Auf einen Blick

Inhalt

5 Referenz XPath-Funktionen 319

6 Referenz XSLT-Elemente 617

Einleitung

Warum ein Buch über XSLT? Wie ist dieses Buch aufgebaut und an wen richtet es sich? Welche Themen werden behandelt? Welche Inhalte werden nicht abgedeckt? Welche Darstellungskonventionen werden verwendet? Was befindet sich auf der Begleit-CD?

Ein beinahe 1000 Seiten starkes Werk über eine Sprache zu schreiben, deren Aufgabe im Wesentlichen die Umwandlung eines XML-Formats in ein anderes ist, verlangt möglicherweise nach einer Begründung. Hätte das nicht ein wenig kürzer, kompakter ausfallen können? Ist das Thema denn überhaupt so wichtig, um diesen Umfang zu rechtfertigen?

Warum ein dickes Buch über XSLT?

Nun, abgesehen davon, dass die Arbeit an diesem Buch mir – zumindest die meiste Zeit über – viel Spaß gemacht hat, gibt es auf diese Fragen noch einige weitere Antworten. Der am Anfang einschüchternde Umfang ergibt sich primär aus dem in der zweiten Version von XSLT und XPath erheblich gesteigerten Sprachumfang. Ein Großteil des Buches ist deshalb den Referenzen der einzelnen Instruktionen von XSLT, den Erläuterungen zu den Funktionen von XPath sowie einem abschließenden, umfassenden Glossar der in diesem Umfeld verwendeten Fachausdrücke und Abkürzungen gewidmet.

Der erste Teil des Buches gibt eine didaktische Einführung in das Thema XSLT und XPath. Die Trennung in einen didaktischen Teil und eine Referenz beruht auf der Überlegung, dass eine vollständige Diskussion der jeweils im Stoff auftauchenden Elemente und Konzepte innerhalb eines einführenden Textes nicht möglich sein würde, ohne die Lesbarkeit und Dynamik eben dieser Einführung zu beeinträchtigen. Der Referenzteil erfordert zudem eine umfassende Detaildarstellung und damit einen eher lexikalischen Stil, der beim didaktischen Teil fehl am Platz wäre – dieser darf und soll auch Spaß machen, und muss deshalb auslassen und vereinfachen dürfen, wenn dies der Darstellung des zu vermittelnden Stoffes dient.

Warum überhaupt ein Buch über XSLT?

Eine simple Antwort auf diese Frage lautet natürlich, dass dieses Thema »wichtig« ist – immerhin ist XSLT ein unabhängiger Standard zur Informationsverarbeitung und gehört zur XML-Familie, ist also naturgemäß Favorit für die Verarbeitung von in XML-Format vorliegenden Daten. Nicht ohne Grund, denn genau hierfür wurde XSLT konzipiert.

XML selbst ist über das Stadium des Hypes und reinen Buzzwords inzwischen weit hinaus und hat sich zum anerkannten Standard der industriellen Informationsspeicherung, -weitergabe und -verarbeitung gemausert. Der Sprachstandard wurde vom W3C unter Mitwirkung zahlreicher Fachleute aus der Industrie definiert, was eine breite, inzwischen geradezu allgegenwärtige Unterstützung der Sprache gewährleistet.

Mit der Verbreitung von XML steigt die Notwendigkeit einer Verarbeitung entsprechender Dokumente – auch diese sollte plattformunabhängig erfolgen. Es ist zwar denkbar, eine Verarbeitung von XML in verschiedensten Programmiersprachen zu schreiben, jedoch dann ohne die Möglichkeit, die so erstellten Verarbeitungsvorschriften einfach portieren zu können.

XSLT stellt in dieser Hinsicht die Lösung dar: als plattformunabhängige, selbst in XML geschriebene Anwendungen sind XSLT-Stylesheets so gut wie problemlos von einer Einsatzumgebung in die andere portierbar, sei es aus einer PHP-Umgebung, in der die Verarbeitung mit Sablotron erfolgt, aus der NET-Umgebung oder einer Java-Umgebung von Cocoon. Wo die Umgebungen wechseln, kann das XSLT-Stylesheet gewissermaßen eine Konstante darstellen.

Die Aufgaben, die XSLT dabei zu bewältigen hat, werden zunehmend komplexer. Heutzutage spielt sich ein Großteil der XML-Verarbeitung in Zusammenhang mit Datenbanken ab, die Gewährleistung einer korrekten Verarbeitung in Bezug auf Gültigkeit sowohl der verarbeiteten Daten als auch der Ergebnisse wirft neue Schwierigkeiten auf, denen mit der neuen Version 2.0 von XSLT und XPath Rechnung getragen wird.

Die Aufeinanderzubewegung von Datenbanken und XML wird an der Verschmelzung von XPath und der als Abfragesprache konzipierten XML-Query Language zur allgemeinen XML-Abfragesprache XQuery 1.0 deutlich, von der XPath 2.0 »nur« eine Untergruppe darstellt. Im Zentrum der Verarbeitung von Daten in XML-Form, gleichgültig, ob diese nun als »reale« Dokumente vorliegen oder als dynamisch mittels Datenbankabfrage erzeugte »virtuelle« Dokumente, wird stets XSLT stehen.

Welche Inhalte werden behandelt?

Dieses Buch beschäftigt sich gleichrangig mit **XSLT** und **XPath** – beide Sprachen werden in der Regel gemeinsam verwendet (XPath kann allerdings durchaus auch in anderen Umgebungen auftauchen), sodass eine Trennung nicht sinnvoll ist.

Sowohl XPath als auch XSLT werden in ihrer aktuellen **Version 2.0** vorgestellt. Auf Änderungen gegenüber XSLT 1.0 wird bei der Behandlung der Instruktio-

nen von XSLT 2.0 hingewiesen; der Versionssprung stellt im Wesentlichen eine Erweiterung gegenüber der Vorgängerversion dar. **XPath 2.0/XQuery 1.0** ist hingegen eine fast neu konzipierte, lediglich »rückwärtskompatible« Sprache – sowohl was den erheblich erweiterten Sprachumfang als auch das Verhalten der einschlägigen Funktionen betrifft.

Dieses Buch soll ausdrücklich keine rückwirkende Einführung in **XSLT 1.0** und **XPath 1.0** sein. Vorkenntnisse dieser Versionen beider Sprachen werden allerdings nicht vorausgesetzt. Nach Durcharbeitung des Buches sollten Sie sich ebenfalls in einer XSLT 1.0/XPath 1.0-Umgebung leicht zurecht finden können – für »Umsteiger« weist der Referenzteil auf entsprechende Unterschiede zwischen beiden Versionen und denkbare Klippen hin.

Kenntnisse in **XML** und zumindest rudimentäre Kenntnisse in **XML Schema** sollten beim Leser vorhanden sein. Eine Behandlung hätte einerseits den ohnehin umfangreichen Rahmen gesprengt, auf der anderen Seite existiert Literatur hierzu bereits in hinreichender Menge.

Ebenfalls nicht Thema dieses Buches konnte die vollständige Spezifikation von **XQuery** sein. XPath 2.0 stellt zwar von dieser eine nicht unwesentliche Teilmenge dar, der ganze Sprachumfang von XQuery 1.0 kann jedoch (zumindest zur gegenwärtigen Zeit) nur separat von XSLT eingesetzt werden. Da dieses Buch aus der Sicht von XSLT geschrieben ist, stellte eine Beschreibung der dort nicht nutzbaren Funktionalitäten von XQuery m. E. lediglich Ballast dar, der bestenfalls zur Verwirrung führen würde.

Hinweis Wer keine Erfahrungen bzw. Kenntnisse in XML oder XML Schema besitzt, sollte entsprechende Einführungswerke zur Hand nehmen (z. B. »Einführung in XML« von Helmut Vonhoegen, Galileo Computing oder vergleichbare). Die Erarbeitung von Grundkenntnissen in XML und XML Schema ist nicht Teil dieses Buches, sondern diese müssen in gewissem Rahmen vorausgesetzt werden. Wo immer möglich, wird auf versteckte Problematiken eingegangen.

Welche Darstellungskonventionen werden verwendet?

Ziel war es, den Text gut lesbar zu erhalten, ohne ein Übermaß an zusätzlichen, im Fließtext verwendeten Formatierungen. Sofern erforderlich oder sinnvoll, werden **Schlüsselbegriffe**, die von zentraler Bedeutung sind oder einer schnellen Sinnerfassung dienen, bei ihrem ersten Auftreten **fett** markiert.

Da eine Übersetzung aller erforderlichen **Fachbegriffe** nicht in jedem Fall eindeutig bzw. festgelegt ist, wird bei der Einführung eines solchen Begriffes der

Klarheit halber oft auch die englischsprachige Originalbezeichnung (*in Klammern, kursiv*) beigefügt.

Die Bezeichner von `Instruktionen` und `Funktionen` sowie kurze, im Fließtext auftretende `Quelltextbeispiele` werden wie hier in nicht-proportionaler Schrift dargestellt.

Dasselbe gilt für die unvermeidlichen längeren **Quelltextbeispiele**:

```
<quelltext-beispiel>
   <!--
   Längere Quelltextpassagen werden als separate Blöcke
   abgesetzt. Soll innerhalb dieser auf
   einzelne Elemente besonders hingewiesen werden,
   so werden diese fett markiert.
   -->
</quelltext-beispiel>
```

Wichtige Hinweise
Wichtige Hinweise werden, wie hier, durch grau unterlegte Boxen hervorgehoben. Es kann sich hierbei um eine wiederholende Kurzdarstellung von Schlüsselkonzepten oder um kontextspezifische Anmerkungen handeln.

Was befindet sich auf der Begleit-CD?

Neben den Quelltextbeispielen, die in diesem Buch verwendet werden, befinden sich auf der CD die in Kapitel 8, *XSLT-Entwicklungsumgebungen*, vorgestellten Programme – soweit sich dies realisieren ließ. Wo eine Aufnahme in die CD aus rechtlichen oder platztechnischen Gründen nicht möglich war, sind Links zum Download angegeben.

An dieser Stelle muss ich allerdings darauf hinweisen, dass die gegenwärtig (Mai 2004) erhältlichen kommerziellen Programme nur den Sprachumfang von XSLT 1.0 in Verbindung mit XPath 1.0 unterstützen. Dies ist auch nicht verwunderlich ist, da XSLT 2.0 einen völlig neuen Standard darstellt. Mit einer Adaption von XSLT 2.0 ist seitens der großen Anbieter jedoch spätestens zum Jahresende 2004 zu rechnen.

Der einzige XSLT-Prozessor, der XSLT 2.0 in vollem Umfang implementiert, ist Michael Kays **Saxon 7.x**. Dieser Prozessor ist ebenfalls Teil der aktuellen Version von Rob Rohans Open Source XSLT-Umgebung **Treebeard** (bei den XSLT-Editoren beschrieben), die hiermit tauglich für XSLT 2.0 ist, wenn auch ohne Unterstützung von XML Schema. Die meisten Beispiele dieses Buches sind somit in Treebeard nachvollziehbar.

Über mehr oder minder einfache Umwege lässt sich Saxon 7 auch in die anderen aktuellen Umgebungen einbeziehen, sodass auch in diesen – mit unterschiedlich starken Abstrichen zum ansonsten gebotenen Komfort – testweise eine XSLT 2.0-Entwicklung möglich ist. Das hierfür jeweils erforderliche Vorgehen wird im betreffenden Kapitel beschrieben.

Danksagungen

Kein Buch schreibt sich von alleine oder entsteht im luftleeren Raum. Dieses hier wäre nicht möglich gewesen ohne die Arbeitsgruppen des W3C, denen es gelang, mit XSLT 2.0, XPath 2.0 und den weiteren, hiermit verbundenen Themen eine zwar höchst komplexe, aber in sich logische Spezifikation zu schaffen. Zum Zeitpunkt des Erscheinens dieses Buches wird XSLT 2.0 noch eine sehr, sehr neue Sprache sein – allerdings eine mit einer großen Zukunft.

Dieses Buch wäre aber auch nicht entstanden ohne die Unterstützung einiger Menschen, denen ich hier ausdrücklich danken möchte. An erster Stelle natürlich meiner Frau, Carina Prange, die mir verständnisvoll den Rücken frei hielt und es mir so ermöglichte, einigermaßen guten Gewissens die erforderliche Zeit zu investieren. An zweiter Stelle meinem Lektor bei Galileo Computing, Stephan Mattescheck, ohne dessen Geduld und stetige Aufmunterung ich dieses Projekt vermutlich nie zu Ende gebracht hätte.

Last not least mein Dank an alle Leser dieses Vorworts; besonders an die, die sonst keine Vorworte lesen. Vielen Dank für Ihre Geduld. Ich hoffe, dass dieses Buch Ihnen einen spannenden und kurzweiligen Einstieg in die Welt von XSLT eröffnen wird. So, nun reicht es aber – gehen wir an die Arbeit!

Frank Bongers
Berlin, Mai 2004

1 XSLT und XPath – der Start

1 XSLT und XPath – der Start

Dieses Kapitel dient als allgemeiner Einstieg in XSLT: Anhand einfacher Beispiele werden die Grundstruktur eines XSLT-Stylesheets und erste XSLT-Instruktionen erläutert. Weiterhin wird ein Blick auf Baumstrukturen und die Bedeutung von XPath im Verhältnis zu XSLT geworfen.

Da dieses Buch sich beinahe ausschließlich mit **XSLT** beschäftigen wird, ist dies der geeignete Platz, um zunächst kurz das Thema zu umreißen. Beginnen wir also beim Anfang, zunächst ohne uns um »technische« Aspekte zu kümmern …

1.1 Die Themenverortung

Ein **XML-Dokument** beschreibt die in ihm enthaltene Information lediglich in ihrer Struktur. Daher muss diese, z. B. zu **Präsentationszwecken**, in eine hierfür geeignete Form gebracht werden. Oft muss dies mit einer **Umformung** der Informationsstruktur einhergehen, beispielsweise deren Sortierung oder Filterung. Auch für den **Austausch von Informationen** in XML-Format, wie es im B2B-Bereich erforderlich ist, müssen XML-Quellformate in anders strukturierte Zielformate überführt werden. Hierbei kann es erforderlich sein, neue Inhalte hinzuzufügen, die im Dokument selbst nicht enthalten sind – sie können aus anderen Quellen stammen oder durch Zusammenfassung oder Berechnungen aus den Ursprungsinformationen gewonnen werden.

Das ursprüngliche Dokument bleibt dabei erhalten: Aus einem **Quelldokument** wird stets ein neues **Ergebnisdokument** erzeugt. Dieses kann wiederum XML-Format, aber auch ein (fast beliebiges) anderes Format besitzen.

Diese Art der Verarbeitung von XML-Dokumenten, wie sie hier betrachtet werden soll, wird mittels **XSL**, der **eXtendable Style Language** beschrieben, bei der es sich um eine XML-Anwendung handelt. XSL zerfällt eigentlich in zwei Untergruppen, von denen die eine, **XSL-FO** (XSL Formatting Objects[1]), die Druckpräsentation zum Ziel hat. Die andere, die uns vorwiegend interessiert und die deshalb Thema dieses Buchs ist, beschreibt ausschließlich die **Transformation** von XML-Dokumenten, also deren Umformung: **XSLT**, die **eXtendable Style Language Transformations**. In XSLT formulierte Dokumente

1 Über XSL-FO müsste ein eigenes Buch geschrieben werden. Für uns interessant mag aber die Tatsache sein, dass der Weg von einem XML-Dokument zu dessen FO-Repräsentation wiederum über eine Transformation mit XSLT führt, XSL-FO also XSLT nachgeschaltet sein kann.

bezeichnet man aus historischen Gründen als **Stylesheets**, obwohl sie funktional weit über eine statische Darstellungsbeschreibung hinausgehen.

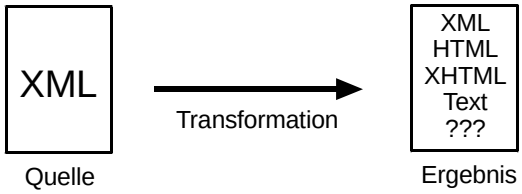

Abbildung 1.1 Quell- und Ergebnisdokument

- Was bedeutet XSLT?
  Die Abkürzung **XSLT** steht für **eXtendable Stylesheet Language Transformations**. XSLT gehört zu den deklarativen Sprachen und beschreibt und steuert die Umwandlung (Transformation) von XML-Dokumenten in andere Formate wie HTML, XHTML, Text oder andere XML-Strukturen.

Obwohl das Buch den Titel »XSLT« trägt, wird genauso oft von **XPath** die Rede sein. In XSLT eingebettet dienen XPath-Ausdrücke sowohl der Steuerung der eigentlichen Transformation als auch der unmittelbaren Erzeugung von Inhalten. XPath ist eine **Pfadbeschreibungssprache**, in der sich der Weg zu einer, in einem XML-Dokument befindlichen Information formulieren lässt.

- Was bedeutet XPath?
  Die Langform des Begriffes XPath ist **XML Path Language**. XPath dient der Navigation innerhalb von Dokumentbäumen und kann beliebige Knotenmengen eines solchen Baums benennen sowie die dort liegenden Informationen zugänglich machen.

In Zusammenhang mit **XSLT 2.0** wird stets auch **XPath 2.0** verwendet;[2] analog arbeitet **XSLT 1.0** ausschließlich mit **XPath 1.0**. Andere Kombinationen sind ausdrücklich nicht möglich.

2 Bei XPath 2.0 handelt es sich eigentlich um eine (allerdings wesentliche) Untergruppe von XQuery, einer äußerst mächtigen Abfragesprache für XML, die in voller Funktionalität jedoch nur unabhängig von XSLT nutzbar ist.

Eine **Kompatibilität** zwischen XSLT 1.0 und XSLT 2.0 ist relativ weit reichend gegeben: Sämtliche Elemente von XSLT 1.0 existieren auch in XSLT 2.0, zum Teil jedoch mit verändertem Verhalten oder erweiterter Funktionalität. Dies hat unter anderem mit dem gegenüber XPath 1.0 wesentlich erweiterten Umfang von XPath 2.0 zu tun, jedoch auch mit dem einen, wesentlichen Unterschied von XSLT 2.0 gegenüber XSLT 1.0: der Unterstützung von **Typisierung** der verarbeiteten Information gemäß **XML Schema**.

Wir werden uns in der Folge mit XSLT 2.0 und XPath 2.0 beschäftigen, die ich der Kürze halber deshalb lediglich mit »XSLT« und »XPath« bezeichnen werde.

Hinweis: Installieren Sie jetzt Ihren XSLT-Prozessor
Die auf den folgenden Seiten beschriebenen Beispiele erschließen sich auf Grund ihrer Einfachheit durchaus auch auf theoretischem Weg. Wenn Sie die Transformationen jedoch sofort in der Praxis durchführen wollen, sollten Sie zunächst den XSLT-Prozessor **Saxon 7** auf Ihrem Rechner installieren – eventuell in Zusammenhang mit einer XML-Entwicklungsumgebung Ihrer Wahl. Alle erforderlichen Installationsdateien und die vorgestellten Beispieldateien sind auf der Begleit-CD des Buches enthalten. Eine entsprechende **Hilfestellung** zur Installation und Verwendung von Saxon 7 bzw. zu den Entwicklungsumgebungen finden Sie im Referenzteil des Buches.

1.2 Eine erste Transformation

Anstatt hier lange theoretische Erörterungen zu beginnen, sollen einige möglichst einfache Beispiele vorgestellt werden, an denen sich aber bereits in Ansätzen zeigen lässt, worum es bei der Anwendung von XSLT geht.

1.2.1 Drei beteiligte Dokumente

Bei jeder Transformation sind drei Dokumente beteiligt – zunächst das **Quelldokument**, bei dem es sich um ein XML-Dokument handelt und das die Informationen beinhaltet. Zweitens das **XSLT-Stylesheet**: In diesem stehen die Anweisungen, wie mit den Informationen aus dem Quelldokument zu verfahren ist. Drittens das im Verlauf der Transformation erzeugte **Ergebnisdokument** – sein Aufbau ergibt sich gleichermaßen aus den Anweisungen des Stylesheets als aus Struktur und Inhalt des Quelldokuments.

Das Quelldokument

Ein Quelldokument muss für die Durchführung einer Transformation zwingend vorliegen – hier wird eine möglichst einfache XML-Datei verwendet, um

den Blick nicht durch nebensächliche Aspekte abzulenken. Ein leeres Wurzelelement genügt vorläufig bereits:

```
<?xml version="1.0"?>
<quelldokument/>
```

Listing 1.1 beispiel1.xml (alle derart markierten Beispiele: siehe Begleit-CD)

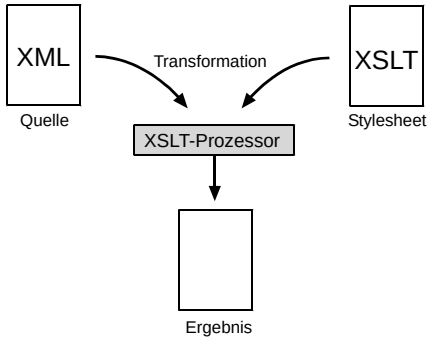

Abbildung 1.2 Schematisch dargestellte Transformation

Das Stylesheet

Zur Verarbeitung wird natürlich auch ein XSLT-Stylesheet benötigt (auch dieses Dokument ist für das Beispiel so einfach wie möglich gehalten). Deutlich ist zu sehen, dass einige Tagnamen mit `xsl:` beginnen – diese bilden das funktionale Gerüst des Stylesheets. Darüber hinaus ist noch »normales« HTML enthalten:

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <html>
      <body>
      <h1 align="center">Dies ist ein Ergebnisdokument.</h1>
      </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

Listing 1.2 beispiel1.xsl

Das Ergebnisdokument

Folgendes Ergebnisdokument wird erzeugt – in diesem Fall handelt es sich um ein simples HTML-Dokument:

```
<html>
<body>
<h1 align="center">Dies ist ein Ergebnisdokument.</h1>
</body>
</html>
```

Listing 1.3 beispiel1.html

Vergleicht man Ergebnisdokument und Stylesheet, so bemerkt man, dass das Ergebnisdokument als Ganzes – wenn auch in weitere Elemente verkapselt – bereits im Stylesheet enthalten war (dort im Quellcode fett ausgezeichnet).

Dies ist in der hier dargestellten Absolutheit natürlich ein Extremfall. Ebenso extrem an diesem Beispiel ist, dass es so scheint, als würde das XML-Quelldokument im Prinzip gar nicht verwendet (was kein Wunder ist, denn es steht nichts drin). Eine gewisse Rolle spielt es dennoch, wie später zu sehen sein wird.

1.2.2 Kopierte Inhalte: Literal Result Elements

Zunächst soll nur einfach vermerkt werden, dass ein Stylesheet Bereiche enthält, die »wie sie sind« im Ergebnisdokument wieder auftauchen. Weil sie dabei nicht verändert werden, also »literal« übernommen werden, bezeichnet man sie auch als **Literal Result Elements** (literale Ergebniselemente).

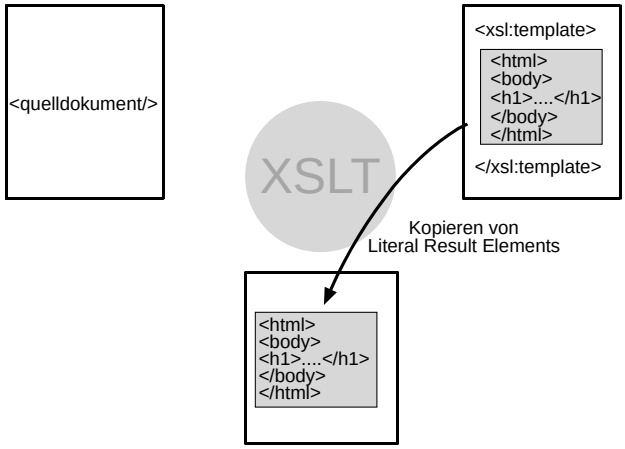

Abbildung 1.3 Kopieren der LRE ins Ergebnisdokument

1.2.3 Erster Einsatz von XSLT-Instruktionen

Das Beispiel wird jetzt erweitert, indem ein XML-Dokument verwendet wird, das verwertbare Information enthält. Im vorliegenden Fall handelt es sich um

eine Zeichenkette, die ins Ergebnisdokument kopiert werden soll. Um das Stylesheet dazu zu veranlassen, muss es gegenüber dem vorigen Beispiel ein wenig ausgebaut werden.

Hier zunächst das neue Quelldokument:

```
<?xml version="1.0"?>
<gruss>Hallo Welt! Dies ist XSLT.</gruss>
```

Listing 1.4 beispiel2.xml

Für das zu erzeugende HTML-Dokument (vorläufig bleibt es bei HTML als Ergebnisformat) wird folgende Form angestrebt:

```
<html>
<body>
<h1 align="center">Dies ist ein Ergebnisdokument.</h1>
<p align="center">Hallo Welt! Dies ist XSLT.</p>
</body>
</html>
```

Listing 1.5 beispiel2.html

Das Ergebnis wird also einen <p>-Container mit dem Text enthalten, der im Quelldokument im Element <gruss> steht. Das benötigte Stylesheet könnte beispielsweise so aussehen:[3]

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <html>
      <body>
      <h1 align="center">Dies ist ein Ergebnisdokument.</h1>
      <p align="center"><xsl:value-of select="gruss"/></p>
      </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

Listing 1.6 beispiel2.xsl

3 Selbst für dieses einfache Beispiel wären bereits andere, gleichwertige Lösungen denkbar, da auch XSLT – wie jede Programmiersprache – für ein Problem unterschiedliche Ansätze ermöglicht.

Das Stylesheet wurde um ein weiteres Literal Result Element ergänzt, nämlich den <p>-Container, wie er im Ergebnisdokument erscheinen soll.

Nun folgt etwas, was das bisher statische Stylesheet »aktiv« werden lässt und gleichzeitig erstmals das Quelldokument in die Verarbeitung einbezieht:

Im Inneren dieses <p>-Containers steht eine **XSLT-Instruktion**. Die Anweisung <xsl:value-of select="gruss"/> extrahiert den gewünschten Inhalt aus dem XML-Dokument und fügt ihn ins Ergebnisdokument ein:

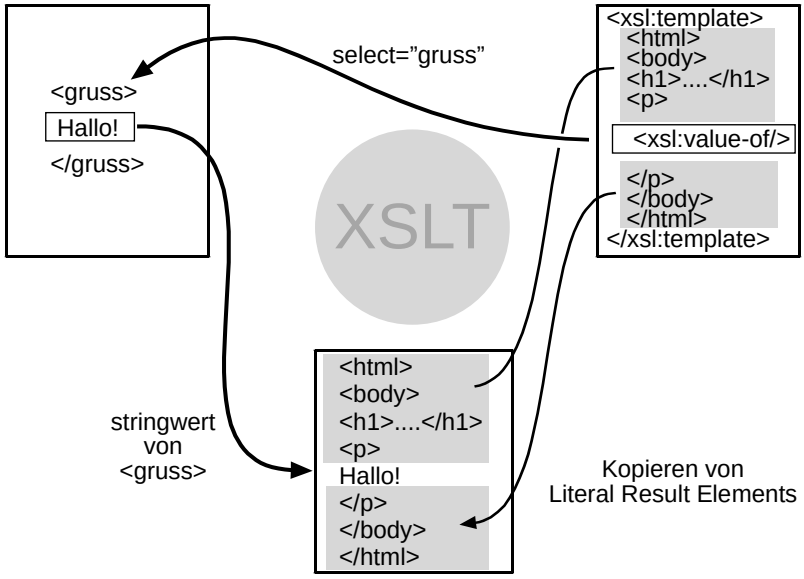

Abbildung 1.4 xsl:value-of holt sich die Information aus XML-Dokument

Neben xsl:value-of, das wie gesehen einen Befehl darstellt, erscheinen noch zwei weitere XSLT-Elemente: xsl:stylesheet, mit dem das Stylesheetdokument beginnt (der XML-Prolog wird der Einfachheit halber außer Acht gelassen), und xsl:template, welches eine Hierarchieebene weiter innen steht und in dessen Inneren sich die Literal Result Elements befinden.

Auch an dieser Stelle ein kleiner vorläufiger Lehrsatz: Im Inneren der Templateelemente eines XSLT-Stylesheets ist es erlaubt zu mischen: Literal Result Elements, die unverändert kopiert werden (wie hier der <p>-Container), und XSLT-Instruktionen dürfen nebeneinander vorkommen.

Hinweis: Alle Codebeispiele, zu denen in der Unterschrift ein Dateiname genannt wird, befinden sich auf der Begleit-CD.

1.3 Einteilung der Elemente von XSLT

Es wurde in den vorangegangenen kurzen Beispielen bereits ansatzweise deutlich, dass Position und Auftreten der XSLT-Anweisungen im Stylesheet festen Regeln unterliegen, sie in diesem Sinne also nicht zueinander gleichrangig sind. Man teilt sie ihrer Stellung in der Stylesheet-Hierarchie entsprechend in drei **Gruppen** ein:

- **Root-Elemente** (dienen als Wurzelelement des Stylesheets)
  Die einzigen beiden XSLT-Elemente, die alternativ zueinander als Wurzelelement des Stylesheets eingesetzt werden können, sind `xsl:stylesheet` und sein selten verwendetes Synonym `xsl:transform`. Sie bilden daher eine eigene, zwei Elemente umfassende Gruppe.
- **Toplevel-Elemente** (Deklarationen und Strukturelemente)
  Unmittelbar dem Wurzelelement untergeordnet, definieren sie die globale Struktur des Stylesheets (Templates, Deklarationen von Funktionen, Parametern, Formatierungsregeln und Weiteres). In XSLT 2.0 existieren rund 16 Toplevel-Elemente.
- **Instruktionen** (Strukturelementen untergeordnete Anweisungen)
  Unterhalb der globalen Ebene auftretende Anweisungen (Subdeklarationen, Templateaufrufe, Bedingungen, Schleifen, Konstruktoren etc.). Dieser Gruppe werden in XSLT 2.0 rund 31 Elemente zugeordnet.

Das hier vorgestellte, kurze Stylesheet enthält einen exemplarischen Vertreter aus jeder dieser drei Gruppen: das Root-Element `xsl:stylesheet`, das Toplevel-Element `xsl:template` und die Instruktion `xsl:value-of`.

Diese drei sollen gleich betrachtet werden, die restlichen Elemente werden nach und nach erarbeitet. Eine vollständige Übersicht aller XSLT-Elemente von XSLT 1.0 und XSLT 2.0 befindet sich weiter hinten im Referenzteil zu XSLT.

1.4 Das Wurzelelement xsl:stylesheet

Nun soll das Stylesheet etwas im Detail betrachtet werden. Zunächst handelt es sich, wie man am XML-Prolog erkennt, offenkundig um ein **XML-Dokument**:

```
<?xml version="1.0"?>
```

Das `version`-Attribut des Prologs bezeichnet den Versionsstand des im Dokument verwendeten XML. Es besitzt den Wert `1.0`, wie derzeit gefordert.[4]

4 Für Dokumente gemäß XML 1.1 müsste hier entsprechend der Wert »`1.1`« stehen. Wenn Sie mehr über XML wissen wollen, möchte ich Ihnen an dieser Stelle »Einstieg in XML, 2.Auflage« von Helmut Vonhoegen, Galileo 2004 empfehlen.

Zusätzlich enthält der Prolog meist noch ein `encoding`-Attribut, um beispielsweise im Zeichensatz des Dokuments Umlaute verwenden zu können – der Einfachheit halber wird hier darauf verzichtet.

1.4.1 Das Wurzelelement und seine Namensräume

Nach dem XML-Prolog folgt das Wurzelelement des Stylesheet-Dokuments: `xsl:stylesheet`. Auch `xsl:stylesheet` enthält ein `version`-Attribut, das hier jedoch den Versionsstand der im Stylesheet verwendeten XSLT-Version[5] nennt.

Für **XSLT 2.0** besitzt es daher den Wert `2.0`:

```
<xsl:stylesheet version="2.0" ... >
```

Das andere »Attribut« `xmlns:xsl` ist eigentlich keines (man bezeichnet es auch als Pseudoattribut), sondern die Deklaration eines so genannten **Namensraums**. Das Partikel nach dem Doppelpunkt ist frei wählbar und dient als **Präfix** für Elemente, denen der deklarierte Namensraum zugeordnet wird. Für XSLT hat es sich eingebürgert, hier `xsl:` zu verwenden:

```
<xsl:stylesheet version="2.0" xmlns:xsl="...">
```

Zu beachten: Auch das Wurzelelement `xsl:stylesheet` verwendet das Präfix `xsl:`, obwohl es in ihm erst vereinbart wird. Dies ist korrekt; ein Namensraum gilt überall *in* dem Element, für das er vereinbart wurde, und auch *für* das Element selbst.

1.4.2 Der Sinn eines Namensraums

Ein Namensraum dient dazu, ein XML-Element **unmissverständlich** einer Gruppe zuzuordnen. Dies beugt zum einen Problemen mit Namensdoubletten bei gleichzeitiger Verarbeitung mehrerer XML-Dokumente unterschiedlicher Herkunft vor. Zum anderen können so Schlüsselkonzepte bestimmter XML-Anwendungen als einem **Bedeutungsraum** zugehörig gekennzeichnet werden, wie XML Schema, Formatting Objects, SVG, MathML – oder eben XSLT.

Verwendet wird dafür (vergleichbar mit der Kennziffer eines Personalausweises) eine unverwechselbare **Zeichenkette**. Weil eben diese Unverwechselbarkeit am einfachsten mit (per Definition einzigartigen) URI-Strings zu erreichen ist, setzt man Webadressen zur Kennzeichnung von Namensräumen ein[6] – deshalb spricht man auch von **Namensraum-URI**s:

5 Einzig weiterer erlaubter Wert ist derzeit `"1.0"` für XSLT 1.0. Verschiedene XSLT-Prozessoren erkennen und verarbeiten allerdings auch – eher aus historischen Gründen – den Wert `"1.1"` für die offiziell zurückgezogene XSLT-Version 1.1.

```
<xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Für XSLT ist dieser URI **reglementiert**; er muss (kategorisch!) der Zeichenkette `http://www.w3.org/1999/XSL/Transform` entsprechen.[7] Nur so wird ein Element, das »in« diesem Namensraum ist, für die verarbeitende Anwendung, den **XSLT-Prozessor**, eindeutig als XSLT-Instruktion gekennzeichnet.

Das Präfix kann als Platzhalter bzw. Abkürzung für diesen URI-String verstanden werden – man könnte daher, rein theoretisch, anstelle von `xsl:` auch ein beliebig anders lautendes Präfix verwenden.

Unterscheidung zwischen XSLT 1.0 und 2.0:
Der Namensraum-URI http://www.w3.org/1999/XSL/Transform ist für XSLT 1.0 und XSLT 2.0 identisch. Beide Sprachversionen werden nur anhand des Wertes des `version`-Attributs von `xsl:stylesheet` unterschieden. Folglich wird mit diesem Attribut also auch indirekt festgelegt, welche XPath-Version innerhalb des Stylesheets genutzt wird.

Der Gesamtbezeichner des Wurzelelements `xsl:stylesheet` ist also im Grunde zusammengesetzt und besteht aus dem **lokalen Teil** `stylesheet` und dem **Präfixteil** `xsl:` – zusammen bilden beide Teile einen so genannten qualifizierten Namen für das Element: einen **QName**

Intern arbeitet eine Anwendung immer mit dem so genannten **expandierten QName**, d.h., ausschlaggebend ist nie das Präfix selbst, sondern der Namensraum-URI, für den es steht. Zwei qualifizierte Bezeichner gelten also als gleich, wenn sie den gleichen lokalen Bezeichner und Namensraum-URI besitzen; das im nicht expandierten QName verwendete Präfix ist irrelevant.[8]

Sollte ein und derselbe Namensraum-URI gleichzeitig an zwei verschiedene Präfixe gebunden sein (denkbar z.B. im Fall zweier gleichzeitig verarbeiteter Dokumente), können für den Prozessor auch zwei, auf den ersten Blick unterschiedlich erscheinende QNames als gleich gelten.

6 Es ist eine verbreitetete Fehlannahme, dass dieser URI auf eine reale Webressource zeigen müsste. Dies ist nicht der Fall; in der Tat kommt es nur auf die unverwechselbare Zeichenkette als solche an (eine Transformation setzt keine Onlineverbindung voraus).

7 Der alte, von Microsoft in einer Übergangszeit verwendete URI `http://www.w3.org/TR/WD-xsl` wird von aktuellen Prozessoren nicht erkannt – inzwischen besteht über den zu verwendenden URI jedoch Einigkeit.

8 Auch wenn daher XSLT-Instruktionen bei entsprechender Namensraum-Deklaration mit einem anderen Präfix als »`xsl:`« versehen werden könnten, unterlässt man dies aus Gründen der Deutlichkeit (für menschliche Leser). Einer verarbeitenden Anwendung wäre es jedoch egal.

Beispiel für expandierten QName:

- Der nicht expandierte QName beispielsweise des Stylesheetelements ist `xsl:stylesheet`. In dieser Form steht der QName im Quellcode.
- Der expandierte QName, mit dem intern gearbeitet wird, lautet jedoch `{http://www.w3.org/1999/XSL/Transform}:stylesheet`.
- Dies bedeutet, dass der Prozessor ein in einem QName verwendetes Präfix bei der Verarbeitung im Grunde nicht sieht, sehr wohl aber kennt – dies ist für die Rückwandlung eines Namensraum-URIs in ein Präfix bei der Serialisierung des Ergebnisdokuments von Bedeutung (siehe Anhang C, *Glossar*, Namespace-FixUp).

Neben dem hier vorgestellten, obligatorischen XSLT-Namensraum können in `xsl:stylesheet` auch weitere Namensräume deklariert werden (dann mit anderem Präfix und URI). Für das Element existieren noch weitere, optionale Attribute, die zu diesem Zeitpunkt jedoch noch nicht von Interesse sind.

Merkenswertes zu xsl:stylesheet:

- Als Wurzelelement des Stylesheets wird entweder `xsl:stylesheet` oder (selten, aber funktionsgleich) `xsl:transform` verwendet.
- Im Wurzelelement muss die XSLT-Version festgelegt und müssen Namensräume vereinbart werden; darunter mindestens der für XSLT selbst.
- Hier vereinbarte Namensräume sind überall im Inneren des Stylesheets verfügbar; Elemente, für die sie gelten sollen, bekommen zur Kennzeichnung das in der Namensraumdeklaration vereinbarte Präfix vorangestellt.
- Durch den XSLT-Namensraum werden die XSLT-Instruktionen innerhalb des Stylesheets gekennzeichnet (dies gilt auch für das Wurzelelement selbst, das ebenfalls das Präfix `xsl:` trägt).

Gehen wir nun im Stylesheet eine Ebene weiter nach innen.

1.5 Die Grundstruktur des Stylesheets: xsl:template

Der Inhalt von `xsl:stylesheet` bildet den Rumpf des »eigentlichen« XSLT-Stylesheet. Unmittelbar im Inneren des Wurzelelements befinden sich weitere XSLT-Anweisungen, die, weil sie hier sozusagen auf oberstem Niveau im Stylesheet stehen, als **Toplevel-Elemente** bezeichnet werden.

In diesem Beispiel wird nur ein Toplevel-Element verwendet, dafür aber das wohl wichtigste: `xsl:template`

Das Element `xsl:template` bildet zusammen mit seinem jeweiligen Inhalt die entscheidende Grundstruktur eines Stylesheets, ein so genanntes **Template**.

bekannt sein, **dass** es ein Element namens `<gruss>` gibt, und **wo** es sich im Quelldokument befindet. Dies ist weniger trivial, als es im Augenblick erscheinen mag.

1.7.2 XPath-Ausdruck vs. XPath-Pattern

Als **XPath-Ausdruck** bzw. **Expression** wird im Grunde alles bezeichnet, was mit Hilfe von XPath formulierbar ist – die »Expression« bildet also den allgemeinen Oberbegriff; das Pattern ist eine spezielle Untergruppe der Expression.

Eine Expression kann also einerseits in Form eines Pfadausdrucks dazu dienen, einen Teil des Quelldokuments zu bezeichnen. Sie kann aber auch Funktionsaufrufe der Form `funktionsname()` beinhalten, die einen Wert zurückgeben, der dann der aufrufenden Instruktion übergeben wird. In diesem Zusammenhang kann ein XPath-Ausdruck auch Schleifen mit `for` und Bedingungen mit `if` enthalten.

XPath-Ausdrücke werden in den folgenden Kapiteln ausführlicher thematisiert. Vorläufig soll die Betrachtung ihrer einfacheren Untergruppe genügen: der Pfadausdrücke. Vor deren eingehenderen Betrachtung ist zunächst ein weiterer Blick auf XML-Dokumente unter einem ganz anderen Aspekt sinnvoll.

1.8 Das XML-Dokument als Baum

Um den eigentlichen Ablauf einer Transformation und in diesem Zusammenhang die Bedeutung von XPath und Pfadausdrücken besser zu visualisieren, soll der ganze Vorgang nun aus einem neuen Blickwinkel betrachtet werden: dem der Baumstrukturen.

1.8.1 Warum Baumstrukturen?

Ein Dokument als Baumstruktur zu betrachten, mutet zwar abstrakt an, erleichtert jedoch das Verständnis im Fall von XSLT und XPath erheblich:

- Für die hierarchische Struktur eines XML-Dokuments stellt ein **Baum** die ideale Metapher dar. Ein Baum besitzt genau eine Wurzel, ein XML-Dokument genau ein Wurzelelement. Die diesem untergeordneten Elemente nebst Inhalten bilden die Zweige.
- XPath wiederum beschreibt die **Bewegung** in einem solchen abstrakten Baum – dient also gewissermaßen als »Motor« von XSLT. Mittels XPath-Ausdrücken können Positionen, Relationen und Bewegungsrichtungen in einem Baum beschrieben sowie einzelne Knoten oder ganze Zweige eines Baums ausgewählt werden.

Zur Darstellung eines XML-Dokuments wird ein Baum im mathematischen Sinne verwendet. Dieser wird stets mit nach unten gerichteter Spitze dargestellt und besitzt einen einzelnen, oben liegenden Startpunkt, der als »Wurzel« bezeichnet wird. Exakt betitelt handelt es sich bei einem Baum, wie er hier vorliegt, um einen **gerichteten Graph**[17]. Er setzt sich aus **Knoten** (engl. *nodes*) und **Verbindungen** (Kanten) zusammen. Knoten, die am Ende eines Zweiges liegen, werden als Blätter (engl. *leaf-nodes*) bezeichnet.

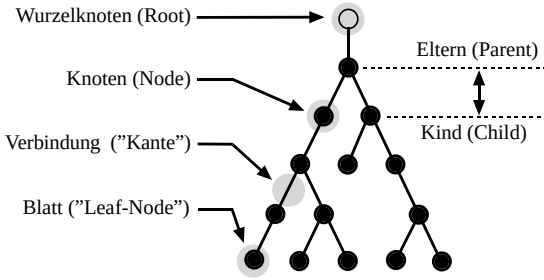

Abbildung 1.5 Dokumentstruktur als gerichteter Graph

1.8.2 Eltern und Kinder im Baumdiagramm

Vergleicht man ein XML-Dokument mit dem es repräsentierenden Baum, so sieht man, dass in den Knoten – soweit diese ein Element darstellen – jeweils **Start- und Endtag** dieses Elements zusammenfallen. Der Inhalt eines Elements wird als von diesem nach unten abgehender Zweig umgesetzt, in dem seinerseits wieder Knoten auftreten können.

Ein Knoten entspricht Start- und Endtag in Einem: Ein Elementknoten als konzeptionelle Einheit lässt die im seriellen Quelltext vorliegende räumliche Trennung von Start- und Endtag entfallen. Beide kann man so als »gleichzeitig« schreibbar verstehen; Inhalte werden an Knoten »angehängt« statt in Container »eingefügt«.

Die durch einen Zweig unmittelbar miteinander verbundenen Knoten stehen zueinander in einem »Eltern-Kind«-Verhältnis: Der enthaltene Knoten gilt als **Kindknoten** (*child*) desjenigen Knotens, von dem er abstammt; dieser wiederum gilt als dessen **Elternknoten**[18] (*parent*). Das Element `<b>` ist ein Kind-

17 Noch genauer: Es handelt sich um einen »gerichteten azyklischen Graph« (directed acyclic graph, DAG) im Sinne der Graphentheorie. Dass der Baum »azyklisch« ist, also keine Zweige in sich selbst zurücklaufen, hat den Vorteil, dass beim Traversieren des Baums keine Endlosschleifen entstehen können.

18 Ich versuche den neutralen englischen Begriff »parent« hier ebenso neutral mit »Elternknoten« zu übersetzen – üblich sind auch: Vater- bzw. Mutterknoten, Vorgänger oder übergeordneter Knoten. Im Gegensatz zur menschlichen Gesellschaft hat ein »Kind« in einer Baumstruktur allerdings niemals zwei Elternteile.

knoten von <a> und hat seinerseits zwei Kindknoten <c> und <d>. Auch die beiden Zeichenketten erscheinen im Baum als Kindknoten des jeweiligen Elements, dessen Inhalt sie bilden:

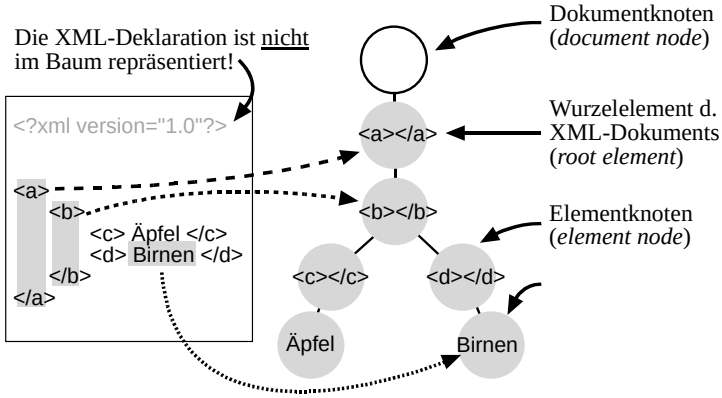

Abbildung 1.6 XML-Dokument und seine Baumentsprechung

1.8.3 Knotengattungen im Dokumentbaum

Eine Besonderheit fällt an dem in Abbildung 1.6 gezeigten Dokumentbaum auf: Seine Wurzel besteht aus einem Knoten (symbolisiert durch einen leeren Kreis), für den es im XML-Dokument **keine Entsprechung** gibt – er repräsentiert auch nicht, wie man vielleicht meinen könnte, die XML-Deklaration!

Man kann, etwas lässig formuliert, sagen, dass dieser Knoten für das Dokument »an sich« steht; das Dokument ist in ihm vollständig enthalten bzw. verkapselt. Dieser Knoten wird daher als **Dokumentknoten** (*document node*) bezeichnet. Bei diesem Knoten setzt stets der Verarbeitungsprozess an, nicht etwa bei dem ihm stets untergeordneten eigentlichen Wurzelelement.[19]

Direkt unterhalb des Dokumentknotens, also als dessen Kindknoten, findet man das Wurzelelement <a> des XML-Dokuments. Ein Knoten, der ein Element repräsentiert, wird als **Elementknoten** bezeichnet. Dieser kann als Inhalt wieder andere Elementknoten, jedoch auch andere Arten von Knoten enthalten.

In diesem einfachen Beispiel besteht der Inhalt der Elemente <c> und <d> nur aus Text, der im Baumdiagramm folgerichtig in Form zweier **Textknoten** erscheint. Kommentare werden durch **Kommentarknoten** vertreten, Processing-Instructions durch **PI-Knoten**.

19 Die etwas abstrakte Notwendigkeit einer solchen, außerhalb des Dokuments liegenden Instanz wird deutlicher, wenn man zulassen möchte, dass ein »Dokument« nicht strikt ein einzelnes Wurzelelement besitzen muss. Überhaupt fasst XSLT 2.0 den Begriff »Dokument« recht weit.

Nicht im Baum dargestellt[20] (aber als Information für XPath zugänglich!) werden üblicherweise die **Attribute** und **Namensräume**, für die aber dennoch entsprechende Knotengattungen existieren.

Ebenfalls **nicht** als Knoten im Baum enthalten ist die XML-Deklaration[21] des Prologs. Im Gegensatz zu Attributwerten und Namensräumen ist sie für XPath nicht zugänglich. Sie ist für uns glücklicherweise auch nicht weiter von Bedeutung; lediglich der XML-Parser verwendet sie, u.a. um zu bestimmen, auf welche Art das Dokument in Bezug auf das Zeichenencoding zu lesen ist.

Wir unterscheiden in XPath **sieben verschiedene Knotengattungen**, anhand der unterschiedlichen in XML-Dokumenten auftretenden Arten von Informationseinheiten. Alle besitzen jeweils individuelle Eigenschaften und werden im XSLT-Prozess dementsprechend unterschiedlich behandelt:

- Dokumentknoten
- Elementknoten
- Textknoten
- Kommentarknoten
- Processing-Instruction-Knoten
- Attributknoten
- Namensraumknoten

Ein paar der oben aufgelisteten Knotentypen sollen nun etwas näher betrachtet werden.

1.8.4 Dokumentknoten (document node)

Der Dokumentknoten ist ein abstraktes Konzept. Es ist ein **namenloser** Knoten, dessen Inhalt das gesamte Dokument umfasst. Den Dokumentknoten mit einem XPath-Ausdruck auszuwählen, bedeutet daher, den gesamten Dokumentbaum auszuwählen.

Im Fall eines normalen wohlgeformten XML-Dokuments enthält der Dokumentknoten mindestens einen Kindknoten, nämlich das **Wurzelelement** des Dokuments. Außer diesem können auf der obersten Ebene aber auch weitere Knoten erscheinen: Besitzt das Dokument einen Prolog, der **Kommentare** oder

20 Dies liegt daran, dass der Baum strikt nach Eltern-Kind-Prinzip aufgebaut ist, Attribut- und Namensraumknoten jedoch aus bestimmten Gründen nicht als Kindknoten (Children) der Elemente gelten, denen sie zugeordnet sind.

21 Die XML-Deklaration gilt trotz identischen Aussehens nicht als Processing-Instruction! Sie daher auch nicht als solche zu behandeln, ist folgerichtig.

Prozessanweisungen enthält, so gehen diese als Geschwisterknoten dem Wurzelelementknoten voran (dies gilt jedoch nicht für die XML-Deklaration selbst).

Achtung: Es ist wichtig, den Dokumentknoten (*document node*) und das Wurzelelement (*root element*) zu unterscheiden.[22] Der Dokumentknoten ist **kein Element**. Er verfügt zwar über einem Element vergleichbare Eigenschaften (er besitzt Kindknoten), hat jedoch keinen Elternknoten, keinen individuellen Bezeichner und kann weder Attribute noch Namensräume tragen.

1.8.5 Elementknoten (element node)

Ein Element wird im Baummodell durch einen Elementknoten vertreten. Ein Elementknoten besitzt einen **Bezeichner** in Form eines QNames, dieser kann also ein Präfix mit an dieses gebundenen Namensraum-URI besitzen. Ein Elementknoten kann als einziger Knotentyp (neben dem Dokumentknoten) **Kindknoten** besitzen.

Als Kindknoten dürfen wiederum Elementknoten, aber auch Textknoten, Kommentarknoten und PI-Knoten erscheinen. Die Attribute und Namensräume eines Elements werden diesem auch in Form von entsprechenden Knoten zugeordnet, jedoch nicht ihrem Inhalt zugerechnet. Entsprechend zählen sie nicht zu den Kindknoten.

1.8.6 Textknoten (text node)

Ein Textknoten enthält ausschließlich **Zeichendaten**. Er besitzt also keine hierarchisch geordnete innere Struktur. Ein zusammenhängender Bereich aus Zeichendaten in einem Dokument wird immer durch genau einen Textknoten repräsentiert. Daraus folgt, dass in einem Baum nie zwei unmittelbar benachbarte Textknoten[23] auftreten können – diese würden zu einem Knoten zusammengefasst.

Auch **CDATA-Bereiche**, wie sie in XML-Dokumenten auftreten, werden stets durch Textknoten repräsentiert – es existiert für sie keine eigene Knotengattung

22 Verwirrenderweise wurde der »document node« früher mit »root node« bezeichnet oder einfach mit »root« (dies wieder analog zur UNIX-Pfadsyntax), während das Wurzelelement des XML-Dokuments wiederum gelegentlich »document element« genannt wird.

23 Ich möchte darauf hinweisen, dass bei der Verarbeitung auch Stringliterale »`xs:string`« entstehen können – bei diesen handelt es sich jedoch nicht um Textknoten, sondern um atomare Werte. Solche Stringliterale können in unmittelbarer Folge hintereinander auftreten und werden erst bei der Serialisierung des Ergebnisses zusammengefasst.

im Dokumentbaum. Ob es sich bei einem beliebigen Textknoten ursprünglich um einen CDATA-Bereich gehandelt hat oder nicht, lässt sich nicht mehr ermitteln, nachdem ein Dokument den XML-Parser durchlaufen hat.

In vergleichbarer Weise werden auch **Textentities** beim Parsing-Vorgang expandiert. Der Ersetzungstext nimmt die Stelle der ursprünglichen Entityreferenz ein. Auch hier lässt sich im Nachhinein nicht feststellen, ob im Dokument entsprechende Referenzen vorhanden waren.

1.8.7 Der Attributknoten (attribute node)

Die Attribute eines Elements werden durch Attributknoten vertreten, die einen Bezeichner (QName) und einen Wert besitzen. Der Elementknoten gilt zwar als Elternknoten »seines« Attributknotens, dieser wiederum aber nicht als »Kind« des Elements – man sagt vielmehr lediglich, der Attributknoten »gehört« dem Elementknoten (vergleichbar einer Eigenschaft).

Attributknoten sind daher nicht »normaler« Teil des Dokumentbaums – dessen Knoten besitzen per Definition alle eine vollständige Eltern-Kind-Relation. Ein Attributknoten ist daher umständlicher zu erreichen als die anderen Knotentypen (mit Ausnahme der Namensraumknoten).

Achtung: Ist ein Attribut über eine DTD oder ein Schema mit einem Defaultwert belegt, so ist es Sache des XML-Parsers, es einzufügen, wenn es im Quelldokument weggelassen wurde. Da ein XML-Parser aber zum Verarbeiten einer DTD oder eines Schemas nicht verpflichtet ist, kann es sein, dass für ein solches Defaultattribut kein entsprechender Knoten existiert.

1.8.8 Der Baum als Abbild eines Infosets oder PSVI

Vereinfacht wurde gesagt, dass ein XML-Dokument beim Einlesen durch den XML-Parser in eine baumähnliche Form gebracht wird. Dies ist jedoch erst die zweite bzw. dritte Stufe. Vielmehr setzt der Parsingvorgang das Dokument zunächst in eine für eine Folgeanwendung verwertbare Sammlung von Informationen um, das **Infoset**. Innerhalb des Infosets sind die Bestandteile des geparsten Dokuments als **Information-Items** vertreten – ein Element beispielsweise in Form eines Element-Information-Items.

Eigenschaften des Infosets

Teil dieses Infosets sind naheliegenderweise alle im geparsten XML-Dokument enthaltenen Informationen (soweit diese als »relevant«[24] angesehen werden).

24 Hierüber gehen die Meinungen allerdings bereits auseinander – sollen beispielsweise Kommentare als relevante Informationen betrachtet werden oder nicht?

Darüber hinaus einbezogen sind (in Form so genannter **Properties**) noch zusätzliche Informationen über die einzelnen Information-Items: Hierzu gehören dessen **Basis-URI**, **Datentyp**, **Name** (für Elemente und Attribute), Informationen über die Position im Dokument (Parent- und Child-Properties) und weitere Eigenschaften. Das Infoset selbst ist jedoch noch kein Baum – dieser und die in ihm versammelten Knoten werden aus dem Infoset erst konstruiert.

Das Post Schema Validation Infoset – PSVI

Ist für den Parser ein **XML Schema**-Dokument zugänglich und ist dieser in der Lage, es auch zu verarbeiten, so wird das Infoset validiert – mit anderen Worten wird zum einen die Gültigkeit des Dokuments geprüft und zum anderen mit den Informationen jeweils ein **Schema-Datentyp** verknüpft (*type annotation*). Hierbei kann es sich um einen Basistyp oder um einen userdefinierten Typ im Rahmen des Schemas handeln. Das Ergebnis dieser Validierung ist ein so genanntes **PSVI** (Post Schema Validation Infoset).

Konstruktion der Knoten des Baums aus dem Infoset

Ausgehend entweder von einem Infoset (ohne erfolgte Validierung des Dokuments) oder von einem PSVI (mit Validierung gemäß eines Schemas) erfolgt nun die »Konstruktion« der Knoten und des aus ihnen zusammengesetzten Baums. Der wesentliche Unterschied im Ergebnis besteht in den mit den Knoten jeweils verknüpften Datentypen.

Ist ein Knoten aus dem »gewöhnlichen« Infoset konstruiert, so ist sein Datentyp meist unspezifisch – so hat beispielwiese ein aus dem Infoset stammender Elementknoten grundsätzlich den XML Schema-Typ `xs:anyType`. Für Attributknoten ist die Lage etwas komplexer; hier kann der unspezifische XPath-Typ `xdt:untypedAtomic` zugeteilt werden, jedoch auch ein anderer (wenn z. B. das Attribut in einer DTD als ID, IDREF, NMTOKEN o. Ä. deklariert wurde).

Liegt dem Baum ein PSVI zugrunde, so besitzt jeder Element- und Attributknoten – sofern er validiert wurde – ein `validity`-Property. Dieses kann die Werte »valid«, »invalid« oder »notKnown« annehmen. Elementknoten erhalten, sofern sie »valid« sind, den entsprechenden Schematyp gemäß des Schemas, andernfalls `xs:anyType`. Analog wird für Attributknoten verfahren.

Weitere Informationen zum Datenmodell:
Die offizielle Beschreibung des XPath-Datenmodells sowie der Konstruktion von Knoten aus Infoset und PSVI befindet sich in der W3C-Spezifikation »XQuery 1.0 and XPath 2.0 Data Model« (**www.w3.org/TR/xpath-datamodel/**).

1.9 Eine Transformation Schritt für Schritt

Betrachtet man eine XSLT-Transformation aus der Sicht von Baumstrukturen, so arbeitet man anstelle von Dokumenten mit **Dokumentbäumen**. Auch das Stylesheet wird durch den XML-Parser eingelesen und in einen Baum umgeformt. In Bezug auf das Ergebnisdokument unterscheiden sich XSLT 1.0 und XSLT 2.0 insofern, als dass XSLT 1.0 hier von einem **Ergebnisbaum** spricht, in den unmittelbar geschrieben wird. Dieses mit »Top-Down« bezeichnete Verfahren ist daher relativ anschaulich.

XSLT 2.0 legt dagegen ein abstrakteres »Bottom-Up«-Verfahren zugrunde und betrachtet das Ergebnis zunächst nicht als Baum, sondern als Datenstrom in Form einer **Ergebnissequenz**. Diese wird allerdings im Zuge der zu Beginn der Serialisierung erfolgenden Normalisierung ebenfalls in einen Baum umgewandelt, sodass es (halbwegs) zulässig erscheint, hier das gleiche Bild anzuwenden.

Im Zentrum der XSLT-Transformation steht der Baum des Quelldokuments; genauer gesagt **steuert** die Struktur des Quelldokumentbaums sogar indirekt die Verarbeitung. Dieser Baum wird von oben her durchwandert, was auch der Leserichtung im Dokument entspricht.

1.9.1 Aktuell: Der Current Node im Quelldokument

Den Vorgang der Verarbeitung kann man als durch den Quelldokumentbaum wandernden **Cursor** visualisieren, der stets auf genau einem Knoten platziert wird: Dieser Knoten wird als »aktueller Knoten« bezeichnet, als **Current Node**.

Zu Anfang fällt der Current Node automatisch auf den **Dokumentknoten** des Quelldokuments. Das XPath-Muster, das den Dokumentknoten repräsentiert, lautet »/.« Die für ihn aktivierte Templateregel ist bekanntlich jene mit `match="/"`. Die Verarbeitung **beginnt** also stets mit dem Dokumentknoten des Quelldokuments und schreitet von da ab sukzessive in Richtung der Kindknoten voran.

Der XSLT-Prozessor beginnt also für den Current Node stets mit der Verarbeitung der Inhalte der passenden Templateregel, die dann als **aktuelle Templateregel** bezeichnet wird. So wie es immer nur genau einen aktuellen Knoten gibt, so existiert auch nur eine aktuelle Templateregel.

Mit dem aktuellen Knoten als Kontext werden durch Instruktionen in der aktuellen Templateregel mittels XPath-Ausdrücken Knotenmengen (Sequenzen) im Quelldokumentbaum ausgewählt. Diese Knotenmenge bezeichnet man als die **aktuelle Sequenz**.

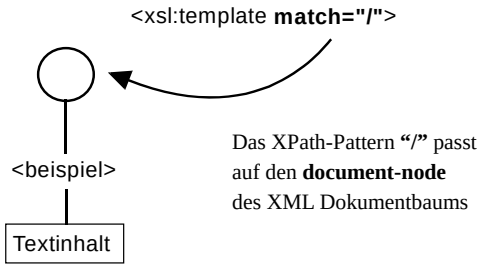

Abbildung 1.7 Das Pattern für den Dokumentknoten

Die Knoten dieser Sequenz werden nun ihrerseits der Reihenfolge nach als Current Node dem Stylesheet zur Verarbeitung »angeboten«: Der Current Node wandert weiter. Unter den vorhandenen Templateregeln wird jeweils nach derjenigen gesucht, deren Vergleichsmuster auf den aktuell angebotenen Knoten »passt« .[25] Diese Templateregel wird zur neuen aktuellen Templateregel.

1.9.2 Der fiktive Ergebnisbaum

Wir haben vorhin bereits festgestellt, dass die Templateregel für den Dokumentknoten stets genau einmal und zu Beginn der Verarbeitung ausgeführt wird: Befinden sich Literal Result Elements in dieser Templateregel, so werden die sie im **Stylesheetbaum** repräsentierenden Knoten kopiert und unmittelbar »unten« an den neu gebildeten Dokumentknoten des **Ergebnisbaums** angehängt.

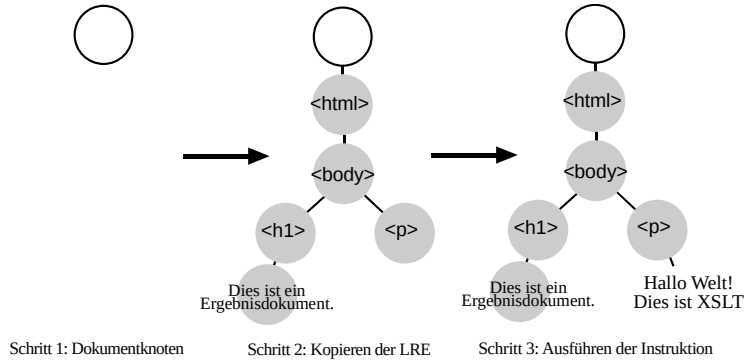

Abbildung 1.8 Schreiben des Ergebnisbaums (Gedankenmodell)

Ebenso wird in Folge mit dem Ergebnis ausgewerteter Instruktionen und weiterer ausgeführter Templateregeln verfahren, sodass der Ergebnisbaum mit fortschreitender Verarbeitung wächst. Vergleicht man diese Betrachtungsweise

25 Kommen mehrere Templateregeln in Frage, so »gewinnt« die am genauesten passende.

gedanklich mit einer Quellcodeansicht des Ergebnisses, in der die Tagcontainer quasi gleichzeitig von vorne und hinten nach innen geschrieben werden müssten, so fällt die Übersichtlichkeit der Baummetapher ins Auge.

Ist das Quelldokument komplett durchlaufen, so ist die Transformation beendet und der Ergebnisbaum vollständig. Er kann jetzt in ein wirkliches Dokument umgewandelt werden: Diesen Vorgang bezeichnet man als »Plätten« des Baums oder (die geläufigere Benennung) als dessen **Serialisierung**.

Die Realität in XSLT 2.0

Dieses so angenehm anschauliche Bild des wachsenden Ergebnisbaums mag für XSLT 1.0 gelten, für XSLT 2.0 ist es leider unkorrekt. Dort wird nicht abschnittsweise ein Baum konstruiert, sondern die Inhalte werden nacheinander in eine Ergebnissequenz ausgegeben. Diese wird nach Abschluss der Transformation serialisiert.

Die **Serialisierung** stellt einen eigenständigen, von der Transformation unabhängigen Verarbeitungsschritt dar. Bei der dabei erfolgenden Normalisierung der Sequenz entsteht (vorübergehend) entweder ein vollständiger Baum oder die Serialisierung schlägt vollständig fehl. Keinesfalls kann (z. B. durch einen vorzeitigen Abbruch) ein unfertiger Baum bzw. ein unfertiges Dokument Ergebnis einer Transformation sein.

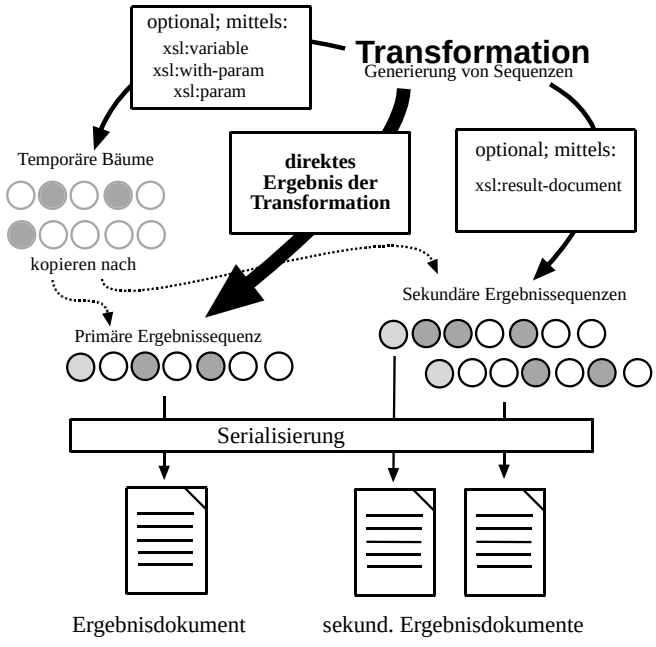

Abbildung 1.9 XSLT 2.0 – eine Vielzahl möglicher Sequenzen

Da XSLT 2.0 im Gegensatz zu XSLT 1.0 mehrere Ergebnisdokumente gleichzeitig produzieren kann, erzeugt der Prozessor gegebenenfalls auch mehrere Ergebnissequenzen. Darunter ist jedoch nur eine **primäre Ergebnissequenz**, die zum oben beschriebenen Ergebnisdokument korrespondiert. Die anderen Sequenzen werden zu sekundären Ergebnisdokumenten serialisiert – sie werden mittels der Instruktion `xsl:result-document` ins Leben gerufen.

Zusätzlich existieren quasi »frei vagabundierende« Sequenzen in Form so genannter **temporärer Bäume**[26] (diese sind in Variablen abgelegt und verfügen über eigene Dokumentknoten), die aber nicht serialisiert werden, sondern höchstens zu einem frei bestimmbaren Zeitpunkt in eine der Ergebnissequenzen kopiert werden.

Dieses komplexe Bild wird früh genug Thema werden. Zunächst soll jedoch noch bei der einfachen Konstellation geblieben werden, anhand derer Sie einen ersten Eindruck von XPath-Pfadausdrücken gewinnen.

1.10 Start mit XPath: Pfadausdrücke

XPath-Pfadausdrücke dienen, vom Stylesheet ausgehend, der Lokalisation von Teilen des Quelldokuments. Die derart durch einen Pfadausdruck beschriebenen Knoten (Elemente, Attribute, Text etc.) werden ausgewählt und können anschließend verarbeitet werden, beispielsweise – wie eben gesehen – indem ihr Textinhalt extrahiert wird.

Im Rahmen des letzten Beispielstylesheets (`beispiel2.xsl`) wurde ein Pfadausdruck vorgestellt, der einfach nur aus dem **Bezeichner** des zu adressierenden Elements bestand. Ganz so simpel ist es allerdings nicht. Um dies darzulegen, muss das vorige Beispieldokument etwas umgeschrieben werden.

1.10.1 Gewusst wo: Vom Auffinden der Inhalte

Zur Demonstration werden zwei um eine Tag-Ebene erweiterte Versionen des simplen XML-Dokuments (`beispiel2.xml`) erstellt – indem einfach ein Tag `<hallo>` hinzugefügt wird, allerdings an unterschiedlichen Positionen.

In der ersten Version liegt dieser im Inneren des Elements `<gruss>`:

```
<?xml version="1.0"?>
<gruss>
   <hallo>
   Hallo Welt! Dies ist XSLT.
```

26 Die Bezeichnung »Baum« ist insofern irreführend, als dass es sich auch hier »nur« um Sequenzen handelt.

2 XSLT und XPath – Runde zwei

2 XSLT und XPath – Runde zwei

Das zweite Kapitel des Buchs schildert die Filterung von Sequenzen – entweder mittels Predicates bei ihrer Erstellung oder bei ihrer Verarbeitung durch Bedingungen. Darüber hinaus werden Konzepte zur Formulierung von Schleifen, zur Gruppierung, Sortierung, Nummerierung und zum Kopieren von Inhalten sowie zur Erzeugung von Knoten vorgestellt.

2.1 XPath – Filtern von Sequenzen

Der Verarbeitungsablauf einer XSLT-Transformation ist direkt abhängig von der Zusammensetzung der zur Verarbeitung anstehenden Sequenzen. Eine Sequenz wird, wie im vorigen Kapitel gezeigt, durch einen Pfadausdruck zusammengestellt. Dies findet in der Regel im Rahmen der Verarbeitung von Instruktionen wie `xsl:apply-templates` statt, aber auch in Instruktionen, die Sequenzen unmittelbar ausgeben, wie `xsl:value-of`.

Ein Pfadausdruck setzt sich dabei aus einer Folge von Location-Steps zusammen. Jeder Schritt erfolgt entlang einer Achsenrichtung und erzeugt eine Ergebnissequenz, deren Knoten jeweils Ausgangspunkt des folgenden Schritts werden. Hat daher der erste Schritt eine Ergebnismenge von drei Knoten, so teilt sich der Pfadausdruck für den folgenden Schritt in drei Zweige.

Der Pfadausdruck ist dabei umfassend, seine endgültige Ergebnissequenz beinhaltet daher alle Knoten der Ergebnismenge des letzten Schritts (bzw. der »Schrittinstanzen«, deren Anzahl wieder von der Ergebnismenge des Schritts davor abhängt). Es ist, da diese Menge sehr groß werden kann und auch Knoten enthalten mag, die im Endeffekt nicht verarbeitet werden, sehr wichtig, frühzeitig **Filterbedingungen** setzen zu können – und zwar bereits für jeden erfolgenden Zwischenschritt einzeln.

Das Beispieldokument – ein Buchkatalog

Das Dokument, anhand dessen in Folge einige Grundkonzepte gezeigt werden, setzt sich zusammen aus einer Reihe von `<buch>`-Elementen, die Attribute `ersch-jahr` und `kategorie`. besitzen. Das Attribut `kat` ist in der DTD mit einem Defaultwert `"edv"` versehen:

```
<!ATTLIST buch
ersch-jahr CDATA #IMPLIED
kat CDATA "edv">
```

Listing 2.1 buchhandel.dtd

Bei den meisten Buchelementen ist dieses Attribut in der Datei nicht explizit genannt, wird aber durch den XML-Parser eingefügt. Die restlichen Informationen stehen in Child-Elementen, das <preis>-Element besitzt ein Attribut waehrung, das die Währung anzeigt – Default (ebenfalls per DTD) ist "Euro". Für einige Bücher ist dieser mit dem Wert "Dollar" überschrieben.

```
<buchhandel>
... <!-- mehr Bücher -->

<buch ersch-jahr="1996">
   <buchtitel>Where Wizards Stay Up Late</buchtitel>
   <buchautor>Hafner, Katie</buchautor>
   <buchautor>Lyon, Matthew</buchautor>
   <buchverlag>Touchstone</buchverlag>
   <preis waehrung="Dollar">14.00</preis>
</buch>

<buch ersch-jahr="1950" kat="tiere">
   <buchtitel>Unter Korallen und Haien</buchtitel>
   <buchautor>Hass, Hans </buchautor>
   <buchverlag>Ozean Verlag</buchverlag>
<preis waehrung="Euro">37.95</preis>
</buch>

... <!-- mehr Bücher -->

</buchhandel>
```

Listing 2.2 buchhandel.xml

Es ist nun auf dem bereits bekannten Wege verhältnismäßig einfach, die Verarbeitung pauschal aller <buch>-Elemente des Dokuments zu veranlassen. Angenommen jedoch, dies soll genau nicht geschehen, so sollten die unerwünschten Buchdatensätze bereits bei der Zusammenstellung der Sequenz ausgefiltert werden. Dies geschieht innerhalb des Pfadausdrucks.

2.1.1 Predicates – Filterbedingungen für Location-Steps

In XPath wird eine hierfür erforderliche Filterbedingung mittels eines so genannten **Predicates** definiert (gewöhnlich etwas unschön mit »Prädikat«

übersetzt[1]). Ein solches Predicate wird, in eckige Klammern gestellt, einfach an Achsenbezeichner und Knotentest eines Schritts angehängt:

```
achsenbezeichner::knotentest[predicate]
```

Der Inhalt des Predicates kann ein beliebiger XPath-Ausdruck sein, der Bezeichner, Operatoren und Funktionsaufrufe enthält. Ein Schritt kann dabei mehrere Predicates in Folge besitzen:

```
achsenbezeichner::knotentest[predicate1][predicate2]
```

Dies kann man als eine Art Folge von Sieben verstehen, durch die die Ergebnismenge des Location-Steps gefiltert wird. Es ist wichtig, sich zu vergegenwärtigen, dass die resultierende Menge **nicht unabhängig von der Reihenfolge** der Predicates ist, zumindest nicht in jedem Falle: Stellt man sich beispielsweise eine Bedingung vor, die pauschal jedes zweite Item einer Sequenz entfernt, und eine weitere, die den verbleibenden Rest mit einem Schwellenwertvergleich filtert, so wird dieser Umstand deutlicher.

Setzt sich ein Pfadausdruck aus einer Reihe von Schritten zusammen, so kann jeder Schritt eine beliebige Menge von Predicates erhalten:

```
step1[predicate]/step2[predicate]/step3[predicate]
```

Jeder Predicate-Ausdruck wird in Zusammenhang mit einem geprüften **Kandidatenknoten** zu einem **Booleschen Wert** ausgewertet. Ist dieser `true`, so wird der geprüfte Knoten der Ergebnismenge des Schritts zugeschlagen (bzw. beibehalten), andernfalls verworfen. Die nach dem letzten Schritt verbleibende Sequenz ist die Ergebnissequenz des Pfadausdrucks.

Auf diesem Wege ist es möglich, die Zusammensetzung einer Sequenz bis ins Detail zu bestimmen. Dies soll jetzt an ein paar einfachen Beispielen gezeigt werden.

2.1.2 Die Filterung einer Sequenz

Prüfung auf Existenz eines Attributs oder Kindelements

Ein simples Predicate prüft auf das Vorhandensein eines Kindelements oder eines Attributs, ohne dessen konkreten Wert zu berücksichtigen. So wählt

```
select="beispiel[test]"
```

1 Adäquater wäre »Filterbedingung« oder »Prädikatfilter«. Ich favorisiere in diesem Fall den kürzeren englischen Ausdruck – im Sinne eines Fachausdrucks, nicht aus Hang zur Anglisierung dieses Textes.

alle Elementknoten <beispiel>, die ein Kindelement <test> besitzen, unabhängig von dessen Inhalt. Analog ist dies für Attributknoten möglich:

```
<xsl:apply-templates select="buch[@kat]"/>
```

wählt alle Bücher in unserem Beispiel, die über ein kat-Attribut verfügen. In diesem Falle nützt dies als Filterbedingung konkret nichts, da für das Attribut ein Defaultwert vereinbart ist, und es, selbst wenn es im Quelldokument faktisch fehlt, durch den XML-Parser ergänzt wird. Da dies jedoch geschieht, *bevor* der XSLT-Prozessor das Predicate prüft, ist für diesen ein real im Quelltext vorhandenes und ein virtuell eingefügtes Defaultattribut nicht unterscheidbar.

Prüfung auf Gleichheit und Ungleichheit mit konkretem Wert

Soll beispielsweise eine Liste aller Bücher erstellt werden, deren Preis in Dollar angegeben wurde, so kann dies einfach bei der Zusammenstellung der Sequenz geschehen (Current Node sei das Wurzelelement <buchhandel>):

```
<xsl:apply-templates select="buch[preis/@waehrung='Dollar']"/>
```

Das Predicate enthält einen XPath-Ausdruck, dessen Auswertung für jeden Knoten der Resultatsequenz des Steps einen Booleschen Wert ergibt: Ein Buch besitzt entweder Kindelement <preis>, das über ein waehrung-Attribut mit Wert "Dollar" verfügt – dann verbleibt es in der Sequenz –, oder das Attribut hat einen anderen Wert (bzw. existiert nicht), woraufhin der entsprechende Knoten verworfen wird. Geprüft werden aber *alle* Bücher.

Achtung: Da der gesamte XPath-Ausdruck in doppelten Anführungszeichen steht, muss als Stringbegrenzer des Attributwerts innerhalb des Predicates das einfache Anführungszeichen verwendet werden:

```
buch[buchverlag='Ozean Verlag']
```

Es lassen sich ohne weiteres Filter kombinieren. Wir können aus dem EDV-Sektor nur die Bücher übernehmen, die nicht von Kati Hafner[2] stammen. Hierfür fügen wir einfach ein weiteres Predicate hinzu:

```
<xsl:apply-templates
 select= "buch[@kat='edv'][buchautor!='Kati Hafner']"/>
```

2 Katie Hafner, Matthew Lyon: »Where Wizards Stay Up Late«, dt. »Arpa Kadabra«; dpunkt-Verlag, 1997. Lesenswerte und historisch korrekte Darstellung der technischen Geschichte des Internets.

Triviale Prüfung auf Position in der Sequenz

Die Items einer Sequenz sind ihrer Reihenfolge entsprechend durchnummeriert, also über einen Index ansprechbar bzw. beschreibbar. Der Index beginnt immer bei 1 für das erste Item. Eine **Indexziffer** kann direkt als Filterkriterium verwendet werden – hierfür wird die entsprechende **Ganzzahl** als Predicate-Ausdruck eingesetzt:

```
buch[1]
```

lässt aus der Sequenz aller Buchelemente, die der Knotentest zusammenstellt, nur das erste Item passieren, in diesem Fall also das erste Buch in Dokumentreihenfolge. Ein solcher Ausdruck ist auch im Inneren eines anderen Predicate-Ausdrucks möglich: Sollten Sie beispielsweise den Namensvergleich auf das erste <buchautor>-Element beschränken müssen (ein möglicher Grund hierfür wird weiter unter gezeigt), so können Sie

```
buch[buchautor[1]='Lyon, Matthew']
```

schreiben, was nur die Bücher durchlässt, für die Matthew Lyon im jeweils ersten <buchautor>-Element genannt wird (im betrachteten Fall ist er Co-Autor eines Buchs, also an zweiter Stelle erwähnt – die Ergebnissequenz ist also leer).

Bei dieser Form des Predicates darf nur genau eine Ganzzahl `xs:integer` übergeben werden. Der Versuch, mit Hilfe zweier gleichzeitig übergebener Indexzahlen – etwa in der Form `beispiel[1 3]` – mehrere konkrete Items zu wählen, scheitert (Syntaxfehler). Ebensowenig funktioniert die Übergabe einer Sequenz aus Ganzzahlen, wie sie mit einem Ausdruck `2 to 5` erzeugt würde: Dieser Ausdruck ergibt in der Tat die Sequenz `(2, 3, 4, 5)`, leider jedoch wählt `beispiel[2 to 5]` nicht das »zweite bis fünfte« Item aus,[3] sondern lässt alle Items passieren, ohne überhaupt zu filtern!

Der Grund besteht darin, dass eine übergebene Sequenz, sobald sie nicht leer ist, stets den Booleschen Wert `true` ergibt. Einen Syntaxfehler stellt dies nicht dar – die Übergabe einer Sequenz ist prinzipiell erlaubt, lediglich wenig sinnvoll …

Wertvergleich – kleiner und größer

In einem Predicate kann auch ein Vergleich auf »größer als« oder »kleiner als« stattfinden. Die Operatorsymbole > und < können allerdings im Stylesheet nicht direkt eingesetzt werden, da der XML-Parser diese als Elementbegrenzer missverstehen würde. Man verwendet deshalb die Entities `>` und `<`.

3 Eine Lösung hierfür wird weiter unten gezeigt.

Der XML-Parser ersetzt die Entities, bevor der XSLT-Prozessor sich an die Arbeit macht – dieser findet daher die richtigen Symbole vor.

Sollen nur vor 2000 erschienene Bücher ausgegeben werden, so schreibt man:

```
select="buch[@ersch-jahr &lt; '2000']"
```

2.1.3 Logische Verknüpfung von Bedingungen mit AND und OR

Innerhalb eines Predicates können problemlos **mehrere, logisch verknüpfte Bedingungen** auftreten, die dann gleichzeitig AND oder alternativ OR erfüllt sein müssen. Die beiden XPath-Ausdrücke, die jeweils für sich einen Booleschen Wert ergeben müssen, werden in der eckigen Prädikatsklammer durch die Token `and` bzw. `or` verknüpft. Da die Präzedenz der **logischen Operatoren** gegenüber anderen Operatoren (Rechenoperatoren, Vergleichsoperatoren etc.) niedriger ist, müssen die verknüpften Ausdrücke nicht geklammert werden.

Vereinigungsmenge – einschließendes Oder

Will ich die Autoren 'Hafner, Katie' *und* 'Hass, Hans' sehen, so muss ein Filterprädikat mit OR-Relation gebildet werden, das beide Autoren passieren lässt:

```
buch[buchautor='Hafner, Katie' or buchautor='Hass, Hans']
```

Dies würde Bücher auswählen, die beide – einzeln oder auch gemeinsam – als Autoren besitzen. Es müssen nicht ähnliche Bedingungen sein, denkbar wäre z. B. auch

```
buch[buchautor='Panter, Peter' or @kat='reisen']
```

Dies entspricht der **Vereinigungsmenge** aller Bücher des Autors Peter Panter mit allen Büchern der Kategorie »Reisen«. Hierbei sind Buchelemente, die beide Bedingungen erfüllen, jedoch nicht doppelt in der Ergebnissequenz enthalten (obwohl Knoten durchaus doppelt in Sequenzen auftreten dürften).

Schnittmenge – ausschließendes Und

Sollen beide Bedingungen gleichzeitig gelten, so verwendet man die AND-Relation. Dies entspricht im Wesentlichen vom Ergebnis zwei unabhängigen Predicates: In der Ergebnissequenz befinden sich nur diejenigen Nodes, die beide Bedingungen erfüllen.

```
[buchautor='Hafner, Katie' and buchautor='Lyon, Matthew']
```

Dies kann nur der Fall sein, wenn ein Buch mindestens zwei `<buchautor>`-Elemente besitzt und unter diesen die beiden getesteten Werte auftreten.

Operatoren in Predicates

Hier noch einmal eine kurze Auflistung der möglichen Operatoren, die bei der Formulierung von Predicates verwendet werden können. Die Reihenfolge von oben nach unten entspricht der oben angesprochenen **Operatorpräzedenz**.

Test/Aufgabe	Operator
Unäres Minus (Vorzeichen)	`-`
Multiplikative Operatoren	`*`, `div`, `idiv`, `mod`
Additive Operatoren	`+`, `-`
Vergleichsoperatoren (<, > zu ersetzen durch `>` und `<`)	`=`, `!=`, <, >, <=, >=, `eq`, `ne`, `lt`, `gt`, `le`, `ge`, `is`, `isnot`
Bedingung, additiv	`and`
Bedingung, alternativ	`or`

Tabelle 2.1 Operatoren in Predicates und ihre Präzedenz

2.1.4 XPath-Funktionen in Predicate-Ausdrücken

Die zur Bildung von Predicates eingesetzten XPath-Ausdrücke erlauben auch den Einsatz von **Funktionen** zur Formulierung der Bedingung. In erster Linie von Interesse ist hier die Teilmenge von Funktionen, die Auskunft über den aktuellen Kontext geben kann – hier als **Kontextfunktionen** bezeichnet. Aber auch alle anderen Funktionen aus XPath und XSLT dürfen in diesem Zusammenhang eingesetzt werden, wobei einige der **Stringfunktionen** und **Sequenzfunktionen** von vorrangiger Bedeutung sind.

XPath-Funktionen – Streiflicht auf die Syntax

Hier soll zunächst ein kurzer Überblick über die Syntax eines Funktionsaufrufs gegeben werden. Wie sieht eine solche Funktion aus?

Ganz allgemein verwendet man für XPath-Funktionen innerhalb von XPath-Ausdrücken folgende Syntax:

```
funktionsname()
```

Ähnlich wie in den meisten anderen Programmiersprachen auch, steht hinter dem Funktionsnamen eine Argumentklammer – diese kann dazu dienen, Eingaben entgegenzunehmen, die verarbeitet werden:

```
funktionsname(argument1, argument2)
```

Jede Funktion hat einen Rückgabewert, den der Prozessor nach der Verarbeitung durch den Rückgabewert (das Ergebnis) ersetzt. Der Funktionsaufruf ist also eine Art von Platzhalter für seinen Rückgabewert.

```
<xsl:value-of select="funktionsaufruf()"/>
```

entspricht daher

```
<xsl:value-of select="rückgabewert"/>
```

wobei dieser Rückgabewert ein atomarer Wert, ein Knoten oder eine Sequenz sein kann – gelegentlich gibt eine Funktion auch eine leere Sequenz bzw. einen leeren String zurück, was beides im jeweiligen Kontext einen neutralen Wert bedeutet (der also keinesfalls Schaden anrichtet).

Der zum Aufruf verwendete Funktionsname kann **qualifiziert** sein, d.h. mit einem Namensraumpräfix verwendet werden. Für XPath-Funktionen verwendet man das Präfix `fn:`, und zwar mit folgender Namensraumdeklaration:[4]

```
xmlns:fn="http://www.w3.org/2003/11/xquery-functions"
```

In der Regel werden XSLT-Prozessoren die XPath-Funktionen auch an ihrem unqualifiziertem Bezeichner erkennen. Sofern Sie die Funktionsnamen mit Präfix verwenden möchten, ist es jedoch obligatorisch, den entsprechenden Namensraum im Stylesheet zu deklarieren! In Folge wird zur Erleichterung der Lesbarkeit auf das Präfix verzichtet.

Zu beachten, wenn auch im aktuellen Zusammenhang noch nicht von Bedeutung, ist die **Typbindung** von Funktionsargumenten: Funktionen erwarten in XPath 2.0 Eingangswerte eines bestimmten Typs und melden einen Typfehler, wenn dies nicht gewährleistet ist. Eine automatische Typkonvertierung hinter den Kulissen, wie man sie von XPath 1.0 her kennt, gibt es nicht mehr. Im Gegenzug besitzt auch der Rückgabewert verlässlich einen bestimmten Datentyp.

Kontextfunktionen: position() und last(),

Hilfreich bei der Verarbeitung einer Sequenz sind Kenntnisse über die Zahl der in ihr existierenden Items und die Position des aktuell verarbeiteten Items in der Sequenz – mit anderen Worten über den Kontext. Im Fall einer Sequenz aus Elementen `<buch>` mag für die Verarbeitung von Interesse sein, wie viele Bücher ausgegeben werden sollen, und ob das aktuell ausgegebene Buch möglicherweise das letzte in der Sequenz ist.

XPath stellt hierfür zwei Funktionen zur Verfügung, die die genannten Informationen liefern können: `position()` und `last()`.

4 Dieser Namensraum-URI entspricht dem der Working Draft vom November 2003 und ist nicht als endgültig zu betrachten.

`position()`

gibt eine Ganzzahl zurück, die der Position des **aktuellen Items** in der aktuellen Sequenz entspricht.

`last()`

gibt eine Ganzzahl zurück, die der Position des **letzten Items** der Sequenz entspricht. Dies ist gleichbedeutend mit der Zahl der Items.

Beide Funktionen werden häufig innerhalb von Predicates benötigt (auch für andere Zwecke, etwa Nummerierungen, wie im Folgenden gezeigt werden wird).

Um aus der Initialsequenz alle bis auf die ersten fünf Bücher mittels eines Predicates auszufiltern, können Sie folgenden Ausdruck verwenden:

```
<xsl:apply-templates select="buch[position() &lt; 6]"/>
```

Hierbei ist die Sequenz, auf die `position()` angewendet wird, die zunächst vom Location-Step gebildete Sequenz aus allen Büchern. Von denen werden diejenigen gewählt, die das Predicate erfüllen – dies sind die ersten fünf `<buch>`-Elemente.

Um nur die letzten fünf Bücher in der Sequenz zu behalten, können Sie Folgendes schreiben:

```
<xsl:apply-templates select="buch[position() &gt; last()-6]"/>
```

Dies funktioniert, weil der Subtraktionsoperator höhere Präzedenz als der Vergleichsoperator besitzt. Eine Klammersetzung ist nicht nötig.

Um auf die oben angerissene Problemstellung mehrerer Indexzahlen zurückzukommen – mit Hilfe von `position()` sind diese einfach lösbar.

```
buch[position()=1 or position()=3]
```

wählt das erste und dritte Buch der Sequenz,

```
buch[position() &gt;1 and position() &lt; 6]
```

wählt das zweite bis fünfte Buchelement der Sequenz.

Boolesche Funktionen: not()

Die Funktion `not()` negiert ihr Eingabeargument und kann zur Formulierung von Predicates herangezogen werden. Ist das Funktionsargument `true`, so gibt die Funktion `false` zurück und umgekehrt:

```
buch[not(buchautor='Panter, Peter' or @kat='reisen')]
```

Dieser Pfadausdruck enthält nur jene Bücher, deren Autor weder Peter Panter heißt noch die in die Kategorie `'reisen'` fallen.

Die Verwendung der Funktion ist selten wirklich erforderlich; sie lässt sich oft umgehen, indem ihr Argument umgekehrt wird. Hier wäre dies

```
buch[buchautor!='Panter, Peter' and @kat!='reisen']
```

Stringfunktionen: contains(), starts-with(), ends-with()

Wenn es um Bedingungen für den Stringwert von Knoten geht, kann auf entsprechende Funktionen von XPath zurückgegriffen werden – für Predicates eignen sich vordringlich die Funktionen `contains()` und `starts-with()`, gegebenenfalls (eher selten) auch `ends-with()`, die alle zwei Stringargumente entgegennehmen und einen Booleschen Wert zurückgeben. So kann beispielsweise mittels

```
buch[contains(buchautor[1], 'Hans')]
```

eine Sequenz aus Buchelementen zusammengestellt werden, die ein <buchautor>-Kindelement besitzen, dessen Stringwert die Zeichenkette 'Hans' enthält. Dies wäre für `'Hass, Hans'` der Fall oder für `'Hansen, Carl'`.

Interessanter ist vielleicht in diesem speziellen Zusammenhang die Funktion `starts-with()`, falls nach Anfangsbuchstaben gefiltert werden soll.

```
buch[starts-with(buchautor[1], 'A')]
```

liefert Bücher aller Autoren, deren Nachname mit A beginnt (was bei der hier verwendeten Inhaltsstruktur des Buchautor-Elements funktioniert).

> **Hinweis**: Das zusätzliche innere Predicate `buchautor[1]` ist notwendig, weil die Stringfunktionen `contains()` und `starts-with()` genau einen String als erstes Argument erwarten. Ein zweites oder drittes Autorelement muss vorsichtshalber ausgefiltert werden, damit die Verarbeitung stattfinden kann.

RegEx-Funktionen: matches()

Komplexere Bedingungen erfordern die Verwendung **regulärer Ausdrücke**, die von XPath 2.0 im Rahmen mehrerer Funktionen unterstützt werden. Für Predicates bietet sich `matches()` an, das `true` zurückgibt, wenn der reguläre Ausdruck auf den untersuchten String passt:

```
buch[matches(buchautor[1], '^[A-H].*ss[a-z]+')]
```

Listing 2.3 buchhandel-regex.xsl

Selektiert die Bücher, deren erstes Autorelement einen Nachnamen (mit denen der untersuchte String beginnt) mit einem Anfangsbuchstaben von A bis H besitzt, dem irgendwo im Namen ein Doppel-S folgt, an das sich mindestens ein weiterer Buchstabe anschließt. Dies gilt für `'Assmann, Peter'`, `'Esser, Klaus'`, `'Hesse, Hermann'`, aber nicht für `'Hass, Hans'` (hier folgt dem zweiten S ein Komma, kein Buchstabe).

2.2 Bedingungen mit XSLT und XPath

Nachdem jetzt die gezielte Filterung der Zusammensetzung einer Sequenz möglich ist, besteht immer noch das Bedürfnis, die Verarbeitung ihrer Items innerhalb einer Templateregel nach bestimmten Kriterien variieren zu können. Von Programmiersprachen her bekannt sind die Entscheidungsmöglichkeiten IF-ELSE (wenn-dann-andernfalls). In XSLT-Stylesheets ergeben vergleichbare Strukturen die Möglichkeit, die Verarbeitung etwa durch die Inhalte des XML-Dokuments zu steuern (bestimmte Werte, Vorhandensein von Kindelementen, Attributen, sonstige Bedingungen).

Es wird immer ein Test mit einem XPath-Ausdruck durchgeführt, der, wie bei den soeben behandelten Predicates, einen Booleschen Wert erzeugt. Dieser Test bestimmt, wie der Entscheidungsblock verarbeitet wird. XSLT kennt zwei Grundformen: das simple `xsl:if` und das komplexere `xsl:choose`.

2.2.1 Entscheidungen mit XSLT: xsl:if

Die Instruktion `xsl:if` stellt im Grunde eine Art praktischen Schalter dar, der Teile eines Templateblocks an oder abschalten kann. Hierbei ist nur der IF-Zweig einer typischen Bedingung vorhanden. Es gibt für `xsl:if` nur die Wenn-Entscheidung, kein Andernfalls, also kein korrespondierendes `xsl:else`.

Ergibt die Auswertung des Werts des `test`-Attributs »true«, so wird der Inhalt des `xsl:if`-Containers ausgeführt, andernfalls wird er ersatzlos übersprungen.

```
<xsl:template match="buch">
<p><xsl:apply-templates/>
<xsl:if test="@isbn">
<br/>(ISBN: <xsl:value-of select="@isbn"/>)
</xsl:if>
</p>
</xsl:template>
```

Listing 2.4 buchhandel-if1.xsl

Index

L

M

N

O

P

Q

R

S

V

W

X

Z