces

Именованное множество имен. Пространства имен именуются при помощи идентификатора URI, который предназначен для обеспечения уникальности имен, однако фактически это может быть любая строка. В каждой конкретной части документа пространство имен может быть также идентифицировано с помощью локального имени, называемого префиксом; различные префиксы могут использоваться для ссылок на одно и то же пространство имен в разных документах или даже внутри одного документа. Каждое имя (элемента или атрибута XML, а также переменной, шаблона, режима и т. п.) принадлежит определенному пространству имен, и два имени можно считать эквивалентными только в том случае, когда они принадлежат одному пространству имен.

Размер контекста (Context Size) XPath

Для выражения XPath, содержащегося непосредственно в таблице стилей, размер контекста – это количество узлов в текущем списке узлов. Для выражения XPath, используемого в качестве предиката на некотором шаге, размер контекста – это количество узлов, выбранных на этом шаге в выражении пути. Размер контекста определяет значение, возвращаемое функцией `last()`.

Расширенное имя (Expanded Name) XML Namespaces

Идентификатор, полученный из полного имени после замены префикса пространства имен на полный URI того пространства имен, к которому он относится. Расширенное имя состоит из двух частей: URI пространства имен и локального имени. Для отображения расширенных имен нет стандартного соглашения, хотя в некоторых интерфейсах, таких как TrAX, расширенные имена записываются в форме «`{uri-пространства-имен}локальное-имя`».

Режим (Mode) XSLT

Режимы разделяют набор шаблонных правил в таблице стилей таким образом, что один и тот же узел может быть обработан с помощью разных правил. Режим, указанный при вызове `<xsl:apply-templates>`, должен совпадать с именем режима, указанного в вызываемом элементе `<xsl:template>`.

Сбалансированный (Well-balanced) XML Fragment Interchange

Фрагмент XML-данных является сбалансированным, если каждому открывающему тегу соответствует закрывающий. Это менее строгое ограничение по сравнению с требованиями к корректному документу: сбалансированный фрагмент не обязан иметь единственный элемент, заключающий в себе все остальные. XSLT и XPath определены таким образом, что могут работать с любыми деревьями, представляющими сбалансированные XML-фрагменты. В XML и XSLT не используется такая терминология; вместо этого говорится о правилах для *внешней общей анализируемой сущности* (*external general parsed entity*).

Связывание переменной (Variable Binding) XPath

Объявление переменной внутри элементов `<xsl:variable>` или `<xsl:param>` одновременно с присвоением ей текущего значения.

Секция CDATA (CDATA Section) XML

Последовательность символов внутри XML-документа, заключенная между ограничителями «`<![CDATA[`» и «`]]>`»; внутри секции CDATA все символы являются текстом, а не разметкой, за исключением последовательности «`]]>`».

Ссылка на переменную (Variable Reference) XPath

Ссылка в форме `$name` на переменную, находящуюся внутри выражения.

Ссылка на символ (Character Reference) XML

Представление символа в виде десятичного или шестнадцатеричного значения Unicode, например «`
`» или «`↤`». Обычно такие ссылки используются для символов, которые трудно или невозможно набрать с клавиатуры.

Ссылка на сущность (Entity Reference) XML

Ссылка на внутреннюю или внешнюю сущность, как правило, в форме «`&name;`».

Строковое значение (String-value) XPath

Каждый узел имеет строковое значение. Для текстового узла строковое значение – это текстовое содержимое, для элемента – это сцепление строковых значений текстовых узлов потомков (то есть то текстовое содержимое элемента, которое останется, если убрать всю разметку). Строковое значение узла выводится при помощи инструкции `<xsl:value-of select=".">`.

Строковый тип (String) XPath

Один из типов данных, допустимых для значения выражения XPath. Это последовательность, состоящая из нуля или более символов Unicode (тот же набор символов, что применяется в XML).

Сущность (Entity) XML

Физическая единица информации, на которую можно ссылаться внутри XML-документа. Внутренние сущности встроены в сам документ в разделе определения типа документа, а внешние сущности, как правило, хранятся в отдельных файлах. Анализируемая сущность (parsed entity) содержит текст с разметкой XML, в то время как неанализируемая сущность (unparsed entity) содержит двоичные данные. Общая сущность (general entity) содержит данные, предназначенные для включения в сам документ, а параметрическая сущность (parameter entity) содержит данные для включения в определение типа документа (DTD).

Таблица стилей (Stylesheet) XSLT

Этот термин используется для ссылок как на одиночный элемент `<xsl:stylesheet>` и его содержимое (называемые в данной книге модулем таблицы стилей), так и на таблицу стилей, загружающую другие таблицы стилей при помощи элементов `<xsl:include>` и `<xsl:import>` (называемую программой таблиц стилей).

Текстовый узел (Text Node) XPath

Узел дерева, представляющий символьные данные (в XML известные как PCDATA) внутри XML-документа. Смежные текстовые узлы всегда сливаются в один узел. Ссылки на символы и сущности, встречающиеся в исходном тексте, заменяются их расширениями.

Текущее назначение вывода (Current Output Destination) Данная книга

В спецификации XSLT говорится, что такие инструкции как, `<xsl:value-of>` и `<xsl:element>`, записывают узлы в конечное дерево. Однако в то время, когда обработке подвергается элемент `<xsl:variable>`, любой вывод, на самом деле, перенаправляется во фрагмент конечного дерева, который и формирует значение этой переменной. Поэтому в данной книге о любом дереве, в которое производится запись, говорится как о текущем назначении вывода. В рабочем проекте XSLT 1.1 (а также во многих реализациях XSLT 1.0, включающих расширения от производителей) имеется возможность производить вывод в несколько деревьев.

Текущее шаблонное правило (Current Template Rule) XSLT

Когда инструкция `<xsl:apply-templates>` выбирает шаблонное правило для обработки конкретного узла, то это шаблонное правило становится текущим. Оно остается текущим шаблонным правилом во время вызовов

<xsl:call-template>, но не во время вызовов <xsl:for-each>. Текущее шаблонное правило используется только при выборе шаблона, который должен быть применен при вызове <xsl:apply-imports>.

Текущий список узлов (Current Node List) XSLT

Текущий список узлов – это список (упорядоченное множество) узлов в дереве исходного документа. Текущий список узлов устанавливается выражением select инструкции <xsl:apply-templates> или <xsl:for-each>. По умолчанию узлы в текущем списке узлов находятся в порядке следования в документе, но порядок может отличаться, если задан элемент <xsl:sort>. При вычислении выражения XPath в таблице стилей положение текущего узла в текущем списке узлов задает положение в контексте (значение, возвращаемое функцией position()), а размер текущего списка узлов определяет размер контекста (значение, возвращаемое функцией last()).

Текущий узел (Current Node) XSLT

Узел в исходном дереве становится текущим, когда подвергается обработке с помощью инструкций <xsl:apply-templates> или <xsl:for-each>. Получить текущий узел непосредственно можно при помощи функции current(). За исключением того случая, когда узел находится внутри предиката в выражении пути, текущий узел – это то же самое, что и контекстный узел, поэтому его можно также получить, используя выражение «.».

Тело шаблона (Template Body) Данная книга

Последовательность инструкций XSLT, элементов расширения, конечных литеральных элементов и текстовых узлов, образующих содержимое элемента <xsl:template> или некоторых других элементов таблицы стилей. Когда тело шаблона подвергается обработке, любые инструкции и элементы расширения обрабатываются в соответствии с определенными для них правилами, в то время как конечные литеральные элементы и текстовые узлы просто копируются в текущее назначение вывода. В Рекомендации по XSLT для этого используется термин *шаблон* (*Template*); я же использую термин *тело шаблона* (*Template Body*), чтобы избежать путаницы с элементом <xsl:template>.

Узел (Node) XPath

Объект дерева. Существуют семь типов узлов: узлы атрибутов, узлы комментариев, узлы элементов, узлы пространств имен, узлы инструкций обработки, корневые узлы и текстовые узлы.

Узел атрибута (Attribute Node) XPath

Узел в дереве, соответствующий атрибуту в документе XML. Так, каждый узел элемента, который имеет в своем открывающем теге определение атрибута, не являющегося определением пространства имен, будет связан с узлом этого атрибута. Также в дереве будут присутствовать узлы атрибутов,

которым присваивается значение по умолчанию в определении типа документа (DTD). Строковое значение узла и есть значение атрибута.

Узел инструкции обработки (Processing Instruction Node) XPath

Узел дерева, представляющий инструкцию обработки XML.

Узел комментария (Comment Node) XPath

Узел дерева, представляющий собой XML-комментарий. Строковое значение этого узла содержит текст комментария.

Узел пространства имен (Namespace Node) XPath

Узел дерева, в котором представлена связь префикса пространства имен с его URI. Узел пространства имен принадлежит элементу, который называется его родителем: это относится только к этому элементу, но не к его потомкам.

Узел элемента (Element Node) XPath

Узел дерева, который соответствует элементу XML-документа. Родителем узла элемента может быть либо содержащий его элемент, либо корень дерева, а его детьми могут быть узлы элементов, текстовые узлы, узлы комментариев, а также узлы инструкций обработки, извлеченные непосредственно из содержимого данного элемента XML.

Упрощенная таблица стилей (Simplified Stylesheet) Данная книга

Таким образом мы называем то, что в XSLT называется *«конечный литеральный элемент» как таблица стилей* (*Literal Result Element as Stylesheet Facility*). Упрощенная таблица стилей – это таблица стилей, состоящая только из конечного литерального элемента, который обрабатывается с использованием корня текущего документа в качестве текущего узла.

Фрагмент конечного дерева (Result Tree Fragment) XSLT

В XSLT 1.0 **фрагмент конечного дерева** – это тип данных временного дерева, созданных в результате обработки непустого содержимого элемента `<xsl:variable>`. В рабочем проекте XSLT 1.1 тип данных временного дерева становится просто набором узлов, который содержит один-единственный узел – корень дерева.

Функция (Function) XPath

Процедура, которая может быть вызвана из выражения XPath; она принимает параметры и возвращает результат. Посредством XPath можно только вызывать функции, но не определять их. Функция может быть либо основ-

ной, то есть определенной в рекомендациях XPath и XSLT, либо расширением, созданным поставщиком или пользователем.

Функция расширения (Extension Function) XSLT

Функция, определенная поставщиком продукта, пользователем или третьей стороной, которая может вызываться из выражения XPath. В Рекомендации по XSLT определяется, как вызывать функции расширения, но не то, как их реализовывать.

Числовой тип (Number) XPath

Один из типов данных, допустимых для значений выражений XPath. Это число с плавающей запятой, определенное в стандарте IEEE 754.

Шаблон значения атрибута (Attribute Value Template) XSLT

Шаблон значения атрибута – это атрибут, находящийся в таблице стилей, который имеет как фиксированные, так и изменяющиеся части. Фиксированные части пишутся как обычные символы, в то время как изменяющиеся части заключаются в фигурные скобки: например, `«file="{$dir}/{$fname}.html"»` может иметь значение `«file="out/page.html"»`, если переменные `$dir` и `$fname` имеют значения `«out»` и `«page»`, соответственно. Шаблоны значений атрибутов могут быть использованы в любом конечном литеральном элементе, что же касается элементов XSLT, то там шаблоны могут использоваться только в тех атрибутах, которые явно позволяют это.

Шаблонное правило (Template Rule) XSLT

Элемент `<xsl:template>` таблицы стилей, имеющий атрибут `match`. Шаблонное правило может быть вызвано при помощи инструкции `<xsl:apply-templates>`; шаблонное правило для каждого выбранного узла определяется на основе ряда критериев, включая совпадение с образцом, а также преимущество импортирования и приоритет.

Шаг (Step) XPath

Шаг используется в выражении XPath для перехода от некоторого узла к некоторому набору узлов. Шаг задается осью, определяющей направление перехода; критерием узла, ограничивающим возможные узлы по типу и имени, а также возможными предикатами, накладывающими на узлы произвольные ограничения.

Элемент (Element) XML

Логическая единица XML-документа, ограниченная открывающим и закрывающим тегами, например `<издательство>Wrox Press</издательство>`; пустой элемент может быть записан в сокращенной форме, например `<издательство название="Wrox"/>`.

Элемент верхнего уровня (Top-level Element) XSLT

Элемент таблицы стилей, который является непосредственным потомком элемента `<xsl:stylesheet>`.

Элемент документа (Document Element) XML

Самый внешний элемент документа, то есть тот, который содержит в себе все остальные элементы. В стандарте XML такой элемент также называется корневым, однако его не нужно путать с корневым узлом дерева в терминологии XPath, который является родителем элемента документа и представляет собой сам документ.

Элемент расширения (Extension Element) XSLT

Элемент, используемый в теле шаблона, который определен поставщиком продукта, пользователем или третьей стороной, но при этом действует также, как и инструкция XSLT. В Рекомендации по XSLT определяется, как обрабатывать элементы расширения, но не то, как их реализовывать.

Алфавитный указатель

Специальные символы

Числа

D

E

F

G

H

I

J

L

M

T

U

V

W

X

A

Б

В

Г

К

Л

М

Н

О

П

P

С

T

У

Ф

Х

Ц

Ч

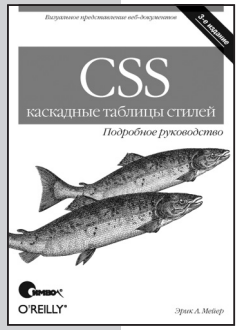
Ш

Э

Я

Эрик МЕЙЕР

CSS – каскадные таблицы стилей, 3-е издание



576 стр., книга в продаже

Хотите быстро и без усилий разрабатывать стилевое оформление веб-страниц, отвечающее современным требованиям? В третьем издании показывается, как реализовать на практике все возможности каскадных таблиц стилей для стандартов CSS2 и CSS2.1.

Известный эксперт по CSS Эрик Мейер, опираясь на свой богатейший опыт, рассматривает свойства, теги, атрибуты и реализации, а также практические вопросы, такие как поддержка различными броузерами и рекомендации разработчикам. Вы узнаете о сложном стилевом оформлении документов, пользовательском интерфейсе, верстке таблиц, списках и генерируемом содержимом, о всех деталях свободного перемещения и позиционирования, о семействах шрифтов и механизмах резервирования, о новых селекторах CSS3, поддерживаемых IE7, Firefox и другими броузерами, а также о том, как работает модель блоков.

В третьем издании подробно рассматриваются все свойства CSS и их взаимодействие. Книга поможет избежать распространенных ошибок. Она является полным справочником по CSS и будет полезна как опытному веб-разработчику, так и новичку в этих вопросах. От читателя потребуется только знание HTML 4.0.

Дуг ТИДУЭЛЛ

XSLT, 2-е издание

960 стр., книга в продаже

Эта книга не только научит эффективно работать с XSLT, но и послужит удобным справочником по всем функциям и возможностям языка. Второе издание включает примеры таблиц стилей для XSLT 1.0 и XSLT 2.0. Вы познакомитесь с основами XSLT, в том числе с настройкой процессоров. Особое внимание уделяется преобразованиям на базе шаблонов, обсуждаются основные концепции XPath 1.0 и 2.0. Вы узнаете, как в XSLT 2.0 интегрирована поддержка XML Schema, научитесь определять элементы и типы данных и использовать их в своих таблицах стилей, рассмотрите сотни таблиц стилей, в том числе примеры для каждого элемента, функции и оператора, определяемых в XSLT и XPath, увидите, как одна и та же задача решается в XSLT 1.0 и 2.0, что поможет решить, какая версия лучше всего подходит для вашего проекта.

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:
главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 324-5353

Москва:
московский филиал издательства –
ул. Беговая, д. 13 (м. Динамо),
тел. (495) 945-8100

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!