

Einführung in XML

Lutz Wegner

Universität Gh Kassel

FB Mathematik/Informatik

D-34109 Kassel

Folienvorlage zur Vorlesung im Sommersemester 2000

Literatur

- [1] W3C. Extensible Markup Language (XML) 1.0, W3C Recommendations 1-Feb-98, <http://www.w3.org/TR/1998/REC-xml-19980210>
- [2] W3C. Document Object Model (DOM) Level 2 Specification, Version 1.0, W3C Candidate Recommendation 10 Dec. 1999, <http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210>
- [3] W3C. XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999, <http://www.w3.org/TR/xslt>
- [4] W3C. XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999, <http://www.w3.org/TR/xpath>
- [5] W3C. XML Fragment Interchange Requirements 1.0, W3C Note 23-Nov-1998, <http://www.w3.org/TR/NOTE-XML-FRAG-REQ>
- [6] W3C. Behavioral Extensions to CSS, W3C Working Draft 04 Aug 1999, <http://www.w3.org/TR/1999/WD-be-css-19990804>
- [7] Alex Homer. XML IE5 Programmer's Reference, Wrox Press, Birmingham, UK, 1999
- [8] Elliotte Rusty Harold. XML Bible, IDG Books, Foster City, CA, 1999
- [9] Robert Eckstein, XML Pocket Reference, O'Reilly, Sebastopol, CA, 1999
- [10] Serge Abiteboul, Peter Buneman, and Dan Suciu. Data on the Web - From Relations to Semistructured Data and XML, Morgan Kaufmann, San Francisco, CA, 2000
- [11] C. F. Goldfarb and P. Prescod. The XML Handbook, 2nd ed., Prentice Hall PTR, Upper Saddle River, NJ, 2000
- [12] Michael Leventhal. XSL Considered harmful, <http://xml.com/xml/pub/1999/05/xsl/xslconsidered.html>
- [13] Paul Grosso und Norman Walsh. XSL Concepts and Practical Use, Foliensatz XML Europe 2000, Paris, France, 12 June 2000 <http://www.sun.com/xml/developers/xsl>

Vorwort

In dieser zweistündigen Vorlesung sollen die Grundlagen der *eXtensible Markup Language*, die sich als Datenaustauschsprache zu etablieren beginnt, erläutert werden. Im Gegensatz zu HTML erlaubt XML die semantische Anreicherung von Dokumenten. Neben XML sollen auch die *eXtensible Stylesheet Language* (XSL), der Übersetzer XSLT und andere Komponenten besprochen werden.

Da dieses Vorlesungsangebot sehr kurzfristig für die entfallene Vorlesung Datenbanken II angeboten wird und da das Thema XML noch sehr im Fluß ist, sowohl was die Standards als auch die Software-Werkzeuge anbetrifft, wird die Besprechung zwangsläufig experimentellen Charakter haben. Die Studenten sind aufgerufen, durch Mitwirkung an kleinen Programmieraufgaben und eigenständige Recherche im WWW sich den Stoff mit zu erarbeiten. Für Korrekturen und Hinweise ist der Verfasser dankbar; für Mängel wird an dieser Stelle bereits um Verzeihung gebeten.

Auf der Basis der Bearbeitung der Projektaufgaben und einer abschließenden mündliche Prüfung kann ein Schein in Informatik erwerben werden.

In diesem Sinne: *no venture, no fun!*

Kassel, im Mai 2000

Lutz Wegner

1 Der XML Standard

1.1 Motivation

Im Vorwort zu [10] schreibt Jim Gray, eine der Vaterfiguren der Datenbankwelt: *All information is moving online*. Dies sei eine Zeit der gegenseitigen intellektuellen Befruchtung, der Kollision dreier Kulturen: *the everything-is-a-document culture, the everything-is-an-object culture, the everything-is-a-relation culture*.

XML könnte in dieser Entwicklung zum Bindeglied werden. XML ist ohne Zweifel eine *Dokumentbeschreibungssprache*, genau genommen eine Vereinfachung der Standard Generalized Markup Language (SGML), genauso wie HTML (Hypertext ML), jedoch ohne HTML's Festlegung auf die Dokumentformatierung.

Ferner ist der XML Standard eine *Beschreibungsvorschrift* für eine *Meta-Sprache*, d.h. die Markierungen (tags) müssen zwar einigen syntaktischen Regeln folgen, sind jedoch frei definierbar und damit offen für jegliche semantische Interpretation.

XML kann gleichzeitig Merkmale der Objektorientierung übernehmen, speziell ein Verhalten (behaviour) aufweisen, indem es z.B. mit Java Script, Tcl oder anderen Skriptsprachen gekoppelt wird. Z.T. wird erwartet, daß die Gestaltungskomponenten für XML, z.B. die *Extensible Stylesheet Language* (XSL), diese Aspekte berücksichtigen wird. Vermutlich ist dies

die schwächste Seite von XML und wird z.B. von Michael Leventhal kritisiert [12].

Zuletzt ist XML ein Zielformat für Datenbankausgaben. Da XML-Dokumente hierarchisch strukturiert sein können, lassen sich sogar aus „flachen“ relationalen Tabellen komplexe Objekte erstellen, in die Objektidentifikationen (OID's) und andere Typattribute eingefügt werden können. Alle großen DBMS-Hersteller werden dies unterstützen.

Zusätzlich wird wieder über Gestaltungskomponenten, wie z.B. XSL, eine weitergehende Aufbereitung der Daten möglich, inkl. Selektion, Projektion und Sortierung, die einem relationalen *View* sehr nahe kommt. Allerdings kann gefragt werden, ob dies Aufgabe einer Dokumentbeschreibungssprache ist.

Schließlich ist es möglich, in XML-Dokumenten auch wieder zu suchen, d.h. man stellt eine (XQL, XML-QL, ...) Abfrage an eine Sammlung von XML-Dokumenten, wie wenn diese eine Datenbank wären (in der Tat könnte dahinter tatsächlich ein DBMS stecken).

Um über die Möglichkeiten von XML kompetent reden zu können, sollte man sich zunächst an konkreten Beispielen einen Eindruck verschaffen. Dies ist Aufgabe der nächsten Abschnitte.

1.2 Ein erstes Beispiel

Nach alter C/Unix-Tradition beginnen wir mit einem „Hello World!“-Dokument (`hello.xml`).

```
<?xml version='1.0' standalone="yes" ?>
<greeting>Hello world!</greeting>
```

Die erste Zeile zwischen `<?xml` und `?>` ist eine *XML-Deklaration*. Sie besagt, daß das Dokument konform dem XML-Standard 1.0 gemäß aufgebaut ist und kein externes Stylesheet (XSL) und keine externe Dokument Type Deklaration (DTD) enthält. Deshalb müßten diese Angaben eigentlich im Dokument stehen, sie sind aber hier weggelassen.

Darunter steht ein *XML-Element*, bestehend aus einer *Anfangs-* und einer *Endemarkierung* (opening and closing tag). Der dazwischenstehende Text ist Teil des Elements.

Daneben sind noch sog. leere Markierungen (empty tags) möglich. Diese werden meist benutzt, wenn besondere nicht-textuelle Informationen an den Browser weitergegeben werden sollen. In `hello2.xml` haben wir eine Bildinformation in ein leeres XML-Element gepackt.

```
<?xml version='1.0' standalone="yes" ?>
<greeting>
  Hello world!
  <picture src="lutz-thumb.jpg" />
</greeting>
```

Den in der Markierung stehenden Teil nennt man ein Attribut. So könnten wir dem Element `greeting` ein Attribut beifügen, daß die verwendete Sprache für den Gruß angibt:

```
<greeting lang="english">Hello World!</greeting>
<greeting lang="german">Hallo Leute!</greeting>
```

Im übrigen enthält bereits die XML-Deklaration `<?xml ... ?>` zwei Attribute. Man erkennt daran, daß XML beide Arten von Hochkommata akzeptiert, ohne geht es allerdings nicht, auch nicht bei numerischen Werten.

Die ersten beiden Beispiele wird ein XML-fähiger Browser (z.B. Internet Explorer 5) zwar nicht anmeckern, aber andererseits auch nur im Quelltext ausgeben. Genaugenommen kommt im IE5 ein sog. Default-Stylesheet zur Anwendung, das aber nur eine Prüfung auf „well-formedness“ vornimmt und den Quelltext farblich und durch Einrücken strukturiert ausgibt, sonst aber nichts macht.

Fügen wir beide „Grüße“ mit dem Sprachattribut auf oberster Ebene in das XML-Dokument ein, gibt IE5 eine Fehlermeldung aus, weil er **ein** Wurzelelement je Dokument möchte (Baumstruktur, aber über Referenzen auch allgem. Graph).

Ändern wir das Dokument zu `hello3.xml`, ist es jetzt zwar wieder syntaktisch korrekt (well formed), die Ausgabe ist aber immer noch unbefriedigend.

```
<?xml version='1.0' standalone="yes" ?>
<greetings>
  <greeting lang="english">Hello world!</greeting>
  <greeting lang="german">Hallo Leute!</greeting>
  <picture src="lutz-thumb.jpg" />
</greetings>
```

Es wird Zeit, daß wir ein Stylesheet, also eine Anzeigevor-

schrift, und eine Dokument Type Definition (DTD) für `hello.xml` einführen.

1.3 Eine erste DTD

Die DTD entspricht einer Formvorlage für ein Textdokument in einem beliebigen Desktop Publishing System, bzw. einem Schema in einem DBMS. Sie kann in dem XML-Dokument stehen (standalone=“yes“) oder getrennt. In der Regel wird man eine getrennte Angabe (hier in `hello.dtd`) vorziehen, da diese mehrfach anwendbar wird.

```
<!-- DTD for hello xml's -->
<!ELEMENT greetings (greeting | picture)*>
<!ELEMENT greeting (#PCDATA)>
    <!ATTLIST greeting lang CDATA #IMPLIED>
<!ELEMENT picture EMPTY>
    <!ATTLIST picture src CDATA #REQUIRED>
```

Zunächst beginnt diese Definition mit einem Kommentar.

Diese werden mit `<!--` eingeleitet und enden mit `-->`.

Dazwischen darf nicht „--“ stehen. Ansonsten dürfen Kommentare an allen Stellen, außer innerhalb von Markierungen und vor den ersten Deklaration (`<? . . . >`), eingestreut werden.

Die nächste Definition eines Elements ist bereits etwas komplizierter und erinnert an einen regulären Ausdruck. In der Tat definiert sie `greetings` als eine beliebige Folge von `greeting-` **oder** `picture-`Elementen. Man beachte auch das „|“ Zeichen für die Alternative; stünde ein Komma zwischen `greeting` und `picture`, wäre dies das `greeting-Ele-`

ment gefolgt von dem `picture`-Element, also eine Pärchenbildung, die beliebig wiederholt wird.

Das Element `greeting` wiederum wird als (`#PCDATA`) definiert, was für *parsed character data* steht. Diese Definition schreibt vor, daß zwischen den Markierungen keine andere Elemente stehen dürfen. Will man gar keine Einschränkungen, schreibt man `<!ELEMENT name ANY>`.

Die Definition `<!ATTLIST Ziel_Element Attr_Name Attr_Type Default-Modifier>` müssen wir später genauer beschreiben. Wie man am Beispiel aber sieht, wird das „Sprach-Attribut“ `lang` als `PCDATA` definiert. Der Modifier `#IMPLIED` steht für optionales Auftreten, `#REQUIRED` offensichtlich für zwangsweises auftreten. Gegenwärtig prüft z.B. IE5 die Einhaltung vieler DTD-Vorschriften nicht.

Üblicherweise dürfen DTD's auch wieder andere DTD beinhalten (`include`), IE5 mag das aber nicht, InDelv's XML-Parser unterstützt dies. Auf diese und viele andere Punkte im Zusammenhang mit DTD's gehen wir später ein. Ein zu `hello.dtd` passende XML-Dokument ist `hello4.xml`.

```
<?xml version='1.0' standalone="no" ?>
<!-- Stylesheet Anbindung folgt -->
<?xml:stylesheet type="text/xsl" href="hello.xsl" ?>
<!-- DTD Anbindung folgt -->
<!DOCTYPE greetings SYSTEM "hello.dtd">
<!-- xml Elemente folgen -->
<greetings>
  <greeting lang="english">Hello world!</greeting>
  <greeting lang="german">Hallo Leute!</greeting>
  <picture src='lutz-thumb.jpg' />
</greetings>
```

```
</greetings>
```

Man beachte, wie die Angabe der passenden DTD im XML-Dokument erfolgt: als Prozessorinstruktion beginnend mit `<!DOCTYPE`. Ganz anders dagegen die Stylesheet-Anbindung. Sie ist eine echte XML-Prozessorinstruktion (XML-Deklaration).

DOCTYPE-Instruktionen können zwei Formen annehmen:

```
<!DOCTYPE Wurzel_Element SYSTEM "URI-of-DTD">  
<!DOCTYPE Wurzel_Element PUBLIC "name"  
    "URI-of-DTD">
```

SYSTEM steht dabei für private Nutzung der DTD, PUBLIC bezieht sich für öffentlich angebotene DTDs, die unter einem öffentlich bekannten Namen (*name*) verfügbar sind. Schlägt die Suche danach fehl, wird auf den angegebenen Universal Resource Indicator (URI) zurückgegriffen. Für Namen gelten spezielle Konventionen, auf die hier nicht eingegangen werden kann.

Eine generelle Kritik an DTDs ist ihre zu einfache Form, weshalb sie z.B. von Microsoft nur halbherzig unterstützt werden. Diese Kritik ist berechtigt. Zugleich ist unverständlich, warum DTDs keine gültigen XML-Dokumente sind. Im Sinn einer orthogonalen Meta-Datenverwaltung wäre dies zu fordern.

1.4 Ein erstes XSL Dokument

Ein „einfaches“ Stylesheet für die Ausgabe von `hello4.xml`

ist das folgende XML-Dokument `hello.xml`.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match=".">
    <xsl:value-of />
  </xsl:template>

</xsl:stylesheet>
```

XSL-Visualisierungen sind trickreich. Zunächst ist jede XSL-Formatvorlage ein gültiges XML-Dokument. Zum zweiten nutzen die XSL-Formatvorlagen eine Namensraumangabe, hier „`xsl:`“. Neben dem `xsl`-Namensraum kennt XSL auch noch den `fo`-Namensraum (*formatting objects*), sowie beliebig viele privat vereinbarte. Der `fo`-Namensraum und die darin angebotenen Anweisungen sind sehr mächtig und ergeben praktisch eine DTP-Oberfläche mit reichen Farb- und Graphik-elementen. Ergänzungen in Richtung XVL (Extensible Vector Language) für Vektorgraphik sind in Vorbereitung. Allerdings gibt es auch heftige Kritik am `fo`-Mechanismus, er wird z.Zt. nicht von IE5 unterstützt.

Wichtig auch: der `fo`-Namensraum wird vom XSL-Standard abgedeckt - ein praktisch unlesbares Dokument. Die hier interessierenden Teile des `xsl`-Namensraums stehen im XSLT-Standard (XSL Translations 1.0, vgl. [3]), der etwas besser verständlich ist.

In der zweiten Zeile der Formatvorlage steht demnach, von welchem URI der Namensraum `xsl` (`ns = name space`) definiert wird. Dies spielt nur eine Rolle bei unbekanntem Namensräumen.

Danach beginnt ein Teil, der sich durch Templates auszeichnet. Dies deutet darauf hin, daß XSL nicht prozedural/operational aufgebaut ist, sondern musterorientiert/deklarativ. Für jedes gelesene Element eines XML-Dokuments sucht der Parser das am besten passende Template und wendet es an. In der Regel ist der Vorgang rekursiv, speziell `<xsl:apply-templates/>` sorgt dafür, daß die unmittelbaren Kinder des gerade betrachteten Elements ebenfalls formatiert (genauer: vom Parser behandelt, „geparsed“) werden.

Weitere Punkte sind `match=" . "` was auf beliebige Knoten paßt und das XSL-Element `value-of`, das einfach den Inhalt des gegenwärtigen Elements („was zwischen den Tags steht“) ausgibt. Die Details werden wir uns erarbeiten müssen. Speziell fehlt in dem Stylesheet `hello.xsl` ein Template für das Bild. Auch sonst ist die Ausgabe im IE5 trostlos: es erscheint die Zeile

Hello world! Hallo Leute!

Eine Verbesserung der Ausgabe und ein tieferes Verständnis von XML und DTDs ist das Ziel der folgenden Kapitel.

2 XML Dokumente

Wir haben im vorherigen Kapitel einen ersten Eindruck von XML gewonnen. Wir stellten fest, daß

- XML von SGML abgeleitet ist und ebenfalls eine Metabeschreibungssprache ist (HTML dagegen ist eine Anwendung von SGML)
- eine DTD die Struktur von XML Dokumenten beschreibt
- ein XSL Dokument die Darstellung von XML Dokumenten beschreibt, wobei die Beschreibungsmethode deskriptiv/regelorientiert ist und mit Templates (Mustern) arbeitet.

In diesem Kapitel wollen wir systematischer XML und DTD kennenlernen. Wir orientieren uns am Aufbau von [9].

2.1 Kleinigkeiten

Im Laufe der Entwicklung von HTML haben sich gewisse tolerierte Vereinfachungen und Unsauberkeiten eingeschlichen. So darf man bei Tabellen das schließende Element eines Tabellenfeldes `</TD>` weglassen, allerdings auf die Gefahr hin, daß das letzte Feld nicht richtig zugeordnet wird.

Hinweis: Wir verweisen gelegentlich auf HTML, eine ganz nette kurze Übersicht (auf Englisch) ist der *Bare Bones Guide to HTML* unter <http://werbach.com/barebones/barebones.html>.

Dies ist in XML nicht zulässig. Alle Elemente müssen paar-

weise auftreten und wohlgeschachtelt (properly nested) sein, außer natürlich bei leeren Elementen wie etwa `<picture src=" . . . " />` im Kapitel 1.

Damit ist

```
<Abschnitt>
  <Para>
    Blablabla
  <Para>
    noch mehr bla
</Abschnitt>
```

ungültiges XML (es fehlen jeweils die `</Para>` Endmarken).

Genauso wird

```
<b><i>Achtung: gekauft wie gesehen</b></i>
zwar in HTML akzeptiert, aber von IE5 als XML-Dokument
angemeckert.
```

Anderherum ist ein leeres XML-Element in HTML eigentlich ein Fehler, z.B. die waagrechte Linie `<HR>` (horizontal ruler) in der „XML-Form“ `<HR />`, wird so aber von IE5 gemalt, von Netscape 4.8 jedoch nicht, Fehlermeldungen gibt es in beiden Fällen nicht.

Argumente von Attributen müssen in XML immer in Hochkommata eingeschlossen werden. HTML Browser erlauben auch das Weglassen, sofern der Wert keine Leerstellen (blanks) enthält. Es sind sowohl einzelne als auch doppelte Hochkommata erlaubt.

Zuletzt sei darauf hingewiesen, daß HTML bekanntlich nicht auf Groß- und Kleinschreibung achtet, XML dagegen *case*

sensitive ist! `<h2>Überschrift</H2>` wird deshalb als XML-Dokument zurückgewiesen, ginge in HTML aber durch.

Weiterhin ist zu beachten:

- Das XML-Dokument muß entweder eine DTD benutzen oder eine XML-Deklaration `standalone="yes"` enthalten.
- Die Markierungszeichen `<` und `&` dürfen nicht als Textzeichen auftreten sondern durch sog. Entity-Referenzen ersetzt werden. Genauso muß `>` durch `>` ersetzt werden.
- XML-Dokumente ohne DTD können nur Attribute vom Typ PCDATA haben.

Wie man also sieht, geht es bei XML nicht nur um Kleinigkeiten, sondern auch um Kleinlichkeiten. Dokumente, die korrekt geschachtelt sind und die obigen Punkte beachten, sind *wohlgeformt* (well-formed). Genügt ein Dokument auch einer zugehörigen DTD, dann nennt der Standard es *gültig* (valid).

2.2 Namensräume

Da XML eine Metasprache ist, kann man beliebige Marken (tags) erzeugen, z.B. `<HTML>` und `</HTML>` als Abkürzung für „Heim-Textilien-und-Möbel-Laden“, bzw.

```
<TABLE>
  <LEG Anzahl="4">gerade, schwarz</LEG>
  <TOP>Eiche poliert</TOP>
</TABLE>
```

für einen Tisch in dessen Prospekt. Wird den Marken eine spe-

zielle Bedeutung zugeordnet oder werden Dokumente aus verschiedenen Quellen gemischt, kann es offensichtlich zu Konflikten kommen.

Der W3C-Standard sieht deshalb sog. Namensräume (namespaces) vor, die zwar nicht Pflicht sind, aber Eindeutigkeit von Marken erzeugen. Namensräume werden als Attribute in beliebigen Anfangsmarken deklariert und gelten dann bis zur zugehörigen Endmarke.

So wird im Beispiel unten ein Namensraum `vorlesung` deklariert.

```
<?xml version="1.0" standalone="yes" ?>
<?xml:stylesheet type="text/xsl" href="hello.xsl" ?>
<beispiel
  xmlns:vorlesung=
    "http://www.informatik.uni-kassel.de/~wegner/">
  <title>Namensraum</title>
  <code>4711-9999-XYZ</code>
  <vorlesung:Anfang>
    Auch Zwerge haben mal klein angefangen
  </vorlesung:Anfang>
</beispiel>
```

Da er im Wurzelement deklariert wird, gilt er für das ganze Dokument. Im Element `Anfang` wird der Namensraum explizit angesprochen, indem man dem Tag den Qualifizierer, hier `vorlesung`, verbunden mit einem Doppelpunkt (aber ohne Leerstellen!) voranstellt.

Die Deklaration hat die Form

`xmlns: Namensraumbezeichner="URI"`,

wobei der Universal Resource Identifier nur zur eindeutigen

Identifizierung des verwendeten Namens dient, d.h. er muß nur global eindeutig sein und könnte auch durch eine email-Adresse ersetzt werden. Es ist tendenziell vorgesehen, daß Anwendungen Kontakt mit angegebenen URLs aufnehmen, um weitere Informationen zum Namensraum herunterzuladen. Dies wird aber momentan (von IE5) nicht unterstützt, ja die Gültigkeit der Angabe wird nicht einmal überprüft.

In den Beispielen in [7] und [9] werden die URLs der Verlage wrox, bzw. OReilly als Namensraum verwendet. Man kann sich vorstellen, daß diese für ihre Publikationen Standardvorgaben machen, die dann auch für die Darstellung genutzt werden.

Läßt man in der Namensraumdeklaration den Namensraumbezeichner weg, wie in

```
<beispiel
  xmlns="http://www.informatik.uni-kassel.de/~morad/">
```

dann wird ein Standardnamensraum (*default name space*) definiert, der für alle Tags ohne vorgestellte Qualifizierer (ohne qualifizierendes Präfix) gilt. Auch darf man mehrere Namensräume parallel einführen, oder geschachtelt (nested):

```
<beispiel
  xmlns:vorlesung=
  "http://www.informatik.uni-kassel.de/~wegner/"
  xmlns="http://www.informatik.uni-kassel.de/~morad/">
```

Zusätzlich kann man Defaultnamespaces ganz ausschalten, indem man das Attribut `xmlns=""` (leere URI) einsetzt, das dann bis zur passenden Endmarke einen existierenden Stan-

dardnamensraum überdeckt.

Natürlich nutzt XML selbst die Namensraumkonventionen für `xml:` und `xsl:` sowie andere Qualifizierer. Das offizielle Dokument der `w3.org` ist

<http://www.w3.org/TR/1999/REC-xml-names-19990114/>

Namensräume gelten auch für Attribute. Die folgenden Beispiele (zur Abschreckung) stammen aus dem Standard und werden im Originalkontext wiedergegeben.

5.3 Uniqueness of Attributes

In XML documents conforming to this specification, no tag may contain two attributes which:

- have identical names, or
- have qualified names with the same local part and with prefixes which have been bound to namespace names that are identical.

For example, each of the bad start-tags is illegal in the following:

```
<!-- http://www.w3.org is bound to n1 and n2 -->
<x xmlns:n1="http://www.w3.org"
  xmlns:n2="http://www.w3.org" >
  <bad a="1"      a="2" />
  <bad n1:a="1"  n2:a="2" />
</x>
```

However, each of the following is legal, the second because the default namespace does not apply to attribute names:

```
<!-- http://www.w3.org is bound to n1 and is the default -->
<x xmlns:n1="http://www.w3.org"
```

```
xmlns="http://www.w3.org" >
<good a="1"      b="2" />
<good a="1"      n1:a="2" />
</x>
```

Der Standard besagt demnach, daß der *default name space* einer Namensraumvereinbarung nicht für Attribute gilt; in [7] gibt es dazu eine widersprüchliche Aussage.

Unklar erscheint mir auch, was passiert wenn zwei Dokumente in einer geschachtelten Form gemischt werden, wobei zufällig gleiche Namensraumbezeichner verwendet werden. Offensichtlich müßten diese **vor dem Mischen** durch die URI ersetzt werden, sonst ist hinterher eine Unterscheidung nicht mehr möglich. Dieser Einwand betrifft nur das geschachtelte „Mischen“, z.B. automatisiertes Ersetzen eines Elements in Dokument *A* durch Elemente aus Dokument *B*. Beim einfachen Aneinanderfügen kann es zu keiner Überdeckung der Namensräume kommen.

Anmerkung: generell wird hier mit „universellen Objektidentifikatoren (OIDs)“ gearbeitet, die aus URLs (Internetadressen) abgeleitet werden und lokal durch Pseudonyme ersetzt werden. Eine ähnliche Idee hatten wir im Datenbankeditor ESCHER für Links vorgesehen.

Wie in Kapitel 1 angedeutet, ist eine XML-eigene Anwendung der $\text{f}\circ$: Namensraum, der sog. *formatting objects* definiert. Dieser wird nicht von IE5 unterstützt.

2.3 XML Instruktionen

Die folgenden XML Instruktionen sind legal.

<?xml ... ?>

```
<?xml version="number" [encoding="encoding"]  
[standalone="yes" | "no" ] ?>
```

XML-Deklarationen fangen mit den Zeichen <? an und werden mit ?> begrenzt. Üblicherweise beginnen XML-Dokumente mit obiger Deklarationsart.

Die Deklaration muß das Versionsattribut enthalten. Z.Zt ist Version 1.0 eine gültige Angabe (und IE5 akzeptiert nur die Angabe "1.0", da kennt der Browser keine Gnade :-).

Die Zeichencode-Angabe für den Dokumentzeichensatz ist optional, eine mögliche Angabe wäre "US-ASCII" oder "iso-8859-1". Ohne geeignete Angabe und mit Umlauten, wird das Dokument zurückgewiesen (Fehlermeldung: ungültiges Zeichen), mit "iso-8859-1" nicht.

Ist die optionale standalone-Angabe auf "no" gesetzt, muß eine XML <!DOCTYPE> Instruktion folgen.

<!DOCTYPE>

```
<!DOCTYPE root-element SYSTEM | PUBLIC [name] URI-of-DTD>
```

Hiermit wird eine DTD (Dokument Type Definition) deklariert. Die Instruktion kann gegenwärtig die folgenden zwei Formen annehmen.

```
<!DOCTYPE root-element SYSTEM URI-of-DTD>  
<!DOCTYPE root-element PUBLIC [name] URI-of-DTD>
```

Im ersten Fall handelt es sich um eine private Deklaration der DTD, die auf alle Elemente innerhalb des Wurzelements angewandt wird. Ein Beispiel wäre

```
<!DOCTYPE <Book> SYSTEM  
"http://uni-kassel.de/fb17/dtds/meinbuch.dtd">
```

Im zweiten Fall handelt es sich um einen öffentlich zugängliche DTD, die entweder als registrierter Name in einem internen oder externen Repository bekannt ist, oder falls da nicht gefunden, über einen URL gesucht wird. Die Namenskonvention für „öffentliche Identifier“ benutzt einen Namen mit 4 Segmenten, die durch Doppelschrägstriche (slashes) getrennt sind, z.B.

```
"-//O'Reilly//DTD//EN"
```

Das Minus deutet auf einen unregistrierten Namen hin, der ggf. nicht eindeutig ist. Ein Plus (+) wäre z.B. ein beim W3C registrierter Eintrag. Das zweite Segment ist der Autor oder die Organisation, die den Namen veröffentlicht, das dritte Segment die Inhaltsform (hier DTD, [7] gibt ein Beispiel mit `TEXT booklist`, d.h. `TEXT` ist die Form und `booklist` der Dokumentname), zuletzt folgt der Sprachcode, hier `EN` für Englisch.

```
<? ... ?>
```

```
<?target attribute1="value2" attribute2="value2" ...?>
```

Mit Verarbeitungsinstruktionen lassen sich Zielanwendungen

(targets) mit Verarbeitungshinweisen versehen. Das Format ist bis auf `<? und ?>` frei, darf aber nicht nur `xml` lauten, da dies für die XML-Deklaration vereinbart ist.

Beispiele sind die Hinweise für die Verwendung der Stylesheets, etwa

```
<?xml-stylesheet type="text/css" href="filename.css"?>
```

Unbekannte Instruktionen ignoriert IE5, Stylesheet-Angaben mit unbekanntem Ziel (`href="aberhallo.xsl"` statt `"hallo.xsl"`) werden entdeckt und bedingen einen Fehlerabbruch des XML-Parsers.

<![CDATA[...]]>

Mit dieser sog. Charakter-Data Sektion (nicht zu verwechseln mit PCDATA-Angabe in DTDs!) lassen sich Zeichenfolgen angeben, die der Parser **nicht** interpretiert. Die Folge muß mit `<![CDATA[anfangen und mit]]>` aufhören (vgl. unzuläßige Zeichenfolgen oben). Als Beispiel könnte man schreiben

```
<![CDATA[ In diesem xml-Kapitel besprechen wir
"Deklarationen", z.B. <?xml ...?> und <!DOCTYPE ...>,
CDATA[...] -Sektionen & vieles mehr
]]>
```

Andererseits darf die CDATA-Sektion keine Entity-Referenzen, das sind Folgen wie

```
&amp;
&lt;
&gt;
&quot;
&apos;
```

enthalten, da diese nicht aufgelöst werden.

```
<!-- . . . -->
```

Zuletzt sei nochmals auf Kommentare hingewiesen. Diese dürfen keine doppelten Bindestriche (hyphens) im Inneren enthalten.

2.4 Vorschriften für Elemente und Attribute

Wie bereits angedeutet, müssen Elemente mit passendem Start- und Endetag eingeschlossen werden, oder leere Elemente sein, die mit /> enden. Ein Elementname (der *tag*) beginnt mit einem Buchstaben oder dem Unterstrich, danach kommen beliebige weitere Buchstaben, Ziffern, Unterstriche, Bindestriche oder Punkte. Der Name darf nicht mit xml (in irgendeiner Groß-/Kleinschreibung) beginnen und endet automatisch am ersten Leerzeichen oder dem >-Zeichen. XML ist *case sensitive*, wie bereits früher angemerkt. Ein Doppelpunkt darf nur zur Trennung von Namensraumangabe zu lokalem Namen verwendet werden. Laut [9] ist

<Italic>	legal
<_Budget>	legal
<Punch line>	illegal: Leerzeichen
<205Para>	illegal: fängt mit Zahl an
<repair@log>	illegal: @ nicht zugelassenes Zeichen
<xml>	illegal: fängt mit xml an.

Die Regeln für Elementnamen gelten auch für Attributbezeichner. Attribute bestehen aus Namen-Wert-Paaren, getrennt mit

dem Gleichheitszeichen. Der Attributwert wird immer in Anführungszeichen eingeschlossen.

2.5 Reservierte Attribute in XML

In [9] werden `xml:lang`, `xml:space`, `xml:link` und `xml:attribute` als sog. *reservierte Attribute* eingeführt, [7] bespricht nur `xml:link`, [8] kennt `xml:space` und `xml:lang` als sog. *vordefinierte Attribute*.

xml:lang

```
xml:lang="iso-639-Identifer"
```

Mit diesem Attribut wird die Sprache des Elements angegeben. Die Angabe verwendet einen zweibuchstabigen Sprachcode (Kleinschreibung ist üblich) gefolgt von einem Bindestrich, gefolgt von einem zweibuchstabigen Landescode (Großschreibung ist üblich).

```
<greetings>
  <greeting xml:lang="en-US">Hello</greeting>
  <greeting xml:lang="en-AU">G'day</greeting>
  <greeting xml:lang="de-DE">Hallo</greeting>
  <greeting xml:lang="ko-KR">Anyunghaseyo</greeting>
  <greeting xml:lang="no-nyn">
    new Norwegian 3-Letter IANA Code, not XML-Standard
  </greeting>
  <greeting xml:lang="x-prol">Eh bo</greeting>
```

Codes findet man z.B. auf <http://www.ics.uci.edu/pub/ietf/http/related/iso3166.txt>
bzw. .../iso639.txt.

xml:space

```
xml:space="default|preserve"
```

Bestimmt, ob die Whitespaces (Leerzeichen, Neue-Zeile, Tabulator) wie angegeben erhalten bleiben sollen (`preserve`) oder ob beliebige Ersetzungen erlaubt sind. Außer für Programmcode sollte man `xml:space="preserve"` nicht setzen.

☞ Scheint für IE5 nicht zu funktionieren.

xml:link

```
xml:link="link_type"
```

Hinweis: nach Durchsicht des W3C Working Draft vom 21. Feb. 2000 zur XML Linking Language (XLink) scheint sich die Syntax geändert zu haben. Was unten z.B. als `xml:link="..." href="..."` geziegt wird, zieht der Vorschlag zu `xlink:href="..."` zusammen. Die Darstellung hier folgt noch der alten Syntax.

Dieses Attribut signalisiert, daß das Element als Verweis (Link) zu betrachten ist. Als mögliche Werte gibt [9] an:

- `simple`
einfacher uni-direktionaler Verweis (entspricht `` in HTML).
- `extended`
ein Verweis auf ein Mitglied einer erweiterten Verweis-

gruppe, ein bi-direktionaler Verweis, usw. Hat keine direkte Entsprechung in HTML, ähnlich zu `<LINK>` in HTML.

Das `xml:link`-Attribut tritt immer in Verbindung mit weiteren Attributen auf, im Falle der *einfachen* Verweise z.B. mit

`href`, `inline`, `show`, `actuate`, `behavior`, `role`, `title`, `content-role`, `content-title`, `steps`.

Zunächst ein Beispiel für einen einfachen Verweis.

```
<?xml version="1.0" standalone="yes" ?>
<!-- xml:stylesheet type="text/xsl" href="html.xsl" -->
<beispiel>
  <zeilen>
    Hier folgt ein Link auf die Homepage der Vorlesung.
  </zeilen>
  <mylink xml:link="simple" inline="true"
  href="http://www.db.informatik.uni-kassel.de/Lehre/XML/">
    XML-Vorlesung
  </mylink>
</beispiel>
```

Das Attribut `inline="true"` gibt an, daß die Verweisinformation im eigenen Dokument ist. Alternativ kann sie extern sein (XLink: *out-of-line*). Letztere Idee geht auf Argumente wie im Design von Hyper-G (H.Maurer, Graz) ein, das Dokumente und Link-Spezifikation trennt. Im Dokument gibt es nur symbolische Namen für Verweise, die in einem getrennten (ggf. schreibgeschützten) Dokument sind.

Das Attribut `href` gibt den URI, URL oder Ort im Zieldokument an.

Tritt das Element `mylink` öfters auf, kann man über eine DTD das Standardverhalten einstellen.

```
<!ELEMENT mylink ANY>
<!ATTLIST mylink xml:link CDATA #FIXED "simple">
<!ATTLIST mylink inline (true | false) "false">
```

Jetzt kann man den Verweis sehr HTML-ähnlich schreiben.

```
<mylink href="http://www.db. ..." >XML-Vorlesung</mylink>
```

Die anderen Angaben sollten wir in einem eigenen Kapitel zu XPointer und XLink besprechen. Die Angabe `role` dient z.B. dazu, Beziehungen zwischen Dokumenten zu definieren, die intelligente Tools entsprechend nutzen können, etwa für Attributwerte `Autor`, `Weiter`, `Literatur`, ... Zukünftige Standards könnten hier eine Systematik einführen. Die Entsprechung in HTML 4.0 ist das `REV` und `REL` Attribute in `<LINK>`.

Die erweiterten Verweise (`xml:link="extended"`) bieten Gruppen von Verweiszielen an, ähnlich einem Drop-down Menü. Man nennt dies im Englischen „multi-ended links“.

Homer [7] gibt das folgende Beispiel.

```
<mylink xml:link="extended" inline="false">
  <mytarget xml:link="locator" inline="false"
    role="Font" title="How to specify different fonts"
    href="http://mysite.com/help/fonts.xml" />
  <mytarget xml:link="locator" inline="false"
    role="Color" title="How to specify different colors"
    href="http://mysite.com/help/colors.xml" />
  <mytarget xml:link="locator" inline="false"
    role="Text Size"
    title="How to specify different text sizes"
    href="http://mysite.com/help/textsize.xml" />
</mylink>
```

In einem Element mit Attribut `xml:link="group"` (das damit zu den erweiterten Verweisen gehört) können Elemente

mit `xml:link="document"` auftreten. Damit wird eine Gruppe von Dokumenten angegeben, die u.U. zusammen zu laden sind, aber einzeln referenziert werden können. Homer [7] gibt auch hierzu ein Beispiel.

```
<mylink xml:link="group" inline="false">
  <mytarget xml:link="document"
    href="http://mysite.com/docs/doc1.xml" />
  <mytarget xml:link="document"
    href="http://mysite.com/docs/doc2.xml" />
  <mytarget xml:link="document"
    href="http://mysite.com/docs/doc3.xml" />
</mylink>
```

Wir beenden hier die Besprechung von `xml:link`.

xml:attribute

`xml:attribute="existing-attribute replacement-attribute"`

Dieses Attribut erlaubt die Umbenennung eines Link-Attributes. Speziell für den häufig gebrauchten Attributbezeichner `title` kann man z.B in einem Element `person` angeben:

```
<person lastname="Schmidt" title="Dr."
  href="http://hiscompany.com/~schmidt.html"
  link-title="Homepage Dr. Schmidt"
  xml:attribute="title link-title" />
```

Aufgrund der Einführung des `xlink:-`Namensraums entfällt dieses Attribut.

2.6 Entity-Referenzen

Dies sind Substitutionen für Zeichenfolgen, die sonst als Markierungen interpretiert würden. Wir haben einige bereits in

Zusammenhang mit CDATA angegeben. Neben den vordefinierten können beliebige weitere in einer DTD angegeben werden (siehe unten).

Allgemein kann man Zeichenreferenzen auch direkt hexadezimal eingeben, und zwar beginnend mit `&#x`, dann den zwei Hexadezimalziffern für den Zeichencode, gefolgt von einem Semikolon „;“.

```
<?xml version="1.0" standalone="yes" ?>
<?xml:stylesheet type="text/xsl" href="html.xsl" ?>
<beispiel xmlns:xlink="http://www.w3.org/TR/2000/WD-xlink-
20000221">
  <zeilen>
    Hier folgt ein Link auf die Homepage der Vorlesung.
  </zeilen>
  <mylink xlink:href="http://www.db.informatik.uni-kas-
sel.de/Lehre/XML/">
    XML-Vorlesung &#xA9; L.Wegner 2000
  </mylink>
</beispiel>
```

Das Beispiel erzeugt ein Copyright-Zeichen ©, aber leider keine Verweisdarstellung :-)

Im nächsten Kapitel wollen wir Document Type Definitions wieder aufgreifen.

3 Document Type Definitions

3.1 Ein Datenbankbeispiel

Wie oben besprochen, werden XML-Dokumente, die einer DTD entsprechen, als gültig (valid) bezeichnet. Wir wollen hier die wesentlichen Elemente besprechen und wählen als ein Beispiel eine geschachtelte Tabelle, die wir aus zwei flachen Tabellen Abteilung und Mitarbeiter mit einer 1:n-Beziehung (jeder Mitarbeiter ist in genau einer Abteilung) herstellen.

ABTEILUNG

AID	ABTBEZ	ORT
FE	Forschung&Entwicklung	Dresden
HR	Personalabteilung	Kassel
EC	e-Commerce	?
VB	Vertrieb&Beratung	Dresden

MITARBEITER

NAME	MID	GJAHR	AID
Peter	1022	1959	FE
Gabi	1017	1963	HR
Beate	1305	1978	VB
Rolf	1298	1983	HR
Uwe	1172	1978	FE

Eine DTD für die geschachtelte Form wäre offensichtlich:

```
<!-- DTD fuer Abteilung und Mitarbeiter -->
<!ELEMENT ABTEILUNGEN (ABTEILUNG*)>
<!ATTLIST ABTEILUNGEN nf2type CDATA #FIXED "set">
```

```
<!ATTLIST ABTEILUNGEN id CDATA #REQUIRED>

<!ELEMENT ABTEILUNG (AID, ABTBEZ, ORT, MITARBEITER)>
<!ATTLIST ABTEILUNG nf2type CDATA #FIXED "ptuple">
<!ATTLIST ABTEILUNG id CDATA #REQUIRED>

  <!ELEMENT AID (#PCDATA)>
  <!ATTLIST AID id CDATA #REQUIRED>
  <!ELEMENT ABTBEZ (#PCDATA)>
  <!ATTLIST ABTBEZ id CDATA #REQUIRED>
  <!ELEMENT ORT (#PCDATA)>
  <!ATTLIST ORT id CDATA #REQUIRED>
  <!ELEMENT MITARBEITER (MITARB*)>
  <!ATTLIST MITARBEITER nf2type CDATA #FIXED "set">
  <!ATTLIST MITARBEITER id CDATA #REQUIRED>

    <!ELEMENT MITARB (NAME, MID, GJAHR)>
    <!ATTLIST MITARB nf2type CDATA #FIXED "ptuple">
    <!ATTLIST MITARB id CDATA #REQUIRED>
      <!ELEMENT NAME (#PCDATA)>
      <!ATTLIST NAME id CDATA #REQUIRED>
      <!ELEMENT MID (#PCDATA)>
      <!ATTLIST MID id CDATA #REQUIRED>
      <!ELEMENT GJAHR (#PCDATA)>
      <!ATTLIST GJAHR id CDATA #REQUIRED>
```

Ein XML-Dokument mit den entsprechenden Daten wäre

```
<?xml version='1.0' standalone="no" ?>
<?xml:stylesheet type="text/xsl" href="client8.xsl" ?>
<!DOCTYPE ABTEILUNGEN SYSTEM "AM.dtd">

<ABTEILUNGEN id="4000" >
  <ABTEILUNG id="4001">
    <AID id="4003">FE</AID>
    <ABTBEZ id="4004">Forschung&Entwicklung</ABTBEZ>
    <ORT id="4005">Dresden</ORT>
    <MITARBEITER id="4006">
      <MITARB id="4007">
        <NAME id="4008">Peter</NAME>
        <MID id="4009">1022</MID>
        <GJAHR id="4010">1959</GJAHR>
```

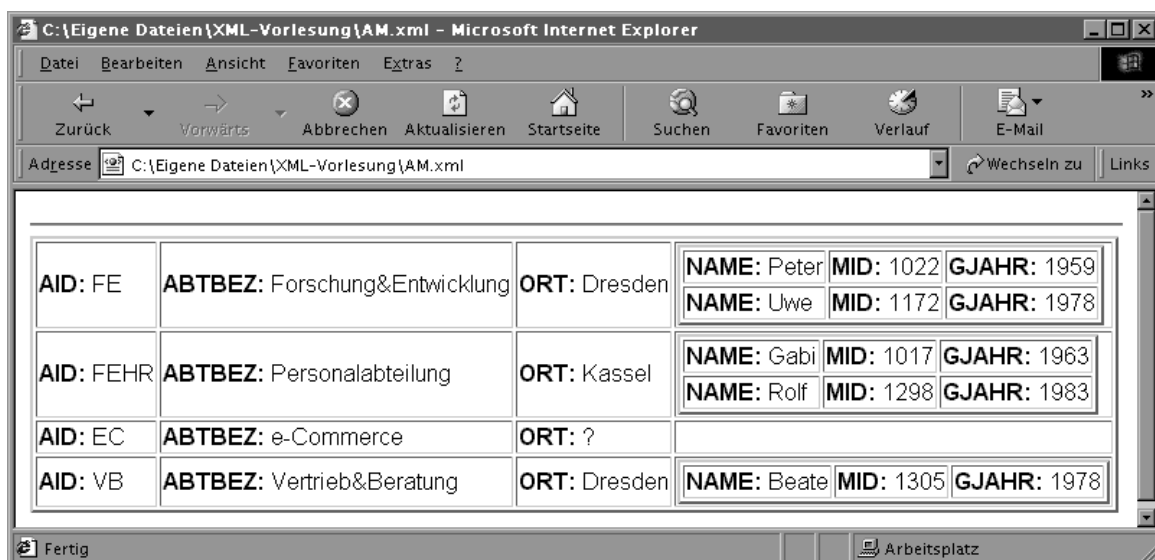
```
</MITARB>
  <MITARB id="4011">
    <NAME id="4012">Uwe</NAME>
    <MID id="4013">1172</MID>
    <GJAHR id="4014">1978</GJAHR>
  </MITARB>
</MITARBEITER>
</ABTEILUNG>
<ABTEILUNG id="4020">
  <AID id="4021">FEHR</AID>
  <ABTBEZ id="4022">Personalabteilung</ABTBEZ>
  <ORT id="4023">Kassel</ORT>
  <MITARBEITER id="4024">
    <MITARB id="4025">
      <NAME id="4026">Gabi</NAME>
      <MID id="4027">1017</MID>
      <GJAHR id="4028">1963</GJAHR>
    </MITARB>
    <MITARB id="4029">
      <NAME id="4030">Rolf</NAME>
      <MID id="4031">1298</MID>
      <GJAHR id="4032">1983</GJAHR>
    </MITARB>
  </MITARBEITER>
</ABTEILUNG>
<ABTEILUNG id="4040">
  <AID id="4041">EC</AID>
  <ABTBEZ id="4042">e-Commerce</ABTBEZ>
  <ORT id="4043">?</ORT>
  <MITARBEITER id="4044">

  </MITARBEITER>
</ABTEILUNG>
<ABTEILUNG id="4060">
  <AID id="4061">VB</AID>
  <ABTBEZ id="4062">Vertrieb&Beratung</ABTBEZ>
  <ORT id="4063">Dresden</ORT>
  <MITARBEITER id="4064">
    <MITARB id="4065">
      <NAME id="4066">Beate</NAME>
      <MID id="4067">1305</MID>
      <GJAHR id="4068">1978</GJAHR>
```

```
</MITARB>
</MITARBEITER>
</ABTEILUNG>
</ABTEILUNGEN>
```

Hinweis: die entsprechenden Dateien sind `AM.xml` und `AM.dtd` und werden in der Web-Seite der Vorlesung zur Verfügung gestellt.

Mit einem passenden Stylesheet (hier `client8.xsl`) ergibt sich die folgende Anzeige (Applaus, Applaus!).



AID: FE	ABTBEZ: Forschung&Entwicklung	ORT: Dresden	NAME: Peter MID: 1022 GJAHR: 1959
			NAME: Uwe MID: 1172 GJAHR: 1978
AID: FEHR	ABTBEZ: Personalabteilung	ORT: Kassel	NAME: Gabi MID: 1017 GJAHR: 1963
			NAME: Rolf MID: 1298 GJAHR: 1983
AID: EC	ABTBEZ: e-Commerce	ORT: ?	
AID: VB	ABTBEZ: Vertrieb&Beratung	ORT: Dresden	NAME: Beate MID: 1305 GJAHR: 1978

Das Stylesheet hierzu werden wir später besprechen. Zunächst sind wir an den DTD-Elementen interessiert.

3.2 Die DTD Elementdeklarationen

Jedes in einem XML-Dokument auftretende Element muß in der DTD deklariert sein. Dies geschieht mit der Elementdeklaration

```
<!ELEMENT elementname rule>
```

wobei Elementname ohne spitze Klammern angegeben wird. Was hier als *rule* bezeichnet wird, legt fest, was zwischen der Start- und Endemarkierung des Elements stehen darf.

ANY und PCDATA

Im einfachsten Fall erlaubt man mit einer Angabe

```
<!ELEMENT book ANY>
```

beliebige Daten, auch andere Tags.

Will man andererseits nur „Nutzdaten“ ohne Metazeichen (sog. *parsed character data*), lautet die Angabe PCDATA.

```
<!ELEMENT title (#PCDATA)>
```

Damit wäre in einem XML-Dokument

```
<title></title>  
<title>Skriptum XML-Vorlesung &#xA9; L. Wegner</title>
```

legal, nicht jedoch

```
<title><emphasis>Skriptum XML-Vorlesung</emphasis></title>
```

Will man angeben, daß ein Element in einem anderen Element auftreten muß, dann gibt man das mit runden Klammern an.

Muß in unserem Beispiel ein Buchtitel in einem Buch erscheinen, lauten die Deklarationen:

```
<!ELEMENT book (title)>  
<!ELEMENT title (#PCDATA)>
```

Mehrfache Elemente

Sollen mehrere Elemente in einer vorgegebenen Reihenfolge auftreten, gibt man diese durch Komma getrennt an.

```
<!ELEMENT book (title, authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
```

Damit kann man in einem XML-Dokument schreiben

```
<book>
  <title>Skriptum XML-Vorlesung</title>
  <authors>Wegner</authors>
</book>
```

Ein Vertauschen oder Weglassen eines Elements wäre aber nicht möglich. Dies erreicht man mit der Alternative, allerdings muß genau *eine* der Alternativen erscheinen!

```
<!ELEMENT book (title | authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
```

Gruppierung und Wiederholung

Wie man dies aus den kontextfreien Grammatiken in EBNF-Notation und von der Shell-Programmierung kennt, kann man Gruppierungen und (optionale) Wiederholungen angeben. Soll ein Buch entweder eine Beschreibung oder einen Titel gefolgt von der Autorenangabe enthalten, schreibt man

```
<!ELEMENT book ((title, authors) | description)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Das optionale Auftreten gibt man mit `?`, das mindestens einmalige mit `+`, und das null- oder mehrmalige Auftreten mit `*` an. Die Metazeichen stehen unmittelbar nach den Elementnamen oder den Gruppen, auf die sie sich beziehen sollen.

Das Beispiel in [9] lautet:

```
<!ELEMENT reviews (rating, synopsis?, comments+)*>
<!ELEMENT rating (tutorial | reference)*, overall)>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT tutorial (#PCDATA)>
<!ELEMENT reference (#PCDATA)>
<!ELEMENT overall (#PCDATA)>
```

Leere Elemente

Auch das Auftreten leerer Elemente in einem XML-Dokument muß angegeben werden.

```
<!ELEMENT elementname EMPTY>
```

Damit kann durch

```
<!ELEMENT statuscode EMPTY>
```

im XML-Dokument entweder

```
<statuscode></statuscode>
```

oder

```
<statuscode/>
```

stehen.

3.3 Entities

Ähnlich wie bei Makros lassen sich Zeichenfolgen mit einem Bezeichner versehen. Einige Standardvorgaben (z.B. &#x26;) haben wir schon gesehen, diese müssen nicht gesondert angegeben werden. Andere gibt man wie folgt an:

```
<!ENTITY copyright "&#xA9;">
```

Im Dokument schreibt man dann

```
<title>XML-Skript &copyright; L.Wegner 2000</title>
```

und erzeugt damit die Ausgabe

XML-Skript © L.Wegner 2000

Übung 3.1

Wie lautet die Angabe für das Copyleft-Zeichen „ © “?



Wie bei Makros üblich, dürfen die Angaben nicht zirkulär sein. Auch dürfen in den Substitutionszeichen keine weiteren Metazeichen auftauchen, da die Ersetzung erst im XML-Dokument erfolgt, nicht bereits in der DTD. Dies ist bei den folgenden parametrischen Entities anders.

Parameter-Entities

Diese werden nur in DTD's verwandt und können als deklarierte Elemente nicht in XML-Dokumenten auftreten. Sie stellen abkürzende Schreibweisen für DTD's dar. Vor dem Namen erscheint ein Prozentzeichen.

```
<!ENTITY % name "replacement_characters">
```

Das Beispiel in [9] lautet:

```
<!ENTITY % pcddata "(#PCDATA)">
<!ELEMENT authortitle %pcdata;>
```

Wie zuvor dürfen die Deklarationen nicht zirkulär sein und ein Parameter-Entity muß zuerst deklariert sein, bevor es verwendet werden kann.

Externe Entities

Hier sollen Daten, die von einer anzugebenden URI stammen, dynamisch in das XML-Dokument geladen werden.

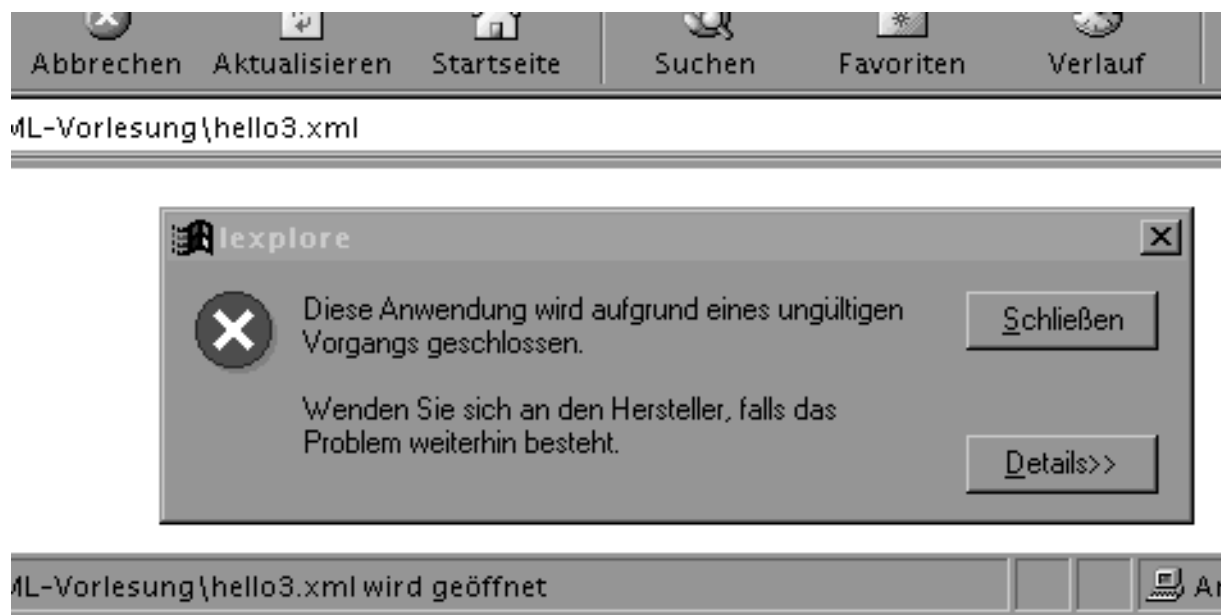
```
<!ENTITY fremd SYSTEM "fremd.xml">
```

Befinden sich in `fremd.xml` (ggf. an einem anderen Ort) XML-Daten, z.B.

```
<einbau>Wer&apos;s glaubt wird seelig</einbau>
```

wird dieser Text zur Laufzeit in das XML-Dokument aufgenommen, das die Referenz `&fremd;` enthält (funktioniert in IE5).

Verwendet man erneut in `fremd.xml` den rekursiven Aufruf `&fremd;`, erzeugt man die folgende Ausgabe (immerhin keinen Systemcrash).



☞ **Randbemerkung:** Man erzeugt damit effektiv ein unendliches Dokument. Frage: Kann dies zu einem Sicherheitsrisiko für den ladenden Client werden?

Unparsed Entities

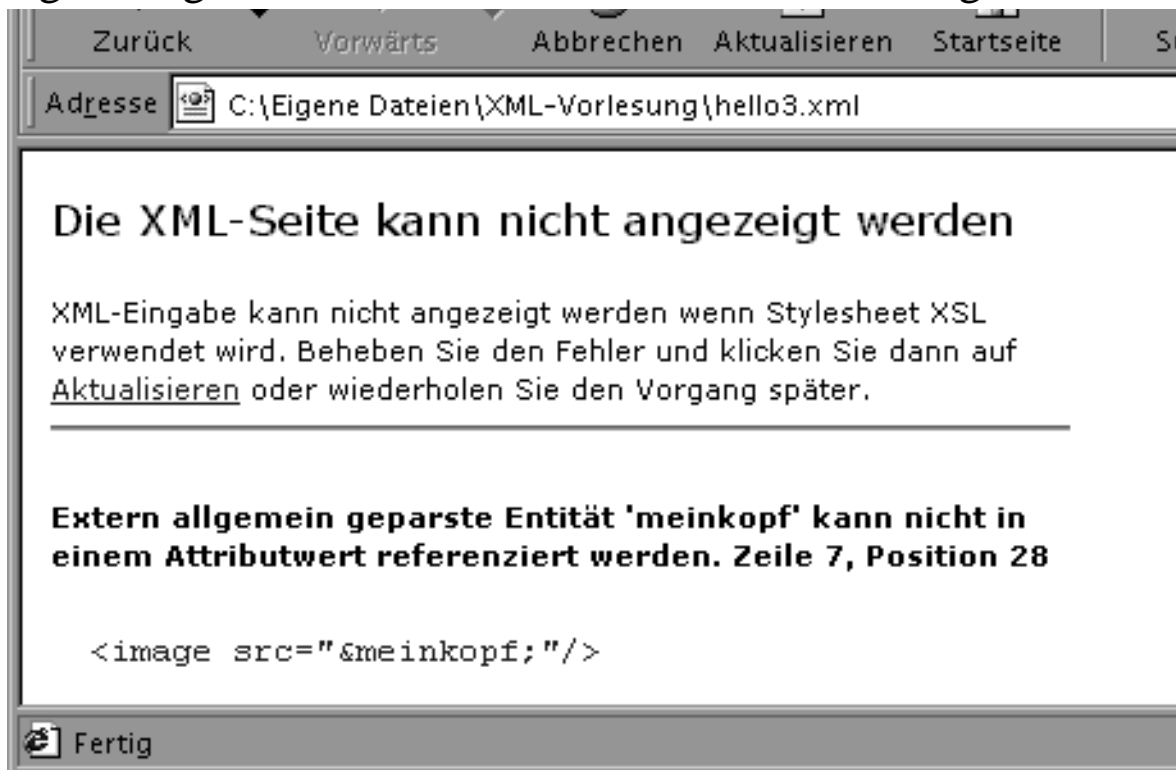
Hiermit lassen sich Daten, die kein XML darstellen, in XML-Dokumente einbauen. Üblich sind Bilder. Das in [9] gegebene Beispiel funktionierte so aber nicht im IE5. Ein ähnlich strukturiertes wäre

```
<!ENTITY meinkopf SYSTEM "lutz-thumb.jpg" NDATA jpg>
```

Speziell wird damit durch das Schlüsselwort NDATA (notation data) angegeben, um welche Art von Rohdaten es sich handelt, hier ein jpg-Pixelbild. In [9] wird die Verwendung in einem Attributwert mittels

```
<image src="&meinkopf;"/>
```

vorgeschlagen, IE5 scheint diese Form nicht zu mögen.



Schreibt man `src="meinkopf"` wird das XML-Dokument akzeptiert, das Bild aber weiterhin nicht angezeigt. Eigentlich

müßte in der DTD auch das Attribut `src` definiert werden:

```
<!ELEMENT image EMPTY>
<!ATTLIST image src ENTITY #REQUIRED>
```

Die Verwendung eines „unparsed entity“ direkt im XML-Text ist nicht gültig, man kann also nicht einfach schreiben

```
<greeting>&meinkopf;</greeting>
```

Auch mit Einbau der folgenden Notation-Angabe funktioniert eine jpg-Bildausgabe nicht (was nicht überrascht, weil die Spezifikation des Graphiktyps doch zu unklar ist).

Notation

Hier soll die Angabe nach dem Schlüsselwort `NDATA` in Verbindung gebracht werden mit „genauerer Angabe“ (Treiberinfo?). Das Beispiel aus Homer [7] lautet:

```
<!NOTATION jpg SYSTEM "file://Oursystem/JPEGs.cat">
```

Homer verwendet den Notation-Wert auch etwas anders, nämlich in

```
<ATTLIST graphic image_format NOTATION (bmp | jpg | gif )
```

für ein Element `<!ELEMENT graphic EMPTY>` und setzt es ein für

```
<graphic image_format="gif"/>
```

Eckstein [9] gibt eine URI für die Notation an

```
<!NOTATION GIF89a SYSTEM "-//CompuServe//NOTATION Graphics
Interchange Format 89a//EN">
```

und bemerkt richtigerweise, daß der XML Prozessor diese Information ignorieren kann.

Generell werden hier die Grenzen einer Metasprache klar, denn für nicht interpretierbare Daten müßten Metaprotokolle angegeben werden, die zur Laufzeit greifen und stabil (über Standardweiterentwicklungen hinaus) sind.

3.4 Attributdeklarationen

Wie schon mehrfach gezeigt, müssen Attribute der Elemente deklariert werden.

```
<!ATTLIST target_element attr_name attr_type default>
```

Beispiele aus [9] sind

```
<!ATTLIST box length CDATA "0">
<!ATTLIST box width CDATA "0">
<!ATTLIST frame visible (true|false) "true">
<!ATTLIST person marital
    (single | married | divorced | widowed) #IMPLIED>>
```

Als Standardwert (default) kann man einen passenden Wert hinschreiben oder eines von drei Schlüsselwörtern:

```
#REQUIRED
#IMPLIED
#FIXED
```

Required heißt, der Wert muß im Element angegeben werden. *Implied* heißt, ein Wert kann angegeben werden. Fehlt er, wird die Anwendung benachrichtigt und kann frei entscheiden, wie sie mit dem fehlenden Attributwert umgehen will. Diese Angabe entspricht in etwa einer Datenbank Null (SQL: `isnull`). *Fixed* heißt, daß der Wert dem Schlüsselwort sofort folgt.

Attributtypen

Als Datentypen für Attribute sind vorgesehen

Attribut	Beschreibung
CDATA	Zeichenfolge
Aufzählungstyp	Folge von Werten unter denen einer aufzuwählen ist
ENTITY	ein in der DTD deklariertes Entity
ENTITIES	mehrere durch Whitespaces getrennte und in der DTD deklarierte Entities
ID	ein eindeutiger Elementbezeichner (Schlüsseleigenschaft)
IDREF	Zeiger auf ID (Wert eines anderen IDs)
IDREFS	mehrere durch Whitespaces getrennte IDREFs
NMTOKEN	ein XML Namenstoken
NMTOKENS	mehrere durch ...
NOTATION	eine in der DTD deklarierte Notation

☞ Hinweis: Attribute haben nur Character Data (CDATA) als Typ, Elemente haben ANY, PCDATA, usw.

Als Beispiel für ID und IDREF könnte man sich die folgende Anwendung `graph.xml` denken.

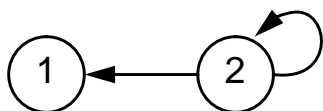
```
<?xml version='1.0' standalone="no" ?>
<!-- ?xml:stylesheet type="text/xsl" href="hello.xsl"? -->
<!DOCTYPE graph SYSTEM "graph.dtd">
<graph>
  <node label="node1"/>
  <node label="node2" arcs="node1 node2"/>
</graph>
```

mit der DTD `graph.dtd`

```
<!-- DTD for ID und IDREFS -->
<!ELEMENT graph (node)*>
<!ELEMENT node EMPTY>
```

```
<!ATTLIST node label ID #REQUIRED>  
<!ATTLIST node arcs IDREFS #IMPLIED>
```

Ein entsprechendes Stylesheet könnte dies als Graphen der Form



Include, Ignore und interne Deklarationen

Wie bei bedingten Includes in C können Teile der DTDs ignoriert oder bedingt zugeladen werden. Wir verzichten hier auf die Angabe.

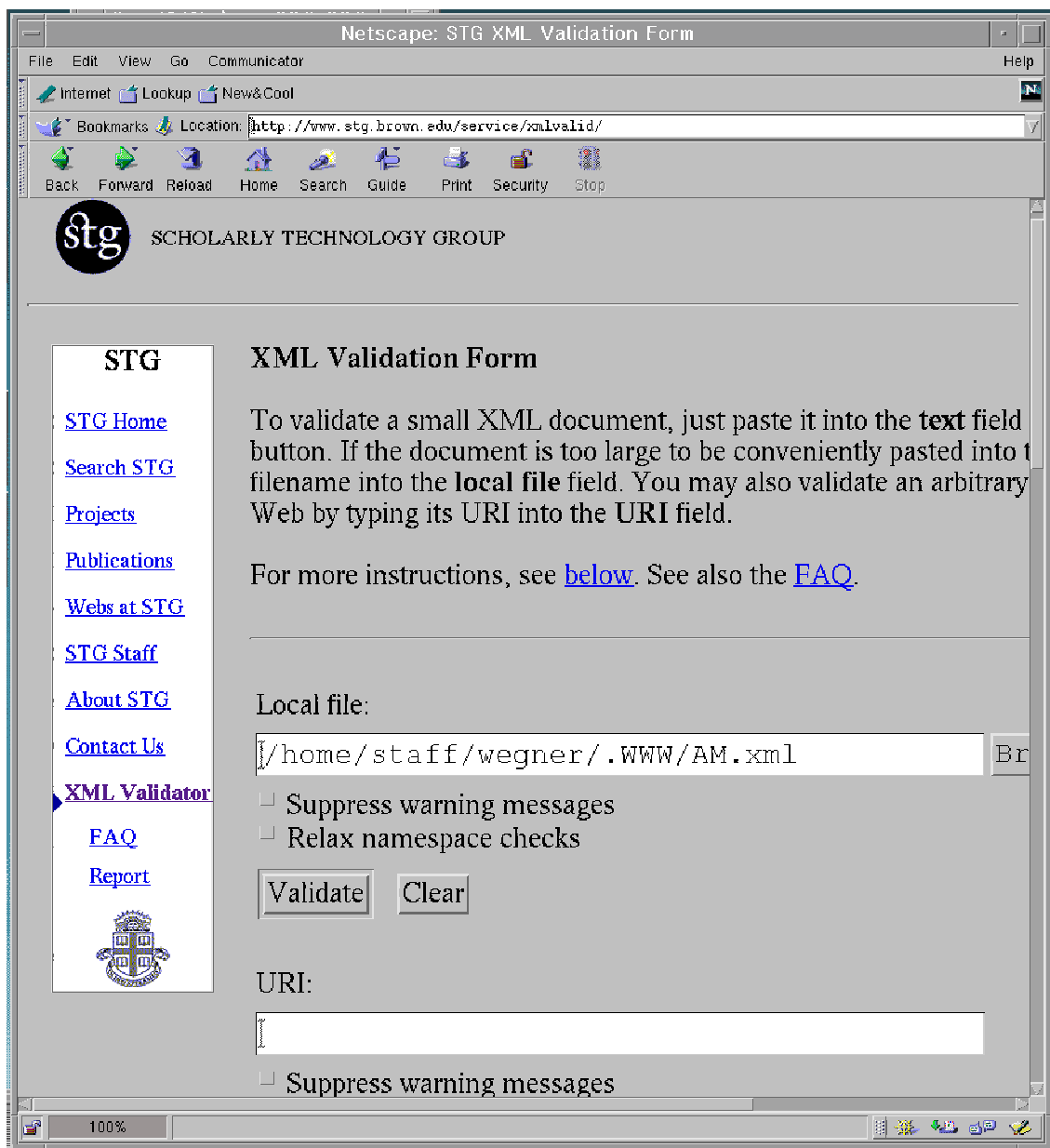
Ferner ist es möglich, in der `<!DOCTYPE . . . >` Instruktion im XML-Dokument Teile der DTD direkt anzugeben. Bei Gleichheit überdecken die lokalen Angaben die externen. Auch hier verzichten wir auf Details.

3.5 Validierende Parser

IE5 und Mozilla überprüfen zur Zeit XML-Dokumente nicht automatisch auf Gültigkeit relativ zu einer angegebenen DTD. In [8] werden Adressen von validierenden XML-Parsern angegeben, z.T. mit Web-Interface (nicht für Dauergebrauch!).

<http://www.stg.brown.edu/service/xmlvalid/>

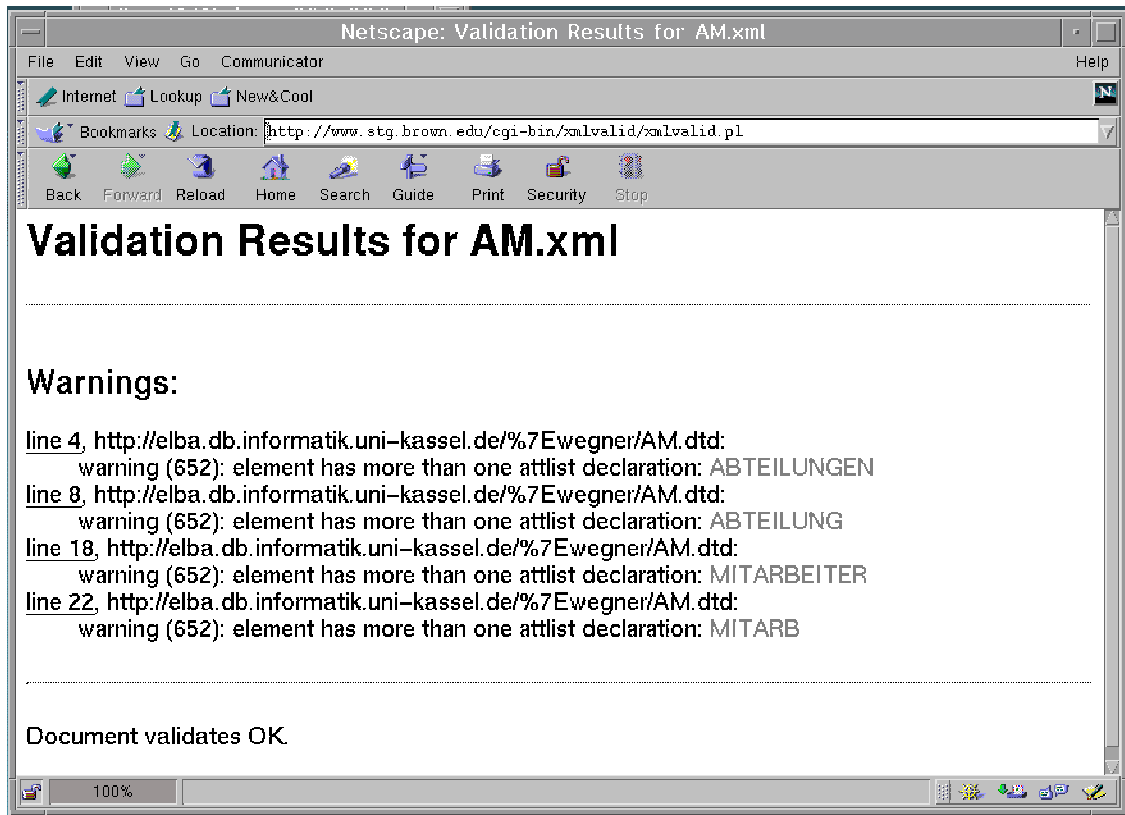
Die folgenden Seiten zeigen den Aufruf und das Ergebnis für `AM.xml` mit `AM.dtd`. Angepaßt werden mußte die DTD-Referenz (auf Internet-Referenz). Danach wurden auf Anhieb nur 4 Warnings geliefert.



Ein anderer Validator ist

<http://www.cogsci.ed.ac.uk/%7Erichard/xml-check.html>

Dieser meldete keinerlei Fehler bei der Validity und erklärte „well-formedness“. Die Web-Seite enthält weitere Hinweise auf Parser und auch Adressen zum Herunterladen von Parsern. Einige beruhen auf Java-Implementierungen und sind



umständlich zu installieren. Leicht zu installieren sind die Tools von Microsoft für IE5 (<http://www.microsoft.com/downloads>, Aufruf im IE5 mit rechter Maustaste). Auch der dort verfügbare Viewer für XSL-Output ist nützlich. In [8] werden als z.Zt verfügbare Tools genannt „XML for Java“: <http://www.alphaworks.ibm.com/tech/xml>, „XJParser“: http://www.datachannel.com/xml_resources/, „SXP“: <http://www.loria.f/projects/XSilfide/EN/sxp>.

Generelle Hinweise zu XML-Tools finden sich unter <http://www.xmlsoftware.com/>

4 XML Schema

Die Besprechung der XML Schema bezieht sich auf die Darstellung durch Homer [7] und die soeben erschienenen Standardisierungsvorschläge:

XML Schema Part 0: Primer, Working Draft 7. April 2000

<http://www.w3.org/TR/xmlschema-0/>

XML Schema Part 1: Structures, Working Draft 7. April 2000

<http://www.w3.org/TR/xmlschema-1/>

XML Schema Part 2: Datatypes, Working Draft 7. April 2000

<http://www.w3.org/TR/xmlschema-2/>

Davon ist *Part 0: Primer* ein zwar „längliches“, aber halbwegs lesbares Dokument (das nicht normativen Charakter hat). Part 1 und 2 sind dagegen die eigentlichen Normungsvorschläge und als Urlaubslektüre nicht geeignet.

4.1 Warum DTD und XML Schema?

Wir hatten bereits früher angedeutet, daß DTDs syntaktisch und bezüglich der Ausdrucksmöglichkeiten (Datentypen, Vererbungskonzepte) nicht optimal sind.

Homer erläutert das etwas und erklärt es mit der Herkunft der DTDs aus der SGML-Welt. Als Beispiele für didaktisch schlechte Konstruktionen erwähnt er den inkonsistenten Gebrauch der Anführungszeichen (aus [7], S. 67):

```
<!ATTLIST COVERTYPE GRAPHICALIGN (left | center | right)
    "center">
```

Auch erwähnt werden die Entities, z.B.

```
<!ENTITY % wrox "Wrox Press Limited">
```

mit Aufruf `%wrox`; in einer DTD, dagegen aber

```
<!ENTITY wrox "Wrox Press Limited">
```

mit Aufruf `&wrox`; in einem Dokument. Schließlich hält er

```
<![CDATA["This is the value of the entity"]]>
```

auch nicht gerade für die Krone der Schöpfung.

4.2 Ein XML Dokument mit einem XML Schema

IE5 unterstützt XML Schemata (XSDs) und validiert XML-Dokumente mit dem oben beschriebenen herunterladbaren Validierer. Einige der im neuen Normierungsvorschlag genannten Eigenschaften werden naturgemäß noch nicht akzeptiert, z.B. der Datentyp ID.

Zum einfacheren Verständnis wurde die oben genannte DTD `AM.dtd` aus dem Abteilungen-Mitarbeiterbeispiel durch ein XML Schema `AMschema.xml` ersetzt (man beachte, XML Schemata sind gültige XML-Dokumente, der neue Standard scheint darüberhinaus das Suffix `xsd` zu favorisieren).

Das XML-Dokument und das existierende Stylesheet (`Client8.xsl`) sollten möglichst unverändert bleiben! Tatsächlich ließ sich dies „auf Anhieb“ realisieren. Das neue XML-Dokument, in dem der Verweis auf das Schema einzutragen war, heißt jetzt `AMS.xml` und enthält statt

<!DOCTYPE . . . > jetzt die Zeile

```
<ABTEILUNGEN id="4000" xmlns="x-schema:AMSchema.xml">
```

Das angegebene Schema `AMSchema.xml` wurde aus dem Beispiel von Homer in dessen Kapitel 3 heraus angepaßt (die Quelltexte stellt Homer unter <http://www.wrox.com/> zur Verfügung). Warnend sei darauf hingewiesen, daß diese sich an den IE5 „Vorabimplementierungen“ orientieren und der Normierungsvorschlag bei erster Durchsicht durchaus davon abweicht.

```
<?xml version="1.0"?>
<Schema name="listSchema"
  xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <AttributeType name="id" dt:type="string"/>

  <ElementType name="NAME" content="textOnly"
    model="closed" dt:type="string">
    <attribute type="id" />
    <description>der Mitarbeitername</description>
  </ElementType>

  <ElementType name="MID" content="textOnly"
    model="closed" dt:type="string">
    <attribute type="id" />
    <description>der Mitarbeiter-Identifier - Schluessel
    </description>
  </ElementType>

  <ElementType name="GJAHR" content="textOnly" model="closed">
    <attribute type="id" />
    <description>das Geburtsjahr des Mitarbeiters
    </description>
  </ElementType>

  <AttributeType name="nf2type" dt:type="enumeration"
```

```
        dt:values="set list gtuple ptuple atomic" />

<ElementType name="MITARB" content="eltOnly" model="closed">
  <description>ein Mitarbeiter-Tuple</description>
  <attribute type="id" />
  <attribute type="nf2type" default="ptuple"/>
  <element type="NAME" />
  <element type="MID" />
  <element type="GJAHR" />
</ElementType>

<ElementType name="MITARBEITER"
  content="mixed" model="open" order="many">
  <description>Menge der Mitarbeiter der Abteilung
</description>
  <attribute type="id" />
  <attribute type="nf2type" default="set"/>
  <element type="MITARB" />
</ElementType>

<ElementType name="AID" content="textOnly"
  model="closed" dt:type="string">
  <attribute type="id" />
  <description>Abteilungs-Identifizier - Schluessel
</description>
</ElementType>

<ElementType name="ABTBEZ" content="textOnly"
  model="closed" dt:type="string">
  <attribute type="id" />
  <description>Abteilungsbezeichnung</description>
</ElementType>

<ElementType name="ORT" content="textOnly"
  model="closed" dt:type="string">
  <attribute type="id" />
  <description>Ort an dem die Abteilung untergebracht ist
</description>
</ElementType>

<ElementType name="ABTEILUNG" content="eltOnly"
  model="closed">
```

```
<description>Abteilungstupel mit Mitarbeitern
</description>
<attribute type="id" />
<attribute type="nf2type" default="ptuple"/>
<group order="seq">
  <element type="AID" />
  <element type="ABTBEZ" />
  <element type="ORT" />
  <element type="MITARBEITER" minOccurs="0" />
</group>
</ElementType>

<ElementType name="ABTEILUNGEN" content="eltOnly"
  model="closed">
  <description>Das Wurzelement der Abteilungstabelle
</description>
  <attribute type="id" />
  <attribute type="nf2type" default="set"/>
  <element type="ABTEILUNG" minOccurs="1"
    maxOccurs="*" />
</ElementType>

<!-- now we must not forget the closing schema tag -->
<!-- and yes, you can use XML comments in a schema -->
</Schema>
```

Zunächst fällt auf, daß das Schema den Schemabaum „von unten nach oben“ auflistet. Laut Homer verlangt IE5 die Definition von Elementen und Attributen, bevor diese in anderen Definitionen erscheinen. Das führt dazu, daß Attribute definiert werden, ohne daß man weiß, in welchen Elementen sie zur Anwendung kommen sollen.

Wie in der DTD zu `AM.xml` definieren wir Attribute `id` für alle (atomaren und komplexen) Elemente. Das Attribut `nf2type` verwenden wir dagegen nur für komplexe Objekte (indem wir es dort durch einen Defaultwert belegen), definie-

ren es aber generell für alle Elemente. Sein Typ ist ein Aufzähltyp

```
dt:type="enumeration"  
dt:values="set list gtuple ptuple atomic"
```

Wir brauchen die Typangabe für unser generisches Stylesheet und vermeiden mit dieser Defaultangabe die explizite Angabe für jedes relevante Element im XML-Dokument.

Genauer muß man festhalten, daß Attribute global für alle Elemente deklariert werden können, indem man sie vor dem ersten Element definiert. Alternativ kann man ein Attribut innerhalb eines Elements definieren mit Wirkung nur auf dieses Element. Homer gibt das folgende generelle Schema, daß allerdings nicht kompatibel zu obigem Beispiel ist (Attributdefinitionen haben Namens-Attribute, nicht Attributs-Attribute). `AttributeType`-Elemente sind in der Regel leere Elemente, usw. Generell ist die Unterscheidung „global/lokal“ aber richtig und nützlich.

```
<Schema ...>  
  
<AttributeType                <!-- globales Attribut -->  
  <description />  
  <attribute .../> <!-- name= ??? -->  
  <datatype ... /> <!-- dt:type= ??? -->  
</AttributeType>  
  
<ElementType ...>  
  
  <AttributeType ...>      <!-- lokales Attribut -->  
    <description />  
    <attribute ... />  
    <datatype ... />  
  </AttributeType>
```

```
<description />
<datatype .../>
<element ... />
<attribute ... />

<group ...>
  <element ... />
  <attribute ... />
</group>

</ElementType>

</Schema>
```

Bei den `ElementTypes` fallen die folgenden Attribute auf:

- `content`: definiert den erlaubten Inhalt eines Elements, z.B. "empty", "textOnly" (sofern nicht `model="open"`), "eltOnly" (nur andere Elemente, kein freier Text), "mixed".
- `dt:type`: erlaubter Text, Werte z.B. string, number, int, float, date., time., ui4, uri, stark an Datentypen in C, C++, VB, SQL, Java angelehnt. Eine Übersicht findet sich in Part 0: Primer, Table 2 des Normvorschlags.
- `model`: "open", "closed"
- `name`: der eigentliche Elementname
- `order`: "one", "seq", "many" (genau ein Element, alle in der gegebenen Reihenfolge, beliebige Auswahl- ziemliche kranke Notation!)

Innerhalb der komplexen Elemente werden mit `<element ... />` die Unterelemente deklariert, die oben (IE5 zur Zeit!) definiert sein müssen.

Das Attribut `type="..."` in diesen `<element .../>`-Deklarationen ist ziemlich mißverständlich, denn es ist der *Name* des Elements, auf den sich diese Deklaration bezieht.

So bedeutet

```
<ElementType name="X" ...>
  <element type="Y" minOccurs="0" />
  ...
</ElementType>
```

daß ein Element Y Teil eines komplexen Elements X ist und ggf. „nullmal“ (garnicht) in XML-Dokumenten innerhalb von X erscheint.

Wie oben angesprochen, gibt es `AttributeType`-Elemente genauso wie es `ElementType`-Elemente gibt. Erstere haben ähnliche Attribute (die Attribute der Attributdeklaration):

- `dt:type`
- `dt:value`
- `default`
- `name`
- `model`
- `required`

Wir verzichten auf die genaue Angabe der Verwendung.

Das `attribute`-Element wiederum ist ähnlich zum `element`-Element und gibt das Auftreten eines Attributs in einem Element an. Seine Attribute sind

- `default`
- `required`

- type

wobei letzteres wiederum der *Name des Attributs* ist, das für ein Element deklariert wird. So galt im Beispiel oben:

```
<ElementType name="ABTEILUNGEN" ...>
  <attribute type="nf2type" default="set"/>
  ...
</ElementType>
```

Homer berichtet auch von einem `extends`-Attribut, mittels dem man Vererbung und Unterklassenbildung betreiben kann. Das Attribut findet sich aber als Schlüsselwort nicht mehr im Normungsvorschlag, wird als Begriff allerdings im Rahmen von Erweiterung und Restriktion bei der *Typhierarchie* in Part 1: Structures erwähnt.

4.3 Ein Beispiel aus dem Normvorschlag

Zuletzt sei ein Beispiel aus dem neuen Normvorschlag, entnommen dem *Part 0: Primer*, vorgestellt. Man erkennt daran die gravierenden Unterschiede.

Das XML-Dokument ist eine Bestellung und in der Datei `po.xml` abgelegt.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
```

```
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <price>148.95</price>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <price>39.98</price>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

Die Bestellung besteht aus dem Hauptelement (Wurzel) `purchaseOrder` und den Unterelementen `shipTo`, `billTo`, und `items`. Diese enthalten wieder Unterelemente, bis man zu atomaren (W3C: *simple*) Elementen wie dem Preis (`price`) kommt, der eine Zahl enthält. Einige der Elemente haben Attribute; Attribute haben immer einen einfachen (*simple*) Typ.

Das zugehörige Schema ist in `po.xsd` abgelegt. Das XML-Dokument `po.xml` enthält allerdings keinen Verweis auf dieses spezielle Schema (W3C: um anzudeuten, daß das Dokument auch ohne Schema verwendbar wäre).

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder"
    type="PurchaseOrderType"/>

  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:element name="shipTo" type="Address"/>
    <xsd:element name="billTo" type="Address"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="Address">
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
    <xsd:attribute name="country" type="xsd:NMTOKEN"
      use="fixed" value="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:element name="item" minOccurs="0"
      maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:element name="productName" type="xsd:string"/>
        <xsd:element name="quantity">
          <xsd:simpleType base="xsd:positiveInteger">
            <xsd:maxExclusive value="100"/>
          </xsd:simpleType>
        </xsd:element>
      </xsd:complexType>
    </xsd:element>
  </xsd:complexType>
</xsd:schema>
```

```
<xsd:element name="price" type="xsd:decimal"/>
<xsd:element ref="comment" minOccurs="0"/>
<xsd:element name="shipDate" type="xsd:date"
  minOccurs="0"/>
<xsd:attribute name="partNum" type="Sku"/>
</xsd:complexType>
</xsd:element>
</xsd:complexType>

<xsd:simpleType name="Sku" base="xsd:string">
  <xsd:pattern value="\d{3}-[A-Z]{2}"/>
</xsd:simpleType>

</xsd:schema>
```

Auch in diesem Schema gibt es die Unterscheidung lokal/global, diesmal für Elemente, hier: `comment`. Man beachte die Verwendung in der Deklaration mittels `ref="comment"`.

Bei Attributdefinitionen taucht eine `attribute use` auf, das im Beispiel den Wert `fixed` annimmt. Daneben sind `required` und `optional` möglich, mit `value` läßt sich ein Wert angeben (muß angegeben werden bei `fixed`).

Auf weitere Details müssen wir hier verzichten.

Diese Seite
wurde absichtlich
freigehalten

5 XSL Transformationen

Damit man an Beispielen mit XML praktisch üben kann, ist es jetzt wichtig, daß wir Methoden zur Ausgabeformatierung kennenlernen. Hierzu gibt es die *Extensible Stylesheet Language* (XSL).

Der Begriff XSL ist aber mehr als ein generischer Begriff zu verstehen. Dahinter stehen

- Transformationen, d.h. Umwandlungen von XML Dokumenten in z.B. HTML Dokumente
- direkte Formatierungsangaben, die sog. *formatting objects*.

XSL Transformations (XSLT) werden als *Version 1.0* in der W3C Empfehlung vom 16. November 1999 beschrieben [3]. Sie verwenden den Namensraum „`xs1:`“. XSLT bildet die Grundlage für die folgende Besprechung.

Die sog. *formatting objects* verwenden den Namensraum „`fo:`“. Sie werden in der W3C Working Draft *Extensible Stylesheet Language (XSL) Version 1.0* vom 27. März 2000 zur Normung vorgeschlagen (<http://www.w3.org/TR/xs1/>) und bilden von der Mächtigkeit her ein Desktop Publishing System. IE5 unterstützt sie gegenwärtig nicht.

5.1 XML Dokumente als Bäume

XML Dokumente sind hierarchische Strukturen. Isomorphe Darstellungsformen sind Bäume oder geschachtelte Tabellen.

Besonders die Sicht als Baum - also mit Wurzel, (inneren) Knoten und Blättern - ist nützlich, denn XSLT transformiert gegebene XML-Bäume in neue XML-Bäume, wobei letztere dann z.B. als HTML-Dokumente interpretierbar sind.

In [8] wird der Dokumentbaum für ein Beispiel (Periodentafel) angegeben. Wir übertragen die Darstellung auf unser Abteilungs-Mitarbeiterbeispiel.

Demnach enthält der Baum als *Knotenarten*

- eine spezielle Wurzel (root), **nicht** identisch mit dem Wurzelelement des Dokuments
- Elemente
- Text
- Attribute
- Namensräume
- Prozessorinstruktionen
- Kommentare

Weiter oben (siehe Kap. 3: Document Type Definitions) haben wir mit `AM.xml` ein passendes Beispiel angegeben.

```
<?xml version='1.0' standalone="no" ?>
<?xml:stylesheet type="text/xsl" href="client8.xsl" ?>
<!DOCTYPE ABTEILUNGEN SYSTEM "AM.dtd">

<ABTEILUNGEN id="4000" >
<!-- der Firma Gott & Golem -->
  <ABTEILUNG id="4001">
    <AID id="4003">FE</AID>
    <ABTBEZ id="4004">Forschung&Entwicklung</ABTBEZ>
    <ORT id="4005">Dresden</ORT>
    <MITARBEITER id="4006">
      <MITARB id="4007">
```

```
<NAME id="4008">Peter</NAME>
<MID id="4009">1022</MID>
<GJAHR id="4010">1959</GJAHR>
  </MITARB>
  <MITARB id="4011">
<NAME id="4012">Uwe</NAME>
<MID id="4013">1172</MID>
<GJAHR id="4014">1978</GJAHR>
  </MITARB>
</MITARBEITER>
</ABTEILUNG>
<ABTEILUNG id="4020">
  <AID id="4021">HR</AID>
  <ABTBEZ id="4022">Personalabteilung</ABTBEZ>
  ...
</ABTEILUNG>
  ...
</ABTEILUNGEN>
```

Der zugehörige Baum in der Notation von [8] sieht wie unten gezeigt (Abbildung 5.1) aus.

Man erkennt daran, daß XSL eine über die Elementstruktur hinausgehende, feinere Sicht des Dokuments hat. Ebenfalls ist zu beachten, daß zu dem Baum auch die Attribute aus den Schemata gehören, die dort als #FIXED (in DTD Notation, "default" in XML Schema) definiert wurden und damit nicht explizit ins Dokument geschrieben werden müssen, sondern dort automatisch gelten.

Das von uns anzugebende Stylesheet ist zunächst wieder einmal ein XML-Dokument (Orthogonalität!). Es besteht aus einem Element mit Namen `xsl:stylesheet`. Dieses enthält wiederum Elemente namens `xsl:template`, die als Regeln aufzufassen sind. Der Begriff *template* deutet darauf hin, daß

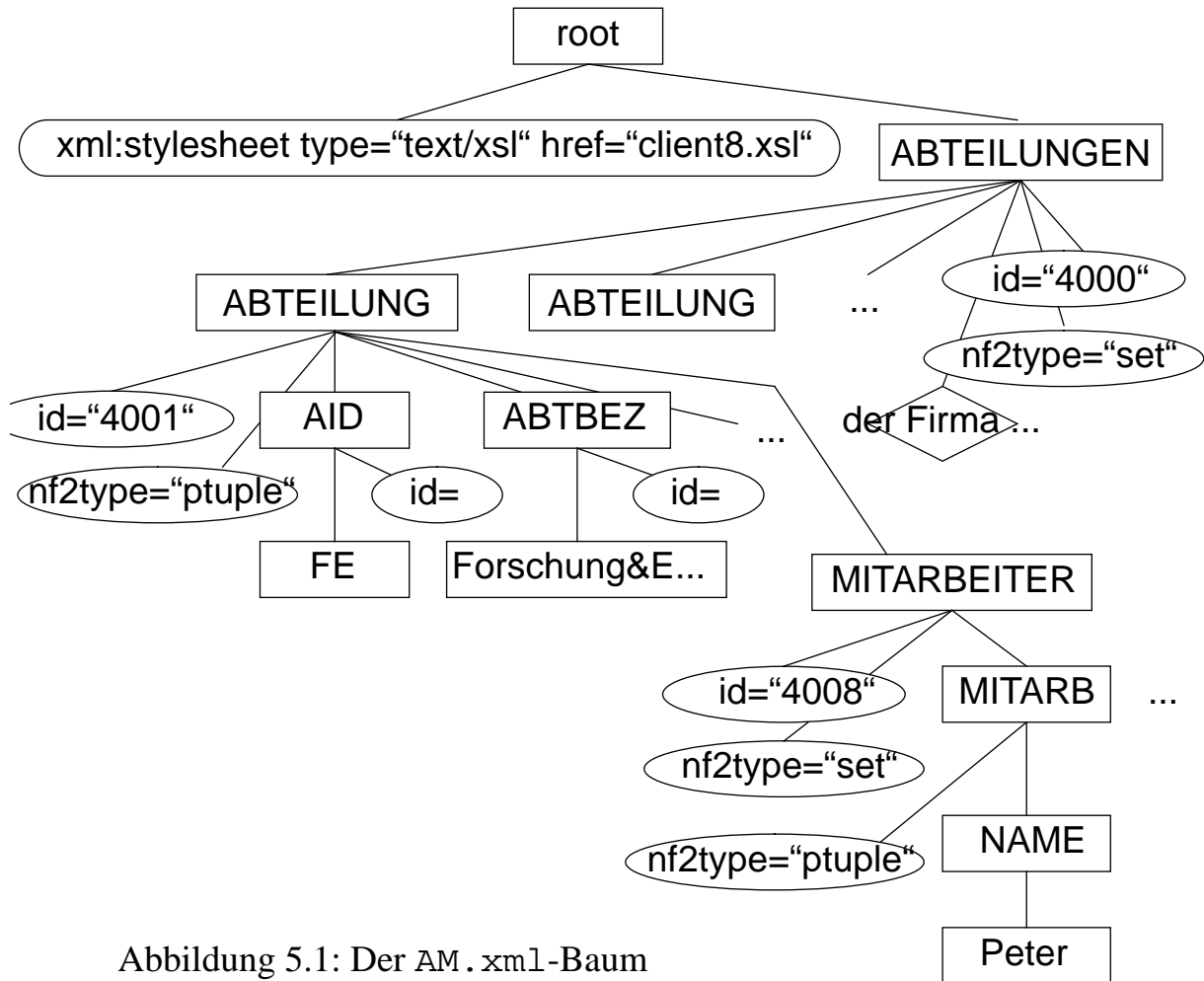


Abbildung 5.1: Der AM.xml-Baum

die Anwendung der Regeln durch Muster gesteuert wird. Dieses Muster wird als Attributwert zu `match="Muster"` ins Template geschrieben.

Der XSL-Transformierer (als Teil eines Browsers, oder getrennt wie in XT) durchläuft nun den XML-Baum und versucht, an jedem Knoten eine passende Regel zu finden, wobei bei mehreren passenden die spezifischste (genauest passende) genommen wird. Der Inhalt des `<xsl:template>` Elements bestimmt, welche Ausgabe produziert wird. Zum zweiten hängt die Ausgabe vom Inhalt des Elements ab, auf das die Regel `<xsl:template>` paßt. Gibt es in diesem Inhalt wei-

tere Elemente können diese ggf. ebenfalls durch passende Regeln interpretiert werden.

Diese Regeln können z.B. angeben, daß der Wert (Inhalt) des gerade betrachteten Knotens in die Ausgabe kopiert wird (`<xsl:value-of>`) oder dessen Name (`<xsl:node-name>`). Bei Elementen, die wiederum Unterelemente enthalten, die gesondert zu behandeln sind, wird man in den Regeln `<xsl:template>` die Anweisung für den Übersetzer `<xsl:apply-templates>` finden, die angibt, daß alle direkten Unterknoten (children) des gerade betrachteten Knotens zu verarbeiten sind. Damit prüft der Übersetzer für jeden enthaltenen Sohn, ob es eine passende Regel gibt und wendet diese ggf. rekursiv an. Auf eine weitere Möglichkeit, dies iterativ über `<xsl:for-each>` zu steuern, gehen wir unten ein. Bevor die Erläuterungen der Möglichkeiten zu undurchschaubar werden, betrachten wir ein einfaches Beispiel. Dazu erzeugen wir ein Stylesheet `AM1.xsl` mit dem folgenden Inhalt.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

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

  <xsl:template match="ABTEILUNGEN">
    <BODY STYLE="font-family:Arial, helvetica,
      sans-serif; font-size:8pt;
      background-color:#FFFFFF">
      <xsl:apply-templates/>
    </BODY>
  </xsl:template>
</xsl:stylesheet>
```

```
<HR></HR>
  </BODY>
</xsl:template>

<xsl:template match="ABTEILUNG">
  <P>
    <xsl:value-of/>
  </P>
</xsl:template>

</xsl:stylesheet>
```

Dieses Stylesheet wenden wir auf unser Abteilungen-Mitarbeiter-Dokument an. Weil wir in diesem XML-Dokument die Verknüpfung zu AM1.xsl herstellen mittels `<?xml:stylesheet type="text/xsl" href="AM1.xsl" ?>`, legen wir die so geänderte Version als AM1.xml in das Quellenverzeichnis der Vorlesung. Welche Ausgangsversion für AM1.xml genommen wird, also ob AM.xml mit DOCTYPE oder AMS.xml mit XML-Schema, ist gleichgültig, da beide unter dem alten Stylesheet Client8.xsl gut funktioniert haben und validiert wurden.

Das etwas einfachere Stylesheet AM1.xsl besteht aus drei Templates (Regeln). Die erste Regel paßt nur auf die Wurzel, die traditionell „/“ heißt. Der Inhalt besagt, daß wir den HTML-Starttag `<HTML>` (natürlich auch `<html>`) und den zugehörigen Endetag `</HTML>` ausgeben. Dazwischen soll alles das stehen, das wir rekursiv durch Anwendung aller Regeln auf die direkt unterhalb der Wurzel stehenden Knoten (das ist bei uns nur ABTEILUNGEN) an Ausgabe erzeugen.

Die zweite Regel läßt sich nur auf unser Wurzelement ABTEILUNGEN anwenden. Sie produziert analog das HTML `<BODY>...</BODY>` Pärchen, dekoriert mit einigem Schnick-Schnack für das Look&Feel. Wesentlich ist wieder das `<xsl:apply-templates>`, das diesmal die vier ABTEILUNG-Elemente betrachtet.

Diese werden mit der dritten Regel „gefangen“. Sie erzeugt ein HTML-Paragraph Element (`<P>...</P>` im XML-Stil!).

Dazwischen produzieren wir mit `<xsl:value-of>` den Inhalt des ABTEILUNG-Elements. Dieser Inhalt setzt sich aus den gesamten Texten (Inhalten) aller darin enthaltenen Elemente zusammen. Diese werden jeweils durch ein Leerzeichen in der Ausgabe getrennt, jedoch nicht weiter formatiert.

Hinweis: In der XML-Bibel von Harold [8] fehlt dieses `<value-of>` Element im Stylesheet zur Periodentafel. In `AM1.xsl` wird dann nichts produziert außer der waagrechten Linie (`<HR>`). Harold erklärt ([8], S. 442f), das Standardverhalten für Elemente, auf die kein Template paßt, sei, ihren Inhalt auszugeben. Harold sagt weiter, IE5 unterstütze dies nicht, d.h. für IE5 sei die Aussage, bereits das `xsl:apply-templates` erzeuge die Inhalte, falsch, es wird nur das Template-matching angestoßen.

Die Ausgabe der Abteilungsdaten im IE5 ist jedenfalls recht schlicht. Im Folgenden werden wir weitere, ansprechendere Formatierungen erzeugen.

FE Forschung&Entwicklung Dresden Peter 1022 1959 Uwe 1172 1978

HR Personalabteilung Kassel Gabi 1017 1963 Rolf 1298 1983

EC e-Commerce ?

VB Vertrieb&Beratung Dresden Beate 1305 1978

5.2 Strategien zum Einsatz von Transformationen

Harold [8] weist darauf hin, daß es drei Strategien zum Einsatz der Übersetzungen von XML-Dokumenten in Verbindung mit XSL-Stylesheets gibt:

- Versand an einen Browser beim Clienten, der XML+XSL „direkt“ verstehen und anzeigen kann; „direkt“ schließt dabei auch einer browserseitige Umwandlung in HTML oder CSS ein. Die Verwendung von IE5 ist hierfür ein Beispiel.
- Bei Eingang einer XML-Seitenanfrage wird auf dem Server eine Übersetzung angestossen, die z.B. HTML erzeugt; dazu kann die Seite ein Servlet enthalten. Das Resultat wird an den Clienten-Browser verschickt. IBM alphaWorks ist ein solches Web-Server Werkzeug (<http://www.alphaworks.ibm.com/tech/LotusXSL>).
- Die XML-Seite wird vorab in z.B. HTML übersetzt und diese auf dem Server abgelegt. Ein Tool dafür ist der von

James Clark stammende Übersetzer XT, der eine Java-Anwendung ist (<http://www.jclark.com/xml/xt.html>). Unter Linux haben wir dies installiert. Der Aufruf erfolgt mit

```
xt Quelltext.xml Stylesheet.xsl Zieltext.html
```

Zuletzt sei noch ein Hinweis aus Homer [7] erwähnt. XSL-Stylesheets können Skripte enthalten (Code). Daher dürfen sie nur vom dem selben URL geladen werden, von dem das XML Dokument stammt, auf das sie angewandt werden. Diese Sicherheitsregel ist analog zur Angabe für `src` im `<SCRIPT>` Element von HTML.

5.3 Templates und Muster

Wir experimentieren zunächst mit den vier XSL-Elementen

- `<xsl:template>`
- `<xsl:apply-templates>`
- `<xsl:value-of>` und
- `<xsl:node-name>`

`<xsl:template>`

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

Die `<xsl:template>`-Elemente sind die eigentlichen Verarbeitungsregeln. Das im Starttag angegebene Muster „*pattern*“ bestimmt, auf welche XML-Elemente das Template angewandt wird. Die Mustererkennung ist sehr flexibel und

kann Pfadausdrücke, Attributnamen und -werte enthalten.

Wie die Ausgabeformatierung aussehen soll, steht zwischen Start- und Endtag. Eckstein [9] gibt ein Beispiel, das zwar mit *formatting objects* arbeitet (`fo:block` definiert einen rechteckigen Textbereich, z.B. für einen Absatz), aber ganz gut den Stil erläutert.

```
<xsl:template match="para">
  <fo:block font-size="12pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Wichtig an diesem Beispiel ist auch die Fomatierungsanweisung `<xsl:apply-templates/>`, die den Prozessor zwingt, im zu formatierenden Element alle auftretenden Sohnelemente ebenfalls dem Template-matching zu unterziehen.

Wird im Beispiel von oben

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
  <HTML>
  <xsl:apply-templates/>
  </HTML>
</xsl:template>
...
</xsl:stylesheet>
```

die unterlegte Zeile herausgenommen, erhalten wir nur ein leeres HTML-Dokument (es enthält nur den `<HTML />` tag). Trotz der anderen vorhandenen Regeln wurden diese nicht angewandt.

Setzen wir statt `<xsl:apply-templates>` die Anweisung `<xsl:value-of />` ein, erhalten wir den gesamten Textinhalt des XML-Dokuments, allerdings unformatiert als eine Zeile hintereinander (die HTML-Browser brechen diese Zeile dann an den Leerstellen je nach Fensterbreite).

`<xsl:value-of />`

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

Das leere Element wirkt wie eine Anweisung, die verlangt, daß der Inhalt eines im `select`-Attribut angegebenen Elements zu nehmen und verbatim (wörtlich) auszugeben ist.

Gemäß der Syntaxangabe (so auch im Standard) muß ein `select`-Attribut angegeben werden. IE5 erlaubt auch das Weglassen und bezieht sich dann auf den gegenwärtigen Knoten (entspricht der Angabe `select="."`). Diese abkürzende Schreibweise ist sehr intuitiv und wird vermutlich von allen Browserentwicklern unterstützt werden.

Wie man am Beispiel erkennt, bezieht sich die wörtliche Inhaltsausgabe auch auf den Inhalt der geschachtelten Unter-elemente. Entities werden vor der Ausgabe ersetzt (siehe Forschung&Entwicklung, ...&...). Kommentare im XML-Dokument werden allerdings nicht ausgegeben.

Anders ist es, wenn der Knoten, der gerade zur Verarbeitung ansteht und für den die `<xsl:value-of>`-Formatieranweisung gilt, kein Element ist, sondern ein Attribut, ein Kommen-

tar, ein Namensraum, oder eine Prozessorinstruktion.

Dann ist der Wert, der ausgegeben wird, der Attributwert, der URI für den Namensraum, der Wert der Prozessorinstruktion ohne `<? und ?>`, bzw. der Text des Kommentars ohne `<!-- und -->`.

Experimentieren wir ein wenig. Zunächst ersetzen wir im ursprünglichen Stylesheet `AM1.xsl` im letzten `<xsl:value-of>` das `select`-Attribute auf `MITARBEITER`.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/W3-XSL">

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

  <xsl:template match="ABTEILUNGEN">
    <BODY STYLE="font-family:Arial, helvetica, sans-serif;
font-size:8pt;
      background-color:#FFFFFF">
      <xsl:apply-templates/>
      <HR></HR>
    </BODY>
  </xsl:template>

  <xsl:template match="ABTEILUNG">
    <P>
      <xsl:value-of select="MITARBEITER"/>
    </P>
  </xsl:template>

</xsl:stylesheet>
```

Wie zu erwarten, erhalten wir nur die Mitarbeiterdaten (MIT-

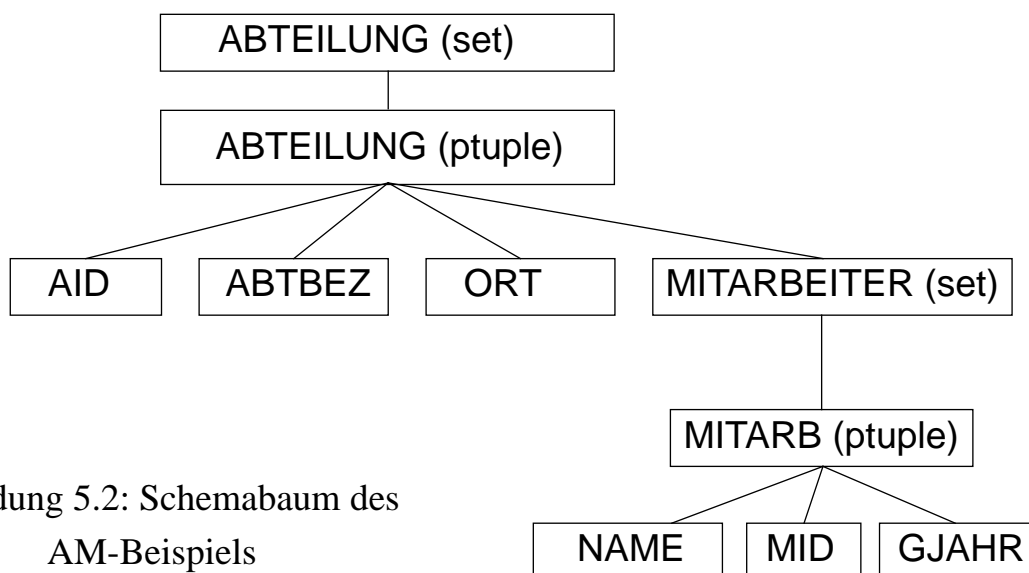
ARBEITER war die Menge der Mitarbeiter einer Abteilung).

Was passiert, wenn wir `select="MITARB"` schreiben?

Spielt es eine Rolle, daß dieses Element in den zwei ersten Abteilungen zweimal vorkommt?

Die Antwort lautet: Es kommt keine Textausgabe heraus, denn das Element `<MITARB>` wird nicht gefunden! Es liegt unterhalb von `<MITABEITER>`, das direktes Kind von `<ABTEILUNG>` ist. Ohne weitere Pfadangabe wird der Vergleich für das `select`-Muster aber nur mit den direkten Kindern des gegenwärtigen Knotens vorgenommen.

Ändern wir jetzt das Muster zu `select="*/MITARB"` wird für jede Abteilung der erste Mitarbeiter ausgegeben. Die Interpretation des Musters mit der Wildcard „*“ lautet: „nimm den Wert eines MITARB-Elements, das direkter Sohn eines *beliebigen direkten* Sohns des gegenwärtigen Knotens ist.“



Wollen wir nur die Abteilung ausgeben, deren `id`-Attribut auf

dem Wert 4020 steht (die HR-Abteilung in Kassel), setzen wir `select=" .[@id='4020'] "`. Alles klar?

Hier muß man wissen, daß „@“ (der at-Operator) der Teil des Musters ist, der auf Attribute des gegenwärtigen Knotens hinweist. Der führende Punkt bedeutet „im gegenwärtigen Knoten“, auch gegenwärtiger *Kontext* genannt. Die eckigen Klammern danach bestimmen ein spezielles Element innerhalb einer Kollektion. In unserem Fall gibt es ja vier `<ABTEILUNG>`-Elemente. Im einfachsten Fall schreibt man einen Index hinein, also z.B. `" . [2] "` für die 2. Abteilung innerhalb von `<ABTEILUNGEN>`. In unserem Fall erfolgt die Auswahl durch Vergleich mit dem Attributwert. Man beachte den Einsatz der einfachen Anführungszeichen.

Wollen wir dagegen die `id`-Attributwerte der Abteilungen, setzen wir `select="@id"`. Die Ausgabe sieht dann wie folgt aus:

4001

4020

4040

4060

Schreibt man `select=" .[@id] "` meint man alle gegenwärtigen Elemente, die ein `id`-Attribut haben (Test auf Existenz), damit wieder alle Abteilungen. Die Unterschiede sind dem-

nach sehr subtil. Auf die Syntax und Semantik der XSL-Muster müssen wir getrennt nochmals genauer eingehen.

<xsl:apply-templates>

```
<xsl:apply-templates [select="pattern"]/>
```

Gibt an, daß die unmittelbaren Söhne des gegenwärtigen Knotens weiter behandelt werden sollen. Über das Muster kann eine Auswahl getroffen werden (siehe Beispiel AM2.xsl unten).

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

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

  <xsl:template match="ABTEILUNGEN">
    <BODY STYLE="font-family:Arial, helvetica, sans-serif;
      font-size:8pt; background-color:#FFFFFF">
      <xsl:apply-templates select="//MITARB"/>
      <HR></HR>
    </BODY>
  </xsl:template>

  <xsl:template match="MITARB">
    <P><xsl:value-of /></P>
  </xsl:template>

</xsl:stylesheet>
```

In ABTEILUNGEN wollen wir nur die MITARB-Söhne weiterbehandeln. Diese sind keine direkten Söhne von ABTEI-

LUNGEN. Wir umgehen das, indem wir im Muster „//“ schreiben, wodurch Knoten beliebiger Tiefe ausgewählt werden. Zugleich ändern wir das letzte Template auf `match="MITARB"`.

Als Ergebnis erhalten wir eine Liste aller Mitarbeiter, also praktisch eine Projektion aus der AM-Tabelle heraus. Ändern wir zusätzlich noch das letzte Template zu

```
<xsl:template match="MITARB">
  <P>
    <xsl:value-of />
    <B><xsl:value-of select="../../../AID" /></B>
  </P>
</xsl:template>
```

erhalten wir in der Ausgabe auch wieder den Fremdschlüssel AID (fettgedruckt, wegen `...`).

Peter 1022 1959 **FE**

Uwe 1172 1978 **FE**

Gabi 1017 1963 **HR**

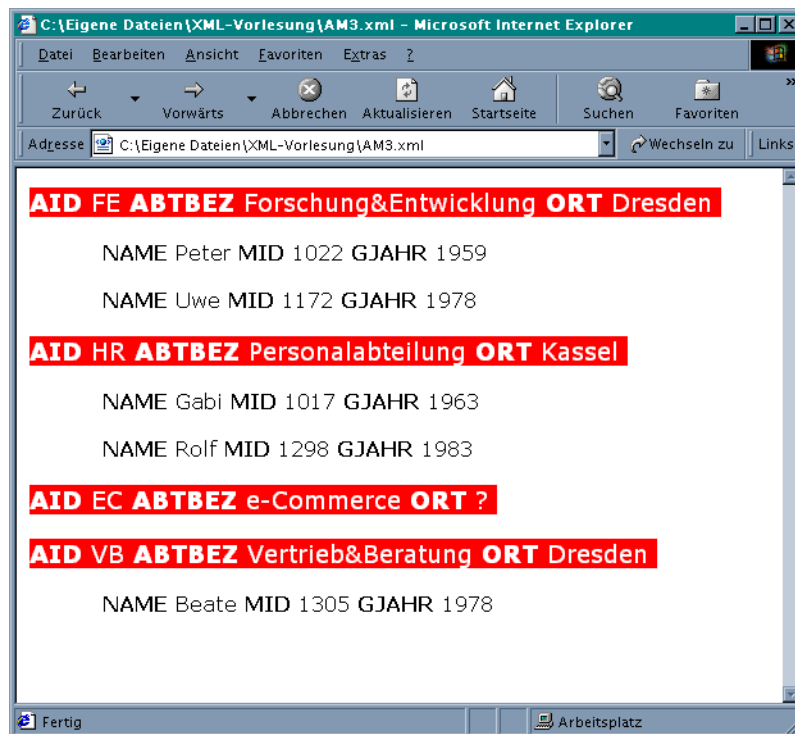
Rolf 1298 1983 **HR**

Beate 1305 1978 **VB**

Dieser Wert befindet sich zwei Stufen höher als MITARB im ABTEILUNG-Tupel, weshalb wir mit „..“ zum Vorgängerelement im Elementbaum hochgehen.

Man erkennt an diesem Beispiel, daß aus geschachtelten XML-Daten über Stylesheets datenbankartige „Abfragen“ programmierbar sind, hier praktisch eine „unnest-Operation“ der relationalen NF²-Algebra. Ob dies Sinn und Zweck eines Stylesheets ist, sei dahingestellt.

Bevor wir weitere XSL-Elemente und die Mustersyntax besprechen, wollen wir zur Entspannung eine etwas schönere Abteilungstabelle produzieren.



Das zugehörige Stylesheet (AM3 . xsl) sieht wie folgt aus.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <xsl:apply-templates/>
    </HTML>
  </xsl:template>
```

```
<xsl:template match="ABTEILUNGEN" >
  <BODY>
    <xsl:apply-templates />
  </BODY>
</xsl:template>

<xsl:template match="ABTEILUNG">
  <SPAN STYLE='font-size:14pt; font-family:Verdana;
    background-color:#FF0000; color:white'>
    <xsl:apply-templates />
  </SPAN>
</xsl:template>

<xsl:template match="MITARBEITER">
  <P/>
  <xsl:apply-templates />
</xsl:template>

<xsl:template match="MITARB" >
  <P style="margin-left:1.5cm;"/>
  <SPAN STYLE='font-size:12pt; font-family:Verdana;
    background-color:#FFFFFF; color:black'>
    <xsl:apply-templates />
  </SPAN>
  <P/>
</xsl:template>

<xsl:template match="*[$not$ @nf2type]">
  <B><xsl:node-name /></B>
  <xsl:value-of />
</xsl:template>
</xsl:stylesheet>
```

HTML-seitig fällt auf, daß wir mit SPAN arbeiten, wodurch sich Stylesheet-Angaben für CSS 1.0 direkt in HTML an beliebige Stellen einbauen lassen. Insbesondere für Färbungen und Fonts ist dies ein einfaches Formatierungsmittel. Für die Einrückung der Mitarbeiter brauchen wir nochmals CSS mit

style="margin-left:1.5cm". Den pseudo-HTML Tag `<spacer type="horizontal" size="100"/>` versteht IE5 nicht, nur N3 oder höher.

Auf Seiten von XSL ist der Test im letzten Template von
Beachtung: `match="*[not @nf2type]"`.

Damit werden durch , * 'alle Elemente, unabhängig vom Namen, ausgewählt, allerdings nur die, welche den in eckigen Klammern stehenden *Testausdruck* erfüllen. Dieser verlangt die Existenz eines Attributs mit Namen `nf2type`, unabhängig vom Wert des Attributs. Der Test wird mit `not` negiert. Also werden alle Elemente, die **kein** `nf2type`-Attribut besitzen, ausgewählt. Das sind bei uns gerade alle atomaren Elemente (AID, ABTBEZ, ORT, MID, NAME, GJAHR). Diese werden mit fettgedruckten Elementnamen `<xsl:node-name />` und Wert ausgegeben.

`<xsl:node-name />`

Diese Anweisung liefert den Namen des gegenwärtigen Knotens als Ausgabertext. `xsl:node-name` hat keine Attribute.

Die gerade genannten Tests lassen sich mit Vergleichen und Booleschen Verknüpfungen in Pfadausdrücken als Filter benutzen. Die Auswahl erfolgt nach

- Existenz eines Kind-Knotens
- Wert eines Knotens
- Existenz eines Attributs

- Wert eines Attributs
- einer Kombination dieser vier Kriterien.

Suchen wir etwa nach Abteilungen, die eine/n Mitarbeiter/in haben, die nach 1980 geboren ist (GJAHR > „1980“), dann formulieren wir:

```
<xsl:template
  match="ABTEILUNG[MITARBEITER/MITARB[GJAHR > `1980`]]">
  <SPAN STYLE='font-size:14pt; font-family:Verdana;
    background-color:#FF0000; color:white`>
    <xsl:apply-templates />
  </SPAN>
</xsl:template>
```

Das Muster fordert ein Abteilungselement, das ein Mitarbeiter-element enthält, das im Mitarb-Element ein Geburtsjahrelement enthält, dessen Wert (Inhalt) größer als die Zeichenkette 1980 ist.

Das Muster funktioniert nicht, wenn wir ABTEILUNG[. . .] durch ABTEILUNG/ . . . ersetzen! Möglich wäre dagegen, [MITARB[. . .]] statt /MITARB[. . .] zu schreiben.

Die Abfrage funktioniert auch für mehrere Treffer (wie man testen kann, indem man z.B. Beate einer Verjüngungskur unterzieht).

Ganz offensichtlich ist die Ähnlichkeit zu der folgenden Abfrage in SQL: select * from Abteilungen A where exists (select * from Mitarbeiter M where GJahr > 1980 and A.AID = M.AID)

☞ Welches Y10K Problem kündigt sich in der XSL-Lösung an?

6 Weitergehende XSL Konzepte

In diesem Kapitel sollen die Mustererkennung (pattern matching), die XSL-Filteroperationen und die weiteren XSL-Elemente besprochen werden. Mustererkennung und Filteroperation bestimmen, welche Knoten auf ein spezielles Template passen. Die Knotenangabe beruht auf zwei Methoden:

- eine Positionsangabe in der Baumstruktur
- die Angabe eines Filters, der einzelne Knoten auswählt.

6.1 XSL Pattern Matching durch Pfadangabe

Durch Pfadoperatoren können wir Pfadausdrücke, ausgehend von der Wurzel des Dokuments oder relativ zum gegenwärtigen Kontext, angeben. Ohne Pfadoperatoren beziehen sich die Angaben immer auf den gegenwärtigen (aktuellen) Knoten. Zu Anfang ist dies automatisch die Wurzel („/“).

Operator	Beschreibung
/	Der Sohnoperator trennt direkte Nachfolger vom Vaterknoten (wie in hierarchischen Dateisystemen üblich). Beginnt der Ausdruck mit „/“, dann beginnt der Pfad an der Wurzel des Dokumentbaums
//	Der Operator für den rekursiven Abstieg wählt Knoten beliebiger Tiefe unterhalb des gegenwärtigen aus. Beginnt der Ausdruck mit „//“, dann wählt man alle Knoten unterhalb der Wurzel aus.
.	Der Operator für den gegenwärtigen Kontext (Knoten). Nützlich für die Angaben der Art „./ <MITARB>“ (alle MITARB-Elemente unterhalb des gegenwärtigen Knotens). Die Angabe „./“ ist dagegen meist überflüssig.

Operator	Beschreibung
@	Der Attribute-Pfadoperator (at-Operator) gibt an, daß sich der Teil des Pfadausdrucks auf ein Attribut des gegenwärtigen Knotens (der ein Element sein muß) bezieht. Er kann mit dem Attributnamen nur am Schluß des Pfadausdrucks stehen.
*	Der Wildcard-Operator paßt auf alle Elemente oder Attribute unabhängig von ihrem Namen. So wählen wir mit <code><ABTEILUNG> / *</code> alle Sohnelemente aller ABTEILUNG-Elemente, mit <code><ABTEILUNG> / @*</code> alle Attribute aller ABTEILUNG-Elemente.

Die Pfadausdrücke liefern alle Knoten, die auf das Muster passen. Mit einem Index kann man unter ihnen spezielle auswählen und mit der XSL-Funktion `end()` gerade den letzten Knoten. Der erste, zweite, letzte Mitarbeiter in der Menge der Mitarbeiter einer Abteilung wäre demnach

```
<MITARBEITER> / <MITARB> [ 0 ]
<MITARBEITER> / <MITARB> [ 1 ]
<MITARBEITER> / <MITARB> [ end( ) ]
```

Genauso kann man aber auch in `AM4.xsl` das letzte Element innerhalb der Kollektionen von atomaren Elementen (solche ohne `nf2type`-Attribut) angeben:

```
<xsl:template match match="*[$not$ @nf2type][end( )]">
```

6.2 XSL Filter und Filtermuster

Wie bereits im Kapitel 5 erwähnt, kann die Auswahl auf Elementnamen, Elementwerte, Attributexistenz oder Attributwert (und allen Kombinationen davon) beruhen. Die generelle Form ist `[Operator Muster]`, wobei der optionale Operator

bestimmt, wie das Muster anzuwenden ist. Das Muster selbst gibt einen der gerade genannten Auswahlwerte/-namen an. Operator und Muster sind durch mindestens einen Whitespace getrennt. Der Operator selbst kann wieder durch mehrere Operatoren (einen Operatorausdruck) dargestellt werden. Fehlt er, werden alle Knoten genommen, die auf das Muster passen. Im Beispiel unten wählen wir in `AM4.xsl` nur die Abteilungen aus, die in Dresden beheimatet sind.

```
<xsl:template match="ABTEILUNG[ORT = 'Dresden']">
```

Homer [7] entnehmen wir die Liste der Vergleichsoperatoren.

Abkürz.	Operator	Beschreibung
=	<code>\$eq\$</code>	case-sensitive Gleichheit, z. B. <code>[GJAHR = 1980]</code>
!=	<code>\$ne\$</code>	case-sensitive Ungleichheit, z. B. <code>[ORT != '??']</code>
< (<i>Anmerkung!</i>)	<code>\$lt\$</code>	case-sensitive kleiner-als, z. B. <code>[GJAHR \$lt\$ 1950]</code>
<= (<i>Anmerkung</i>)	<code>\$le\$</code>	case-sensitive kleiner-oder-gleich, z. B. <code>[UMSATZ \$le\$ 999]</code>
>	<code>\$gt\$</code>	case-sensitive größer-als, z. B. <code>[GJAHR > 1980]</code>
>=	<code>\$ge\$</code>	case-sensitive größer-oder-gleich, z. B. <code>[NAME >= 'Rolf']</code>
	<code>\$ieq\$</code>	case-insensitive Gleichheit
	<code>\$ine\$</code>	...
	<code>\$ilt\$</code>	...
	<code>\$ile\$</code>	...
	<code>\$igt\$</code>	...
	<code>\$ige\$</code>	...

Anmerkung: Die Abkürzungen < und <= können wegen der

XML-Syntax nicht ohne Escape-Zeichen direkt eingegeben werden.

Man beachte auch, daß bei einigen Vergleichen mit numerischen Werten die Anführungszeichen weggelassen wurden. IE5 nimmt eine „typgerechte“ Umwandlung vor, abhängig vom Schema (sagt Homer, fügt man aber im Schema AMschema.xml, das für AM4.xml benutzt wird, in

```
<ElementType name="GJAHR" content="textOnly" model="closed"
  dt:type="string">
```

ein, dann funktioniert der Vergleich [GJAHR > 1980] trotzdem).

Sehr bedenklich ist auch das Resultat des folgenden Experiments. Wir setzen den Datentyp von GJAHR auf `dt:type="number"` und fügen in das Stylesheet den Fehler `match="MITARB[GJAHR > `19H0`]"` ein. Das Ergebnis ist die Liste genau der Abteilungen, die keine Mitarbeiter haben!

Man beachte auch, daß die Operatoren selbst *case sensitive* sind, also kleingeschrieben werden müssen.

Wie zu erwarten, kann man die Vergleiche mit Booleschen Operatoren mischen.

Abkürz.	Operator	Beschreibung
&&	<code>\$and\$</code>	logisches UND
	<code>\$or\$</code>	logisches ODER
	<code>\$not\$</code>	logische Negation

Zusätzlich können wir noch *Quantoren* einsetzen: *any* und *all*. Homer ([7], Seite 424) nennt sie *Filter Set Operatoren*.

Wenn wir uns an das Beispiel aus Kapitel 5 erinnern, in dem Abteilungen gesucht waren, die einen Mitarbeiter hatten, der nach 1980 (GJAHR > 1980) geboren war, so entsprach dies dem Existenzquantor *any*.

Wir können einfach schreiben

```
<xsl:template match="ABTEILUNG[MITARBEITER/MITARB[GJAHR > 1980]]">
```

oder davor ein `any` setzen. Das Ergebnis ist immer die HR-Abteilung, weil ihr Mitarbeiter Rolf 1983 geboren ist.

```
<xsl:template match="ABTEILUNG[$any$ MITARBEITER/MITARB[GJAHR > 1980]]">
```

Setzen wir für `any` den *Allquantor* `all` ein, sind die Ergebnisse überraschend.

```
<xsl:template match="ABTEILUNG[$all$ MITARBEITER/MITARB[GJAHR > 1980]]">
```

```
<xsl:template match="ABTEILUNG[MITARBEITER/MITARB[$all$ GJAHR > 1980]]">
```

```
<xsl:template match="ABTEILUNG[MITARBEITER[$all$ MITARB[GJAHR > 1980]]]>
```

Keines der drei Muster verändert die Ausgabe. Die beiden ersten sind bei näherer Betrachtung unsinnig, da ABTEILUNG nur ein MITARBEITER-Element enthalten kann. Genauso hat MITARB nur ein GJAHR-Element, das entweder größer oder kleiner-gleich 1980 sein kann. Der Allquantor macht in diesen Testausdrücken keinen Sinn. MITARBEITER hat aber mehrere MITARB-Elemente. Sollen alle diese Mitarbeit nach 1980

geboren sein, dann wäre der dritte Ausdruck eigentlich der richtige. Man kann daher davon ausgehen, daß dieser Quantor nicht richtig implementiert ist. Die Standardempfehlung XSLT vom Nov. 99 [3] kennt entsprechend `any` und `all` nicht.

6.3 Eingebaute Methoden

Homer [7] nennt eine Reihe von nützlichen Funktionen, die als Teil von Mustern aufgerufen werden können. Der XSLT-Standard kennt diese jedoch nicht, ggf. sind sie in andere Teile (z.B. DOM, *Document Object Model*, ein Hauptspeicherparser der Objektbäume erzeugt) gewandert. Wir geben deshalb hier nur summarisch die Methoden an.

Name	Beschreibung
<code>end ()</code>	wählt den letzten Knoten einer Kollektion aus
<code>index ()</code>	liefert den Index (Nummer) des gegenwärtigen Knotens einer Kollektion
<code>nodeName ()</code>	liefert den Elementnamen (tag name) des gegenwärtigen Knotens inklusive des Namensraum-Präfixes
<code>nodeType ()</code>	liefert im numerischen Format den Knotentyp zurück (wie im DOM)
<code>date ()</code>	liefert einen Wert im Datumsformat
<code>text ()</code>	liefert den Textinhalt des gegenwärtigen Knotens
<code>value ()</code>	liefert einen typgewandelten Wert des gegenwärtigen Knotens

Die `value ()` Methode ist dabei die Standardvorgabe. Entsprechend sind die Muster

ABTEILUNG[AID = "HR"] und

`ABTEILUNG[AID!value() = "HR"]`

gleichwertig. Das Ausrufezeichen (bang operator) trennt die Methode vom Knotennamen; der sonst in Programmiersprachen übliche Punkt ist wegen der Verwechslung mit dem gegenwärtigen Knoten nicht verwendbar.

Homer nennt auch als Beispiel (angepaßt auf unsere AM-Tabelle):

`ABTEILUNG[3]` gleichwertig mit

`ABTEILUNG[index() = 3]`,

Nützlich wäre eine Anwendungen der Art

`ABTEILUNG[index() > 3 and index() < 7]`.

Homer nennt die obigen Methoden Informationsmethoden im Gegensatz zu den folgenden „Kollektionsmethoden“.

Name	Beschreibung
<code>ancestor()</code>	wählt den am nächsten liegenden Knoten oberhalb des gegenwärtigen Knotens aus, der auf das Muster paßt; liefert genau einen Knoten oder den Wert null
<code>attribute()</code>	liefert alle Attribute des gegenwärtigen Knotens als Kollektion zurück; ein optionaler Parameter kann den gewünschten Attributnamen angeben
<code>comment()</code>	liefert als Kollektion alle Söhne ab, die Kommentarknoten sind
<code>element()</code>	liefert alle Sohnelemente des gegenwärtigen Knotens als Kollektion ab; der optionale Parameter kann ein gewünschtes Element angeben
<code>node()</code>	liefert als Kollektion alle Söhne ab, die keine Attribute sind
<code>pi()</code>	liefert als Kollektion alle Söhne ab, die Prozessorinstruktionen sind
<code>textnode()</code>	liefert als Kollektion alle Söhne ab, die Textknoten sind

Anwendungen wären

`ABTEILUNGEN/comment ()`

liefert uns den Kommentar (ohne Entity-Replacement: Firma Gott & Golem Inc.),

`ABTEILUNGEN/ABTEILUNG` und

`ABTEILUNGEN/element ('ABTEILUNG')`

liefern uns jeweils die ABTEILUNG-Söhne von ABTEILUNGEN. Man beachte dabei auch die Notwendigkeit der Anführungszeichen.

6.4 Weitere XSL Elemente

Neben den bereits erwähnten Elementen

```
xsl:template
xsl:apply-templates
xsl:value-of
xsl:node-name
```

gibt es eine Reihe weiterer Stylesheet-Elemente.

xsl:copy

```
<xsl:copy>...</xsl:copy>
```

Dieses Element kopiert den gegenwärtigen Knoten in die Ausgabe. Anders als `xsl:value-of` wird jedoch nicht der Textinhalt des Knotens und der seiner Kinder kopiert, sondern nur Start- und Endtag. Ohne weitere Angabe werden auch nicht die Kinderknoten, die Attribute oder andere Knoteninhalte ausgegeben. Allerdings kann man dies dadurch steuern, daß man zwischen `xsl:copy`-Starttag und -Endtag ein `xsl:apply-`

templates streut.

Die gängige Anwendung für `xsl:copy` ist die Umwandlung eines XML-Baums in einen geringfügig modifizierten Baum, z.B. einen, dem Attribute fehlen oder der zusätzliche bekommt.

Das folgende einfache Stylesheet `AM5.xsl` erzeugt z.B. eine sog. Inorder-Traversierung¹ des XML-Baums, wenn man unter „Behandlung der Wurzel“ die Ausgabe des gesamten Unterbaumtextes mittels `value-of` versteht.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

<xsl:template match="/">
  <HTML>
    <BODY>
      <xsl:apply-templates/>
    </BODY>
  </HTML>
</xsl:template>

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

</xsl:stylesheet>
```

Die Ausgabe im IE5 ist zunächst ein endlos langer Textsalat.

FE Forschung&Entwicklung Dresden Peter 1022 1959 Peter 1022 1959
 Uwe 1172 1978 Uwe 1172 1978 Peter 1022 1959 Uwe 1172 1978 FE For-
 schung&Entwicklung Dresden Peter 1022 1959 Uwe 1172 1978 HR Per-
 sonalabteilung Kassel Gabi 1017 1963 Gabi 1017 1963 Rolf 1298 1983

1. Inorder Traversierung (L W R) = für alle Knoten gilt rekursiv: behandle linken Teilbaum, behandle Wurzel, behandle rechten Teilbaum

Rolf 1298 1983 Gabi 1017 1963 Rolf 1298 1983 HR Personalabteilung
 Kassel Gabi 1017 1963 Rolf 1298 1983 EC e-Commerce ? EC e-Com-
 merce ? VB Vertrieb&Beratung Dresden Beate 1305 1978 Beate 1305
 1978 Beate 1305 1978 VB Vertrieb&Beratung Dresden Beate 1305 1978
 FE Forschung&Entwicklung Dresden Peter 1022 1959 Uwe 1172 1978
 HR Personalabteilung Kassel Gabi 1017 1963 Rolf 1298 1983 EC e-
 Commerce ? VB Vertrieb&Beratung Dresden Beate 1305 1978

Bei Betrachtung der XSL-Ausgabe erkennt man die von
`xsl:copy` erzeugten Tags, die HTML natürlich ignoriert.
 Man sieht aber dann gut, wie der XML-Parser depth-first den
 Baum durchkämmt. Wesentlich ist, daß das `value-of-Element`
 hinter dem `apply-templates-Element` steht.

```
<HTML>
<BODY>
<ABTEILUNGEN>
<ABTEILUNG>
<AID>
FE</AID>
<ABTBEZ>
Forschung&Entwicklung</ABTBEZ>
<ORT>
Dresden</ORT>
<MITARBEITER>
<MITARB>
<NAME>
Peter</NAME>
<MID>
1022</MID>
<GJAHR>
1959</GJAHR>
Peter 1022 1959</MITARB>
<MITARB>
<NAME>
Uwe</NAME>
<MID>
1172</MID>
<GJAHR>
```

```

1978</GJAHR>
Uwe 1172 1978</MITARB>
Peter 1022 1959 Uwe 1172 1978</MITARBEITER>
FE Forschung&Entwicklung Dresden Peter 1022 1959 Uwe
1172 1978</ABTEILUNG>
<ABTEILUNG>
<AID>
HR</AID>
<ABTBEZ>
Personalabteilung</ABTBEZ>
.
.
.
</ABTEILUNG>
...
</ABTEILUNGEN>
</BODY>
</HTML>

```

Weitere Methoden um Knoten in der Ausgabe zu erzeugen sind `xsl:attribute`, `xsl:cdata`, `xsl:comment`, `xsl:entity-ref`, `xsl:pi`.

xsl:attribute

```
<xsl:attribute name="Name">...</xsl:attribute>
```

Erzeugt einen Attributknoten in dem Element, in dem es steht. Das Attribut hat den name-Namen und sein Wert bestimmt sich aus dem Inhalt des `xsl:attribute`-Elements.

So wird im folgenden Land-Element ein Kennzeichen-Attribut eingefügt.

```

<xsl:element name="LAND">
  <xsl:attribute name="CODE">DE</xsl:attribute>
  <xsl:text>Deutschlands Internetkuerzel</xsl:text>
</xsl:element>

```

Im ersten Kapitel wurde angedeutet, daß die Verwendung von

Attributwerten anstelle von Elementinhalten eine Frage des Geschmacks ist. Prinzipiell kann man alle Elemente `<item/>` nennen und alles andere, also speziell Namen und Inhalt, über entsprechende Attribute `NAME= "... "` und `CONTENT= "... "` regeln.

Eine automatisierte Konvertierungen (generisch, d.h. schema-unabhängig) mit XSL-Stylesheets erscheint möglich.

```
<xsl:template match="*">
  <xsl:element name="ITEM">
    <xsl:attribute name="NAME">
      <xsl:node-name select="."/>
    </xsl:attribute>
    <xsl:attribute name="CONTENT">
      <xsl:value-of select="."/>
    </xsl:attribute>
  </xsl:element>
</xsl:template>
```

xsl:cdata

Hiermit lassen sich XML CDATA-Sections erzeugen, die nicht weiter verarbeitet werden. So erzeugt

```
<xsl:cdata><P /></xsl:cdata>
```

im Stylesheet die XSL-Ausgabe

```
<![CDATA[<P />]]>
```

(soweit so gut), was aber im IE5 in der HTML-Interpretation zur Anzeige von `...]]>... führt. :-)`

xsl:comment

```
<xsl:comment>Ohne Kommentar</xsl:comment>
```

erzeugt den Knoten `<!--Ohne Kommentar-->`.

xsl:element

```
<xsl:element name="Name | URI#Name">...</xsl:element>
```

Die Erzeugung von Elementknoten haben wir bereits mehrfach ausgenutzt. Die Variante mit URI erzeugt einen Namen in einem Namensraum.

xsl:entity-ref

```
<xsl:entity-ref name="Entity_Ref_Name">...</xsl:entity-ref>
```

Erzeugt einen Entity-Reference-Knoten, dessen Wert der Inhalt des `xsl:entity-ref`-Elements ist.

xsl:pi

```
<xsl:pi name="xml">version='1.0'</xsl:pi>
```

erzeugt z.B. die wohlbekannte XML-Prozessorinstruktion

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

Hinweis: Es ist ziemlich einfach, ein XML Dokument mit einem XSL-Stylesheet anzugeben, daß ein nahezu identisches, ggf. größeres XML-Dokument erzeugt, das mit dem selben Stylesheet erneut ein identisches, größeres XML-Dokument erzeugt, das ... Wird in einem Browser die erzeugte Ausgabe sofort wieder in einen XML-Parser geleitet, ist der Crash vorprogrammiert. Was macht IE5?

Zuletzt besprechen wir noch einige Kontroll-Elemente, mit

denen sich prozedural programmieren läßt.

xsl:for-each

```
<xsl:for-each select="Muster" [order-by="Muster-Liste"]>...  
</xsl:for-each>
```

Diese Schleifenkonstruktion wendet die im Inhalt angegebenen Formatieranweisungen (hier ...) nacheinander auf alle Söhne des gegenwärtigen Knotens an. Über das Muster kann zusätzlich eine Auswahl getroffen werden (nur solche mit einem speziellen Wert, mit einem Attribut, usw.). Obwohl die Schleife meist mit Templates ersetzbar wäre, ist es oft leichter nach einer Auswahl des gesuchten Vaters mit `xsl:template` dessen Söhne gezielt mit der `for`-Schleife zu bearbeiten. Über `order-by` kann die Reihenfolge der Bearbeitung gesteuert werden (Sortierung).

xsl:if

```
<xsl:if match="Muster">bedingte Ausgabe</xsl:if>
```

Hiermit läßt sich bedingte Ausgabe erzeugen. Ähnlich läßt sich mit `xsl:choose` wie in einer `case`-Anweisung eine Fallunterscheidung programmieren.

xsl:choose

```
<xsl:choose>...</xsl:choose>
```

Dieses Element wird in aller Regel mehrere Fallunterscheidungen (Dijkstra's guarded commands) anbieten, die mit

`<xsl:when match="...">` eingeleitet werden. Zuletzt gibt man ein `xsl:otherwise`-Element an, um nichtgefangene Knoten zu verarbeiten.

xsl:when

```
<xsl:when match="Muster">Anweisungen für den Trefferfall</xsl:when>
```

xsl:otherwise

```
<xsl:otherwise>Anweisungen sonst</xsl:otherwise>
```

Damit müßte sich jetzt auch das öfters zitierte generische Stylesheet `Client8.xsl` verstehen lassen.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<!-- xsl:import href="common.xsl"/ -->

  <xsl:template match="/">
    <HTML>
      <BODY STYLE="font-family:Arial, helvetica,
                sans-serif; font-size:8pt;
                background-color:#FFFFFF">
        <HR></HR>

        <xsl:apply-templates/>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="*[@nf2type = 'ptuple']">
    <xsl:for-each >
      <TD>
        <xsl:apply-templates select='.'/>
      </TD>
    </xsl:for-each>
  </xsl:template>
```

```

<xsl:template match="*[@nf2type = ,gtuple']">
  <!-- a so called genuine tuple =
        not part of a set or list -->
  <!-- create a form like display in rows with 2 columns -->
  <TABLE BORDER="2" FRAME="BOX">
    <xsl:for-each select="./node()">
      <xsl:value-of select=".[\$not$ @nf2type]"/>
      <TR>
        <TD VALIGN="TOP"><B><xsl:node-name /></B></TD>
        <TD>
          <xsl:value-of select=".[\$not$ @nf2type]"/>
          <xsl:apply-templates select=".[@nf2type]"/
        >
          </TD>
        </TR>
      </xsl:for-each>
    </TABLE>
  </xsl:template>

<xsl:template match="*[@nf2type = ,list`
  || @nf2type = ,set']">
  <TABLE BORDER="2" FRAME="BOX">
    <xsl:for-each select="./node()">
      <TR>
        <xsl:apply-templates select='.'/>
      </TR>
    </xsl:for-each>
  </TABLE>
</xsl:template>

<xsl:template match="*[\$not$ @nf2type]">
  <!-- all atomic values become a table field -->
  <!-- unless specified otherwise -->
  <B><xsl:node-name select="."/>:
  </B><xsl:value-of select="."/>
</xsl:template>

</xsl:stylesheet>

```

Somit fehlen uns als wesentliche Elemente nur noch
xsl:script und **xsl:eval**.

In HTML läßt sich mit `<script>` ein Code-Segment angeben. Dieses Element ist in XML aber ohne Bedeutung. Homer [7] nennt `xsl:script` und `xsl:eval` als Ausweg, diese Elemente sind im XSLT-Standard aber nicht enthalten.

xsl:script

```
<xsl:script language="Script-language">...</xsl:script>
```

Der so angegebene Code läßt sich von überall im Dokument aus aufrufen. Das Beispiel von Homer erklärt den Sachverhalt am besten.

```
<xsl:script language="VBScript">
  Function getDateime()
    getDateime = Now()
  End Function
</xsl:script>
```

xsl:eval

```
<xsl:eval language="Script-language">... </xsl:eval>
```

Das Element bringt den Inhalt zur Ausführung und fügt den dabei gelieferten Text an dieser Stelle in die Ausgabe ein.

```
<xsl:eval language="VBScript">getDateime()</xsl:eval>
```

Alternativ hätte man den Aufruf von `Now()` auch direkt einfügen können.

```
<xsl:eval language="VBScript">Now()</xsl:eval>
```

Anmerkung: Die zuletzt gezeigte Variante wurde für IE5 getestet und funktioniert.

6.5 Variablen, Konstanten und benannte Templates

Tendenziell kann man in XSL mit symbolischen Konstanten arbeiten. Die XML-Bibel [8] nennt dafür `<xsl:variable>`, bei Eckstein [9] heißt der Konstrukt `<xsl:constant>`, beides funktionierte nicht in IE5.

Wir geben hier ein Beispiel aus dem neuen XSLT-Standard an.

The following example creates an `img` result element from a `photograph` element in the source; the value of the `src` attribute of the `img` element is computed from the value of the `image-dir` variable and the string-value of the `href` child of the `photograph` element; the value of the `width` attribute of the `img` element is computed from the value of the `width` attribute of the `size` child of the `photograph` element:

```
<xsl:variable name="image-dir">/images</xsl:variable>

<xsl:template match="photograph">

</xsl:template>
```

With this source

```
<photograph>
  <href>headquarters.jpg</href>
  <size width="300"/>
</photograph>
```

the result would be

```

```

Attraktiv wäre hier eine Kombination von Variablen und Ausdrücken im Stil von Tcl mit den Template-Regeln von XSL.

7 Ausblick

Man könnte an dieser Stelle jetzt die XML Komponenten DOM, XPath, XPointer und XLink besprechen. Genauso wichtig wäre die Datenbankbindung, z.B. auch das XML Query Data Model. Vieles ist aber noch im Fluß oder die Tools sind nicht verfügbar.

Interessant wäre auch die Mathematical Markup Language (MathML), Scalable Vector Graphics (SVG), die Synchronized Multimedia Integration Language (SMIL) Animation.

Sicherlich wird es hierzu im Herbst einen Schwung gut lesbarer Bücher geben. Mit dem hier erworbenen Basiswissen müßte es dann möglich sein, kritisch die zukünftigen Entwicklungen zu verfolgen.

Speziell wird empfohlen, die Hinweise im Web unter `www.xml.org` und `www.xml.com` zu verfolgen. Aus dieser Quelle stammt auch der unter `http://www.sun.com/xml/developers/xsl/` verfügbare Satz von 110 Folien zum Thema XSL, die von Paul Grosso (Arbortext) und Norman Walsh (Sun Microsystems) auf der XML Europe Konferenz am 12. Juni 2000 in Paris vorgetragen wurden. Die Besprechung dieser Folien soll hier als Ergänzung und Wiederholung des Stoffs dienen. Der Folientext wird hier nicht gesondert abgedruckt.

ENDE

Inhaltsverzeichnis

Literatur	2
Vorwort	3
Kap. 1: Der XML Standard	4
1.1 Motivation	4
1.2 Ein erstes Beispiel	6
1.3 Eine erste DTD	8
1.4 Ein erstes XSL Dokument	10
Kap. 2: XML Dokumente	13
2.1 Kleinigkeiten	13
2.2 Namensräume	15
2.3 XML Instruktionen	20
2.4 Vorschriften für Elemente und Attribute	23
2.5 Reservierte Attribute in XML	24
2.6 Entity-Referenzen	28
Kap. 3: Document Type Definitions	30
3.1 Ein Datenbankbeispiel	30
3.2 Die DTD Elementdeklarationen	33
3.3 Entities	36
3.4 Attributdeklarationen	41
3.5 Validierende Parser	43
Kap. 4: XML Schema	46
4.1 Warum DTD <i>und</i> XML Schema?	46
4.2 Ein XML Dokument mit einem XML Schema	47
4.3 Ein Beispiel aus dem Normvorschlag	54

Kap. 5: XSL Transformationen	59
5.1 XML Dokumente als Bäume	59
5.2 Strategien zum Einsatz von Transformationen	66
5.3 Templates und Muster	67
Kap. 6: Weitergehende XSL Konzepte	79
6.1 XSL Pattern Matching durch Pfadangabe.....	79
6.2 XSL Filter und Filtermuster	80
6.3 Eingebaute Methoden	84
6.4 Weitere XSL Elemente	86
6.5 Variablen, Konstanten und benannte Templates	96
Kap. 7: Ausblick	97