Informationsstrukturierung

Marcus Kracht
Universität Bielefeld

29. Januar 2018

Dieses Manuskript behandelt den Inhalt der Veranstaltungen “Informationsstrukturierung” und “Informationsstrukturierung 2”. Behandelt werden: XML Schema, XSLT und XPath, FO und XQuery.

Kapitel 1

XML Schema und XSLT

1.1 Wiederholung

1.1.1 Was ist XML?

XML (Extensible Markup Language) ist ein allgemeines Format, wie man Markup in eine Datei einfügt. Die Grundstruktur ist wie folgt:

```
<?xml version="1.0"?>
<katalog>
    <buch isbn="1230456">
    </buch>
    <buch isbn="1230457">
    </buch>
</katalog>
```
(1.1)

Dazu folgende Anmerkungen.

1. Die erste Zeile ist immer eine Prozessinstruktion. Prozessinstruktionen erkennt man an der Folge <?$\cdots$?> Diese Instruktion sagt, dass der Text in XML Version 1.0 abgefasst ist. Es versteht sich von selbst, dass die Buchstabenfolge "xml" obligatorisch ist. Die Versionsnummer ist aber veränderlich. Inzwischen gibt es sogar eine Version 1.1. Jedoch sind die Änderungen für uns vernachlässigbar, weswegen wir mit 1.0 arbeiten.

2. Eine Datei wird eingefasst in ein frei wählbares Wurzeltag. (Es darf nur ein einziges Wurzeltag geben und dies darf auch nur einmal auftreten, ansonsten ist die Datei fehlerhaft.) In diesem Fall ist es "katalog".

Die Struktur von Tags ist wie folgt. Ein öffnendes Tag hat diese Form:

```
<buch isbn="1230457" id="hpq">                    (1.2)
```

Hierbei ist `buch` der Name des Tags. Ein Leerzeichen trennt den Namen von der Zusatzinformation. Diese besteht aus einer (beliebig langen) Reihe von Paaren `{Attribut}={Wert}`, wo `{Attribut}` eine Zeichenkette ist, das sogenannte Attribut, und `{Wert}` ebenfalls eine Zeichenkette, der Wert dieses Attributs.

> [Notationskonvention]
> Die geschweiften Klammern bezeichnen einen beliebigen Ausdruck. Deswegen ist `Attribut` eine spezielle Zeichenkette (wobei der Schreibmaschinenschrift angibt, dass die Buchstaben so wie geschrieben in der Datei enthalten sind) während `{Attribut}` für eine beliebige Zeichenkette steht, welche ein Attribut ist (was das im Einzelnen ist, wird jeweils erläutert). Es ist also `{Attribut}` eine Variable für Zeichenketten. Manchmal werden wir den Typ einer Variablen in die geschweiften Klammern setzen, sodass eine Typbezeichnung entfallen kann.

Beachten Sie: der Wert des Attributs muss in Anführungszeichen gesetzt werden. Dabei dürfen Sie wählen, ob Sie einfache oder doppelte Anführungszeichen setzen wollen. Wählen Sie doppelte, so müssen sie innerhalb des Werts doppelte Anführungszeichen vermeiden! Wählen Sie einfache, so müssen Sie dann diese vermeiden. Alles zusammen wird dann in ein Paar von spitzen Klammern gesetzt.

> [Anführungszeichen]
> Der XML Parser setzt sich bei dem ersten Anführungszeichen in einen besonderen Modus: er liest die Buchstaben ab jetzt als Zeichenkette ein, nicht als XML-Kode. Sobald wiederum Anführungszeichen erscheinen, schaltet er diesen Modus aus. Wollen wir also, dass in der Zeichenkette selbst Anführungszeichen enthalten sind (was wir im Folgenden benötigen, etwa in XPath-Ausdrücken), dann müssen wir diese irgendwie schützen. Deswegen die Alternierungsregel: innerhalb einer Zeichenkette der Form `"···"` verwende man einfache Anführungszeichen, innerhalb einer Zeichenkette der Form `'···'` verwende man doppelte Anführungszeichen.

Aus diesen beiden Prinzipien folgt ein wichtiges weiteres Prinzip.

> [Zeichenketten]
> Unter gewissen Umständen muss eine Zeichenkette bereits in Anführungszeichen gesetzt werden, zum Beispiel, weil sie der Wert eines Attributs ist. Damit der Parser ihr den Wert einer Zeichenkette zuweist, muss sie deshalb noch einmal in Anführungszeichen gesetzt werden.

Als Beispiel sei eine Variable erwähnt. Damit die Variable `s` den Wert `o12` vom Typ einer Zeichenkette bekommt, muss man Folgendes schreiben.

```
<xsl:variable name="s" select="'o12'"/>          (1.3)
```

Hier ist `select` ein Attribut, dessen Wert obligatorisch in Anfühungszeichen gesetzt wird. Diese muss man abziehen und erhält `'o12'`. Die Anfühungszeichen weisen dies nun als Zeichenkette aus. Dies ist deswegen wichtig, weil gewisse Zeichenketten ansonsten missverstanden werden. Eine Telefonnummer wie `012` ist keine Zahl, aber die Sequenz `select="012"` würde genau das aus ihr machen, weil dies die Notationskonvention für Typen so vorsieht.

Ein schließendes Tag besteht aus der Folge

```
</buch>          (1.4)
```

Dabei entfallen sämtliche Attribut-Wert Paare, und die spitze Klammer wird unmittelbar von einem Schrägstrich gefolgt. Es besteht die Möglichkeit, ein Tag, das sofort schließt, wie folgt zu vereinfachen. Anstelle von

```
<buch isbn="1230457" id="hpq">
</buch>          (1.5)
```

schreibt man

```
<buch isbn="1230457" id="hpq"/>          (1.6)
```

Der Schrägstrich schließt sich also an die Sequenz von Attribut-Wert-Paaren unmittelbar an.

Die Behandlung von sogenanntem *whitespace*, also Leerzeichen allgemeiner Art (einfaches Leerzeichen, Tabulatoren, Zeilenunmbruch und Eingabe (Return-Taste)) verdient besondere Beachtung. Eine erschöpfende Erklärung fand ich in dem Dokument von Using XML. Dort findet man auch die Links zu den W3C-Standards.

[Leerzeichen]
Es gibt Leerzeichen, die nur für das Auge gesetzt werden (Einrückungen). Sie existieren in der Datei, werden aber vom Parser an vielen Stellen ignoriert. Tagnamen, Namen von Attributen und vordefinierten Elementen dürfen niemals Leerzeichen enthalten. Werte von Attributen werden normalisiert, das heißt, alle Folgen von Weißzeichen werden zu einem zusammengezogen und diese werden vor und hinter der Zeichenkette gelöscht. (Dieser Effekt wird von der Funktion `normalize-space()` erzeugt.) Werte von Elementen werden nicht immer normlisiert (das ist abhängig von der Implementierung).

Es ist erlaubt (aber irrelevant):

- ein Leerzeichen zwischen < und dem Tagnamen sowie dem Tagnamen und > bzw. />;
- ein Leerzeichen zwischen Attributnamen und dem Gleichheitszeichen, dem Gleichheitszeichen und dem Wert;
- ein Leerzeichen zwischen dem Tag und dem folgenden Text bzw. zwischen dem Text und dem schließenden Tag.

Wir schon angedeutet lässt sich das Verhalten in Bezug auf Text steuern. Ich gehe darauf im Einzelnen noch ein.

1.1.2 Was ist XML Schema?

Normalerweise ist die Syntax von Tags bereits definiert; dafür sorgt schon XML. Aber XML hat über die Beschaffenheit benutzerdefinierter Tags keine Ahnung. Deswegen kann deren Verwendungsweise eigens festgelegt werden. Dazu dient XML Schema. Dies ist eine Sprache für Dokumentgrammatiken. XML Schema ist selbst auch in XML abgefasst; allerdings sind die dort verwendeten Tags jetzt in ihrer Wirkungsweise festgelegt. Damit ein XML-Dokument auf seine Gültigkeit im Sinne des festgelegten Schemas geprüft werden kann, muss man einen *Validierer* haben. Dieser muss die Syntax von XML Schema kennen und richtig interpretieren können.

Wie aber legt man fest, welches Schema für ein XML Dokument benutzt werden soll? Dazu kann man wiederum in die Datei eine Prozessinstruktion einsetzen.

1.1.3 Editoren

Die Universität hat eine Lizenz auf Oxygen, deswegen können Sie es innnerhalb der Universität frei benutzen. Oxygen ist ein Werkzeug zum Erstellen, Validieren und Transformieren von XML-Dateien. Dieses Dokument bietet *keine* Einführung in die Benutzung von Oxygen. Dies kann man ohnehin leicht erlernen. Frei erhältlich sind zum Beispiel die Personal Edition von xmlmind. Xmlmind verkauft eine höherwertige Version als Professional Edition, aber die freie Software (Version 1.1, vom 28. März 2007) ist gut genug. Ebenso gibt es für alle Plattformen den kostenlosen Editor bluefish, sowie für Windows und für die gängigen Linux-Distribution auch Pakete für den XML Copy Editor (Version 1.2.1.3 vom 6.9.2014), der inzwischen sowohl XML Schema validieren kann wie auch XSLT und XPath beinhaltet. Dies dürfte für diesen Kurs ausreichen.

Ich weise darauf hin, dass die Bearbeitung von Dokumenten (das bedeutet, das Edieren) mit jedem der Editoren erfolgen kann, auch gemischt, sodass Sie im Kurs für Ihre Dateien Oxygen verwenden können und zu Hause zB XML Copy Editor. Es ist *auch* möglich, die Dateien mit normalen Editoren (Emacs, Gvim) zu erstellen und zu bearbeiten. Wichtig ist, dass die Editoren *nichts* in die Datei hineinschreiben, was Sie nicht angeordnet haben; das ist zum Beispiel bei Formaten wie RTF, DOCX der Fall.

1.1.4 Quellen

Die offiziellen Dokumente zu den verschiedenen Sprachen sind in den Veröffentlichungen des WWW-Konsortiums zu finden. Insbesondere stehen dort die sogenannten technischen Empfehlungen (Technical Recommendations). Die Anzahl dieser Dokumente steigt rasant an. Zu XML findet man bereits mehrere verschiedene, die ich im Einzelnen nicht besprechen werde. Ich verlinke hier nur die Seite für XML 1.1.

1.2 Graphen

Ein XML Dokument ist zunächst einmal nichts als eine Datei. Diese Datei bezeichnet allerdings als XML Datei eine Struktur, welche der Computer mittels eines Parsers erzeugt. Die abstrakte Grundlage dieser Struktur, welche auch Document Object Model heißt, bespreche ich in diesem Kapitel.

Ein *gerichteter Graph* ist ein Paar (E, K), wo E eine Menge ist, die Menge der

Ecken und K eine Teilmenge von E^2, das heißt, K ist eine Relation auf E. Hier sind ein paar Beispiele.

1. Die Menge E der Zahlen kleiner als 10 und $(i, j) \in K$ gdw. $i < j$.

2. Die Menge E der U-Bahnhöfe in Bielefeld, und $(b, c) \in K$ gdw. b auf einer Linie mit c benachbart ist.

3. Die Menge E der Buchstaben, und $(x, y) \in K$ gdw. y unmittelbar auf x folgt.

Ein *Wald* ist ein Graph, für den Folgendes gilt.

1. Aus $x\,K\,z$ und $y\,K\,z$ folgt, dass $x = y$. (K ist, wie man sagt, *aufwärts eindeutig*. Aber es ist nicht abwärts eindeutig. Denn es darf $x\,K\,y$ und $x\,K\,z$ sein, auch wenn $z \neq y$!)

2. K ist zykelfrei, dh es gibt keine Folge $x_0\,K\,x_1\,K\,x_2 \cdots K\,x_n$, mit $n > 0$, wo $x_0 = x_n$.

Wir können Zykelfreiheit auch wie folgt definieren. Es bezeichne K^+ die *transitive Hülle* von K. Dies ist die kleinste transitive Relation, welche K enthält. Dann ist K genau dann zykelfrei, wenn für kein x: $x\,K^+\,x$. Zur Wiederholung: eine Relation R heißt *transitiv*, wenn aus $x\,R\,y$ und $y\,R\,z$ immer folgt, dass $x\,R\,y$.

Ein Wald ist ein *Baum*, falls es ein r gibt mit $r\,K^+\,x$ für alle x verschieden von r.

1. y heißt *Tochter* von x, wenn $x\,K\,y$.

2. y heißt *Mutter* von x, wenn $y\,K\,x$.

3. y heißt *Nachfahre* von x, wenn $x\,K^+\,y$.

4. y heißt *Vorfahre* von x, wenn $y\,K^+\,x$.

Es sei F eine Menge. Wir nennen eine *Kantenfärbung mit Farben aus F* eine Funktion $f : K \to F$. $x \in K$ hat die *Farbe w*, falls $f(x) = w$. Entsprechend nennen wir eine *Eckenfärbung* (*mit Farben aus F*) eine Funktion $f : E \to F$.

1.2.1 Document Object Model (DOM) I

Einem XML-Dokument ordnen wir eine Struktur zu, das sogenannte DOM (*Document Object Model*). Diese ist in erster Näherung ein gerichteter gefärbter Graph. (Erste Näherung heißt, wie sich noch herausstellen wird, dass die tatsächliche Struktur zwar komplizierter ist, aber auf einem gerichteten gefärbten Graphen aufbaut.) Sei folgendes Dokument gegeben.

```
Eine Bibliothek
<bibliothek>
    <buch id="lm">
        <autor>
            <vorname>
            Victor
            </vorname>
            <nachname>
            Hugo
            </nachname>
        </autor>
        <titel>
            Les Misérables
        </titel>
    </buch>                                    (1.7)
    <buch id="mb">
        <autor>
            <vorname>
            Gustave
            </vorname>
            <nachname>
            Flaubert
            </nachname>
        </autor>
        <titel>
            Madame Bovary
        </titel>
    </buch>
</bibliothek>
```

Dieses Dokument wird auf einen gefärbten gerichteten Graphen abgebildet.

Dies geschieht wie folgt. Wir beginnen mit einem Knoten, nennen wir ihn o1. Dies ist die *Dokumentwurzel*.

1. $o1$ hat genau eine Tochter, nennen wir sie $o2$. Ferner ist $(o1, o2)$ eine Kante mit Farbe `bibliothek`.

2. $o2$ hat genau zwei Töchter, nennen wir sie $o3$ und $o4$. Ferner sind $(o2, o3)$ und $(o2, o4)$ Kanten mit Farbe `buch`.

3. $o3$ hat zwei Töchter, $o5$ und $o6$, und $(o3, o5)$ ist eine Kante mit Farbe `autor`, $(o3, o6)$ eine Kante mit Farbe `titel`, usf.

Einige der Knoten bekommen im Dokument sogar Namen zugewiesen. So heißt $o3$ denn auch `lm`. Dies nennt man einen *Objektidentifizierer*, weil der Name eindeutig sein muss.

Die Textbausteine, die wir eingefügt haben, müssen natürlich auch noch untergebracht werden. Ich nehme hier eine Vereinfachung vor: nur solche Knoten tragen Text, die keine Töchter haben (sogenannte *Blätter* des Baumes). Hier ist nun das **DOM I**.

Hinweise.

1. Die Namen $o1$, $o2$, usf. sind beliebig und können frei gewählt werden! Die XML Datei gibt nichts über diese Namen bekannt. Wir hätten natürlich auch

Tabelle 1.1: DOM I

$$
\begin{aligned}
E = \; & \{o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12\} \\
K = \; & \{(o1, o2), (o2, o3), (o2, o4), (o3, o5), (o3, o6), (o5, o7), (o5, o8), \\
& (o4, o9), (o4, o10), (o9, o11), (o9, o12)\}
\end{aligned}
$$

Kantenfärbung

Kante	Farbe	Kante	Farbe
$(o1, o2)$	`bibliothek`	$(o2, o3)$	`buch`
$(o2, o4)$	`buch`	$(o3, o5)$	`autor`
$(o3, o6)$	`titel`	$(o5, o7)$	`vorname`
$(o5, o8)$	`nachname`	$(o4, o9)$	`autor`
$(o4, o10)$	`titel`	$(o9, o11)$	`vorname`
$(o9, o12)$	`nachname`		

Eckenfärbung

Ecke	Farbe
$o6$	`Les Misérables`
$o7$	`Victor`
$o8$	`Hugo`
$o10$	`Madame Bovary`
$o11$	`Gustave`
$o12$	`Flaubert`

folgendes Dokument aufschreiben können:

```
<bibliothek id="o2">
    <buch id="o3">
        <autor id="o5>
            <vorname id="o7">
            Victor
            </vorname>
            <nachname id="o8">
            Hugo
            </nachname>
        </autor>
        <titel id="o6">
        Les Misérables
        </titel>
    </buch>
    <buch id="o4">
        <autor id="o9">
            <vorname id="o11">
            Gustave
            </vorname>
            <nachname id="o12">
            Flaubert
            </nachname>
        </autor>
        <titel id="o10">
        Madame Bovary
        </titel>
    </buch>
</bibliothek>
```
(1.8)

2. HTML mischt Text mit Markup. In diesem Fall verfährt man wie folgt: Jedem zusammenhängenden Stück Text wird zusätzlich ein Knoten gegeben, als wenn man ein Tag `text` hätte, das den Text einrahmt: Anstelle von

```
<p>Dies hier ist <i>nur</i> ein Beispiel</p>
```
(1.9)

hätte man dann

```
<p>
    <text>Dies hier ist </text>
    <i>nur</i>                                    (1.10)
    <text> ein Beispiel</text>
</p>
```

3. Eine letzte Bemerkung zu Attributen. Jedes Attribut Und jeder Namensraum) bekommt technisch gesehen einen eigenen Knoten. Dies führt meist nicht zu Problemen, aber bei der Dokumentordnung wird dieser Punkt noch relevant werden.

1.2.2 Kontextfreie Grammatiken

In der Linguistik stellt man die obere Struktur gerne wie folgt dar:

$$\begin{aligned}&[_{\texttt{bibliothek}}[_{\texttt{buch}}[_{\texttt{autor}}[_{\texttt{vorname}}\,\texttt{Victor}][_{\texttt{nachname}}\,\texttt{Hugo}]]\\&\quad[_{\texttt{titel}}\,\texttt{Les Misérables}]][_{\texttt{buch}}[_{\texttt{autor}}[_{\texttt{vorname}}\,\texttt{Gustave}]\\&\quad[_{\texttt{nachname}}\,\texttt{Flaubert}]][_{\texttt{titel}}\,\texttt{Madame Bovary}]]]\end{aligned} \tag{1.11}$$

Diese wird nämlich auch gerne als Baum dargestellt. Hierbei sind die Farben aber den Knoten zugeordnet. Technisch gesehen ist dies bei den XML Dokumenten denn auch genau so der Fall. Die Tags sind hier als Knotenfarben angesehen, nicht als Kantenfarben. Warum ist das möglich?

Hinweis. In der Klammerstruktur habe ich nur die Farbe der öffnenden Klammer angeben. Die Farbe der schließenden ergibt sich automatisch (wie sieht man das?). Trotzdem hat man bei XML nicht darauf verzichtet, die Tagnamen zu doppeln. Das hat einzig den Grund, dass der Nutzer bei einem schließenden Tag noch einmal daran erinnert wird, wo er sich in der Struktur befindet. Es entfällt das mühselige Klammerzählen!

In einem Baum hat ein Knoten y genau eine einlaufende Kante. Deswegen besteht eine eineindeutige Abbildung f zwischen den Knoten ungleich der Wurzel und den Kanten, die wie folgt bestimmt wird: $f(y)$ ist dasjenige x, für das $(x, y) \in K$. (Mengentheoretisch gesehen *ist* f nichts anderes als K.)

Daraus folgt: jede Kantenfärbung kann in eine Eckenfärbung transformiert werden. Jede Eckenfärbung kann in eine Kantenfärbung transformiert werden, wobei die Farbe der Wurzel aber verlorengeht. (Deswegen wird in dem DOM immer noch eine Wurzel neu hinzugefügt!)

Tabelle 1.2: DOM II: (E,K,L,t,f)

$$
\begin{aligned}
E = \;& \{o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12\} \\
K = \;& \{(o1, o2), (o2, o3), (o2, o4), (o3, o5), (o3, o6), (o5, o7), (o5, o8), \\
& \quad (o4, o9), (o4, o10), (o9, o11), (o9, o12)\} \\
L = \;& \{(o3, o4), (o5, o6), (o7, o8), (o9, o10), (o11, o12)\}
\end{aligned}
$$

Farben

Kante	Farbe	Kante	Farbe
$(o1, o2)$	`bibliothek`	$(o2, o3)$	`buch`
$(o2, o4)$	`buch`	$(o3, o5)$	`autor`
$(o3, o6)$	`titel`	$(o5, o7)$	`vorname`
$(o5, o8)$	`nachname`	$(o4, o9)$	`autor`
$(o4, o10)$	`titel`	$(o9, o11)$	`vorname`
$(o9, o12)$	`nachname`		

Eckenfärbung

Ecke	Farbe
$o6$	`Les Misérables`
$o7$	`Victor`
$o8$	`Hugo`
$o10$	`Madame Bovary`
$o11$	`Gustave`
$o12$	`Flaubert`

1.2.3 Document Object Model (DOM) II

Zusätzlich zu der hierarchischen Ordnung auf den Knoten gibt es auch eine lineare. Ist $x\,K\,y$ und $x\,K\,z$ mit $y \neq z$, so liegt y in der Datei entweder vor oder hinter z. Zum Beispiel ist $(o2, o3) \in K$ und $(o2, o4) \in K$. Und es steht $o3$ vor $o4$. Diese Ordnung unter Schwesterknoten wird auch mitaufgenommen.

Hinweis. Attribute sind von der Form `{Attribut}={Wert}`, wobei der Wert eine Zeichenkette ist. Insofern könnten wir sie durch die Folge

```
<{Attribut}>Wert</{Attribut}>
```
(1.12)

ersetzen. Es gibt aber einen Unterschied: Attribute sind nicht geordnet! Das Tag

```
<autor vorname="Victor" nachname="Hugo"/>
```
(1.13)

mag für uns identisch sein mit

```
<autor>
    <vorname>
        Victor
    </vorname>
    <nachname>
        Hugo
    </nachname>
</autor>
```
(1.14)

Aber das ist nicht richtig. Denn wir haben in der Struktur jetzt drei Knoten, und die sind geordnet. Deswegen ist

```
<autor vorname="Victor" nachname="Hugo"/>
```
(1.15)

nach dem Einlesen *identisch* (dh erzeugt ein identisches DOM) wie

```
<autor nachname="Hugo" vorname="Victor"/>
```
(1.16)

Nicht identisch sind aber:

```
<autor>          <autor>
    <vorname>        <nachname>
        Victor           Hugo
    </vorname>       </nachname>
    <nachname>       <vorname>
        Hugo         Victor
    </nachname>      </vorname>
</autor>         </autor>
```
(1.17)

Formal fehlt uns aber jetzt noch ein Gegenstück zu den Attributen. Wir müssen also noch eine weitere Kantenfärbung einführen, die als Werte eine Zuordnung von Attributen zu Werten hat (die also eine Funktion $f : A^* \to A^*$ ist, mit A dem Alphabet).

Es gibt insgesamt 7 Typen von Knoten:

1. Der **Dokumentknoten**. Ist die Wurzel des Baumes und *kein* Element!
2. **Elementknoten**. Dies sind Knoten, die über ein Paar von Tags erzeugt werden.

3. **Textknoten**. Enthalten maximale Segmente von Text (technisch: **PCDATA**).

4. **Attributknoten**. Enthält den Namen und den Wert eines einzelnen Attributs.

5. **Kommentarknoten.** Werden durch die Kommentarklammern definiert: `<!--` und `-->`.

6. **Prozessinstruktionsknoten**. Eine Prozessinstruktion hat die Form <?(Name) (Inhalt)?>. Der Wert der Instruktion ist alles, was nach dem Namen folgt. Die erste Zeile (also `<?xml version="1.0"?>`) ist *keine* Prozessinstruktion und definiert deswegen keinen Knoten.

7. **Namensraumknoten**.

Die Dokumentordnung besteht zwischen Elementen, Namensraumknoten und Attributknoten. Sie wird technisch wie folgt bestimmt.

1. Die Dokumentwurzel ist das erste Element.

2. Jeder Knoten ist vor einen Unterknoten sowie deren Unterknoten.

3. Namensraumknoten kommen nach dem Knoten, zu dem sie assoziiert sind. Ihre relative Ordnung ist stabil aber Implementuerungsabhängig.

4. Attributknoten folgen den Namensraumknoten des Knotens, zu dem sie assoziiert sind. Ihre relative Ordnung ist stabil aber Implementierungsabhängig.

5. Unterknoten eines gegebenen Knotens sind so angeordnet wir im Dokument selbst.

6. Unterknoten eines Knotens stehen vor jedem Schwesterknoten, der diesem nachfolgt.

1.2.4 Document Object Model (DOM) III

Zu guter Letzt wollen wir noch auf die Indizes zurückkommen. Durch Vergabe von Marken (*Identifizierern*) schaffen wir schließlich noch eine weitere Struktur. Dies ist eine weitere Relation, nennen wir sie R. Dabei soll $x\,R\,y$ sein, wenn y für das Attribut `idref` denselben Wert gibt wie x für das Attribut `id`.

1.3 XML Schema

1.3.1 Strukturbeschreibung

In diesem Abschnitt will ich einen Überblick über XML Schema geben. XML Schema ist eine Sprache, um Dokumenttypen zu beschreiben, dh festzulegen, welche Dokumente wohlgeformt sind. Die Definitionen sind in den Dokumenten W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures sowie W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes festgelegt. Die Syntax von XML Schema ist wiederum die von XML; der Namensraum ist `xs`, was bedeutet, dass alle XML Schema Tags dadurch gekennzeichnet sind, dass sie mit `xs:` beginnen. Ein XML Schema Dokument beginnt mit den folgenden Zeilen, dem magischen Spruch, den ich im Folgenden allerdings stets weglassen werde:

```
<?xml version="1.0"?>
<xs:schema                                                    (1.18)
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Das abschließende `</xs:schema>` lasse ich natürlich ebenfalls weg, darf aber in der Schema-Datei nicht fehlen.

Falls wir unsere Bibliothek in eine kontextfreie Grammatik fassen wollten, so sähe diese ungefähr wie folgt aus.

```
<bibliothek>→<buch> <bibliothek> | <buch>
<buch>      →<autor> <titel>                                  (1.19)
<autor>     →<vorname> <nachname>
```

Dies beschreibt die nichtterminalen Regeln. Die terminalen Regeln sind hier unwesentlich, weil zB im Augenblick nicht geregelt ist, was man für Vornamen setzen darf und was nicht. (Dies kann man natürlich ebenfalls tun.)

In XML Schema wird diese Grammatik nun in eine besondere Form gegossen. Dabei werden nicht exakt die oben angegebenen Regeln verwendet. Die Grammatik, die de facto implementiert ist, hat folgende Form.

```
<bibliothek>→<buch>+
<buch>      →<autor> <titel>                                  (1.20)
<autor>     →<vorname> <nachname>
```

Hierbei bedeutet das hochgestellte Pluszeichen, dass das Element beliebig oft wiederholt werden darf (mindestens jedoch einmal). Denn in dem DOM, welche ja

die Struktur für unsere Grammatik gibt, hat ein Bibliotheksknoten beliebig viele Töchter. Zur Erinnerung: ein DOM bestand aus einer Eckenmenge E, mit einer Relation K (Dominanz) und L (Präzedenz zwischen Töchtern) und einer (de facto) Eckenfärbung f. K und L definieren zusammen auf E einen geordneten Baum und f assoziiert mit jeder Ecke eine Farbe (das Nichtterminalsymbol).

Ich verweile noch kurz bei dem Unterschied zwischen diesen Grammatiken, bevor wir zu XML Schema übergehen. Nehmen wir an, wir haben 3 Bücher. In der Grammatik (1.20) können wir dies wie folgt ableiten:

$$\texttt{<bibliothek>} \rightarrow \texttt{<buch> <buch> <buch>} \tag{1.21}$$

Dies ist so, weil das Pluszeichen eigentlich eine Abkürzung für unendliche viele Regeln ist, deren eine (1.21) ist. Nicht so in (1.22):

$$\begin{aligned} \texttt{<bibliothek>} &\rightarrow \texttt{<buch> <bibliothek>} \\ &\rightarrow \texttt{<buch> <buch> <bibliothek>} \\ &\rightarrow \texttt{<buch> <buch> <buch>} \end{aligned} \tag{1.22}$$

Dies ist aber keine zulässige XML Struktur, weil der Knoten namens `<bibliothek>` eingebettet vorkommt.

Doch nun zurück zu XML Schema. XML Schema erlaubt, die erste Regel von (1.20) wie folgt zu fassen.

```
<xs:element name="bibliothek">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="buch"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```
(1.23)

Schaut man sich die Struktur etwas genauer an, so besteht sie aus einer Definition, die sagt, dass hier der Typ `bibliothek` (nicht das Tag!) definiert wird, und einer Spezifikation, wie dieser Typ auszusehen hat. Dabei muss man angeben, ob es sich um einen einfachen Typ oder um einen komplexen Typ handelt (`xs:simpleType` vs. `xs:complexType`). In diesem Fall ist es ein komplexer Typ, weil es in dem Dokument noch untergeordnete Tags gibt, denen jeweils eigene Typen entsprechen. Ich komme auf den Unterschied noch zu sprechen. Die untergeordneten

Tags müssen den Typ buch haben. Angegeben ist ebenfalls, wie viele Unterknoten vom Typ buch es geben darf. In diesem Fall ist die Anzahl unbegrenzt. Hier nun ist die Spezifikation der zweiten Regel.

```
<xs:element name="buch">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="autor" maxOccurs="1"/>
            <xs:element ref="titel" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```
(1.24)

Knoten, die Töchter haben oder Attribute, sind stets komplex. Ein komplexer Typ muss deswegen weiter spezifiziert werden. Unter anderem gilt, es Folgendes zu spezifizieren:

- Welche Töchter sind zugelassen und wie oft?
- Welche Attribute sind zugelassen?

Ich habe bei bibliothek angegeben, dass buch beliebig oft auftreten darf, aber nichts über Mindestwerte gesagt. Die Defaultwerte sind 1 sowohl für minOccurs wie auch für maxOccurs, deswegen hätte ich das Attribut maxOccurs für autor und titel, nicht aber für buch weglassen können. Bei beiden habe ich hingegen minOccurs weggelassen, sodass eine Bibliothek mindestens ein Buch enthalten muss (wie in der Regeln oben verlangt). Die Spezifikation ist also treu für die oben gegebenen Regeln.

Alle weiteren Regeln werden genauso kodiert. Eine terminale Regel besitzt die folgende Form.

```
<xs:element name="titel"/>
```
(1.25)

Optional kann man auch type="xs:string" einfügen; dies legt fest, dass das Element eine Zeichenkette beherbergt; doch davon später. Dies erledigt noch nicht ganz die kontextfreie Grammatik, denn wir können ja ein und dasselbe Symbol auf verschiedene Weisen entwickeln, dh wir können Regeln der Form A $\rightarrow$ BC|CD haben. Für diesen Fall gibt es das Konstrukt <xs:choice>. Betrachten wir die

Möglichkeit, dass ein Buch einen Autor oder einen Herausgeber haben kann. Dies können wir so kodieren.

```
<xs:element name="buch">
    <xs:complexType>
        <xs:choice>
            <xs:sequence>
                <xs:element ref="autor"/>
                <xs:element ref="titel"/>
            </xs:sequence>
            <xs:sequence>
                <xs:element ref="herausgeber"/>
                <xs:element ref="titel"/>
            </xs:sequence>
        </xs:choice>
    </xs:complexType>
</xs:element>
```
(1.26)

Die zugehörige kontextfreie Regel ist wie folgt.

$$\texttt{<buch>} \rightarrow \texttt{<autor><titel>} \mid \texttt{<herausgeber><titel>} \tag{1.27}$$

Es ist nun zusätzlich möglich, die Reihenfolge der Töchter freizugeben (mit Hilfe von `<xs:all>`). Dies geht jedoch zu Lasten der Performanz (das Abprüfen kostet Zeit exponentiell in der Anzahl der Knoten!). Deswegen ist die Empfehlung, damit sparsam umzugehen. Es gab bzw. gibt bei `<xs:all>` auch gewisse Einschränkungen bei den Werten von `maxOccurs`.

1.3.2 Zwei verschiedene Modelle der Definition

Sehen wir und die Definition von zwei Elementen an. Es gibt zunächst die Möglichkeit, diese mit Hilfe des Attributs `name` für ein Element explizit anzugeben. Zweitens aber kann man eine einmal gegebene Definition an anderer Stelle noch einmal verwenden, indem man an der Stelle von `name` nunmehr das Attribut `ref` wählt. Da die Definition anderswo bereits gegeben ist, darf der Knoten von `xs:element` in diesem Fall keinerlei Unterelemente besitzen.

Sehen wir uns das genauer an.

```
<xs:element name="konto">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="inhaber"/>
            <xs:element ref="nummer"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="inhaber">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="name"/>
            <xs:element ref="addresse"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```
(1.28)

Ich habe hier die Definition von `inhaber` referenziert und explizit angegeben. Eine Definition von `nummer` müsste ebenfalls noch gegeben werden. Es ist dabei Folgendes zu beachten: Falls eine Definition referenziert wird, so wird die Definition für das Element unterhalb das Knotens `xs:schema` gesucht. Sollten Sie also die Definition tiefer eingebettet haben, so findet das Programm diese nicht. Die Idee dahinter ist wie folgt. Eine Definition eines Elements kann kontextabhängig sein. Es ist zulässig, an verschiedenen Stellen verschiedene Definitionen eines Elements zu geben. Gibt es zwei Definitionen für Elemente desselben Namens, so müssen sie allerdings einen verschiedenen Kontext besitzen. So kann deswegen auch unterhalb von `xs:schema` nur *eine* Definition des Elements `inhaber` gegeben werden, alles andere wäre ein Fehler. Diese (und nur diese) Definition wird aufgesucht, wenn Sie irgendwo `ref="inhaber"` verwenden. Es ist zulässig, stattdessen `name="inhaber"` zu verwenden, aber dann müssen Sie die Definition am Ort angeben. Dies sieht so aus.

```
<xs:element name="konto">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="inhaber">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="name"/>
                        <xs:element ref="addresse"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element ref="nummer"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```
(1.29)

Diese unterscheiden sich wie folgt. In der ersten wird das Element `konto` dadurch definiert, dass auf die Definitionen für `inhaber` und `nummer` lediglich verwiesen wird. Diese müssen selbstverständlich im Dokument vorhanden sein. In der zweiten Variante wird auf keine Definition von `inhaber` verwiesen, sondern das Element wird an Ort und Stelle definiert. Deswegen wird auch nicht das Attribut `ref` verwendet sondern `name`. Denn `ref` *verweist* auf eine im Dokument gegebene Definition (die direkt unter dem Element `xs:schema` erfolgen muss), während `name` angibt, dass hier ein Element definiert wird.

Es ist möglich, an verschiedenen Stellen eine Definition von `inhaber` abzugeben, und diese müssen nicht gleich sein. Nur wenn der Kontext derselbe ist (zum Beispiel als Tochterelement von `xs:schema`) muss die Definition eindeutig sein, was in der Praxis bedeutet, dass dort nur eine Definition gegeben werden darf.

Keine der Varianten ist an und für sich gut oder schlecht. Es kommt auf die Anwendung an. Der Vorteil der ersten Variant ist, dass eine Definition für einen Elementtyp nur einmal gegeben werden muss; dieser kann dann beliebig oft verwendet werden. Der Vorteil der zweiten Variante ist, dass sie kontextabhängige Abweichungen zulässt und ein wenig ökonomischer ist. Sie bildet die XML Struktur direkt ab. Es ist möglich, gewisse Elemente getrennt zu definieren, andere wiederum kontextabhängig.

1.3.3 Attribute und Werte

Unter einem Elementknoten lassen sich Attribute vereinbaren. Im einfachsten Fall haben wir nur

```
<xs:attribute name="isbn" type="xs:string"/>          (1.30)
```

Diese Vereinbarung muss innerhalb von `xs:complexType` erfolgen. Sie erlaubt zunächst, das Attribut als legal zu vereinbaren, und gleichzeitig zu sagen, dass es als Wert eine Zeichenkette haben muss. (ISBN Nummern sind keine Zahlen, weil die Behandlung von Nullen anders ist; siehe dieses Wiki zur ISBN).

Wichtig ist der Typ `xs:ID`, welcher das Attribut zu einem Identifikator macht, sodass der Parser die Arbeit übernimmt, diese auf Eindeutigkeit zu prüfen. *Achtung:* In diesem Fall hat der Wertstring den Typ NCName (noncolonised name = Zeichenkette ohne Semikolon). Außerdem darf eine ID auch keine Leerzeichen enthalten oder mit einer Zahl beginnen.

> [Syntax]
> Stellen Sie Attributvereinbarungen stets *hinter* die Elementvereinbarungen, sonst akzeptiert Oxygen dies nicht.

Der Wert eines Attributs kann ein einzelner Typ sein oder eine Liste. Dies wird durch einen nachgestellten Stern gekennzeichnet (also `xs:int*`, `xs:string*`, und so weiter). Ist es eine Liste, so wird das Leerzeichen als Trennsymbol gelesen. Dies führt im Falle von Zeichenketten zu folgendem Problem: Eine Zeichenkette, die ein Leerzeichen enthält, wird als mehrteilige Liste von Zeichenketten aufgefasst. Aus diesem Grund müssen Zeichenketten in Anführungszeichen gesetzt werden. Es ist dann `'eine Maus'` verschieden von `'eine' 'Maus'`. Das erste ist eine einzige Zeichenkette, das zweite ein Paar von Zeichenketten (die ihrerseits kein Leerzeichen enthalten).

1.3.4 Terminale Ketten

Terminale Ketten können auf viele Weisen beschränkt werden.

1. Wir legen den Typ fest (`xs:string`, `xs:date`, `xs:int`).
2. Wir zählen die möglichen Werte auf.
3. Wir beschreiben die möglichen Werte durch einen regulären Ausdruck.

Illustration von (2).

```
<xs:simpleType name="klasse">
    <xs:restriction base="xs:string">
        <xs:enumeration value="a"/>
        <xs:enumeration value="b"/>
        <xs:enumeration value="c"/>
    </xs:restriction>
</xs:simpleType>
```
(1.31)

Illustration von (3).

```
<xs:simpleType name="isbn">
    <xs:restriction base="xs:string">
        <xs:pattern value="\d{9}[0-9X]"/>
    </xs:restriction>
</xs:simpleType>
```
(1.32)

Die Spezifikation von regulären Ausdrücken folgt im Wesentlichen der Standardsyntax. Die Symbole '?', '*', '+', '.', '-', '^' und die Klammern sind technische Symbole zum Aufbau des Ausdrucks; sie müssen maskiert werden, wenn man sie als Symbol wirklich haben will. Es bedeutet '?' (Postfix), dass der voranstehende Term optional ist, es bezeichnet '\?' dagegen das Fragezeichen selbst.

1.3.5 Typen

Wie schon erwähnt, hat ein Objekt stets auch einen *Typ*. Typen sind eigentlich abstrakte Struktureigenschaften, die die Objekte eines Typs teilen. Mengen haben die Eigenschaft, aus Elementen zu bestehen. Mengen sind ungeordnet und die Multiplizität spielt keine Rolle. Es ist $\{a, b, a, b\}$ dieselbe Menge wie $\{a, b\}$ oder $\{b, a\}$, aber nicht wie $\{a, c\}$ oder $\{a, b, d\}$. Mengen unterscheiden sich von Listen dadurch, dass in Listen sowohl die Anordnung zählt wie auch die Multiplizität. So sind die folgenden Listen alle verschieden: $[a; b; a; b]$, $[a; b]$, $[b; a]$, $[a; c]$ und $[a; b; d]$.

Der Typ kodiert in der Informatik nicht nur die abstrakten Eigenschaften, sagt uns also nicht nur, was die Objekte wirklich sind, sondern gibt uns auch (und insbesondere) an, wie wir die Objekte verwenden dürfen und welche Objekte untereinander gleich sind. Man kann zB Mengen als Listen abspeichern, und das wird auch oft getan. Nur sind die Funktionen, die auf Mengen als Listen definiert sind,

andere, als diejenigen, die auf Listen normalerweise verwendet werden. Wenn wir zB ein bereits vorhandenes Objekt einer Menge hinzufügen, so ändert sich die Menge nicht, und das soll dann auch unsere als Liste abgespeicherte Menge nicht ändern.

Hier ist eine Liste von wichtigen Grundtypen.

`xs:boolean`	:	boolesche Werte (`true` oder `false`, sowie `1` und `0`)
`xs:int`	:	ganze Zahlen
`xs:string`	:	Zeichenketten
`xs:decimal`	:	Dezimalzahlen
`xs:float`	:	Gleitkommazahlen

Anmerkung. Es gibt auch noch den Typ `xs:integer`, der ein Untertyp von `xs:decimal` ist und aus den ganzen Dezimalzahlen besteht. Der Unterschied zwischen `xs:int` und `xs:integer` dürfte für unsere Zwecke unwesentlich sein. Bitte benutzen Sie `xs:int`.

Aus den Grundtypen lassen sich höhere Typen zusammenstellen. So kann man Listen von Zahlen, Listen von Zeichenketten, und so weiter erstellen. Man unterscheidet zwischen *schwach* und *stark* getypten Sprachen. In einer *stark* getypten Sprache ist der Typ eines Objekts unveränderlich. Es gibt Funktionen, die Objekte eines Typs in Objekte eines anderen Typs umwandeln, aber ein Objekt kann nicht als ein Objekt anderen Typs gebraucht werden. *Schwach* getypte Sprachen sind hingegen toleranter. Sie erlauben es, ein Objekt ohne Warnung als Objekt eines anderen Typs zu gebrauchen. Beliebt ist es, die Zeichenkette `0123` auch als ganze Zahl zu verwenden, nämlich die Zahl 123 (sodass die Null plötzlich unwesentlich ist). Oder eine Zahl als boolesch (0 steht für `false`, alles andere für `true`). Skriptsprachen (Bash, Tcl) sind oft schwach getypt, während sich bei höheren Programmiersprachen das starke Typen langsam durchsetzt. XPath vollzieht gerade diese Wandlung.

> [Kommunikation von Typinformation]
> Letztlich und endlich werden alle Typen als Zeichenketten kommuniziert. Dies erfordert eine Syntax, die die Identität eines Objektes eindeutig festlegt. In XML ist es zusätzlich das Schema, das eine Typzuweisung (und eine Typdefinition) erlaubt.

Die Terminologie vom W3-Konsortium ist wie folgt: Zu einem Typ gehören ein *Werteraum*, welcher die möglichen Werte des Typs enthält, ein *lexikalischer Raum*, welcher die zulässigen Zeichenketten enthält, und ein paar Funktionen und Relationen, welche Folgendes festlegen:

1. welche Zeichenkette welchem Wert zugeordnet ist,
2. welche Zeichenketten gleiche Werte besitzen,
3. in welcher Weise die Zeichenketten angeordnet sind.

Eine Liste der Datentypen und deren Zeichenketten findet sich hier.

Wie schon in XML gesehen, werden alle Werte zunächst als Zeichenketten kommuniziert. Ein Attribut-Wert Paar hat die Form `{Attribut}="{Wert}"`. Dabei ist das Attribut stets eine Zeichenkette und wird *nicht* in Anführungszeichen gesetzt. (Daraus folgt allerdings auch, dass Attribute keine Leerzeichen enthalten dürfen. Technisch gesehen ist deswegen der Typ eines Attributs nicht der einer Zeichenkette.) Der Wert hingegen wird *stets* in Anführungszeichen gesetzt, ist also zunächst einmal eine Zeichenkette. Wann aber versteht XML die Kette `"0"` als Zeichenkette und wann als Zahl? Dazu bedient es sich gewisser Vereinbarungen. Als Wert von `minOccurs` wird `"0"` als Zahl interpretiert, als Wert eines unbekannten Attributs jedoch nicht. Woran liegt das?

> [Nicht eigentliche Zeichenketten]
> Grundsätzlich sind alle in doppelte Anführungszeichen gesetzte Zeichenketten auch intern Zeichenketten. Sollen sie anders interpretiert werden, so muss eine Regel dafür vorliegen. Diese besteht in einer Vereinbarung, die entweder in XML, oder in XML Schema, oder in einem sonst irgendwie festgelegten Schema explizit vermerkt ist.

In XML sieht die Lage wie folgt aus. Jedes Objekt wird als *Paar* abgelegt; das erste Element ist der Typ, das zweite das Objekt selbst. Zwei getypte Objekte sind gleich, wenn sie sowohl den gleichen Typ wie den gleichen Wert haben. Wird ein Schema angelegt und mit dem Dokument assoziiert, dann erstellt XML in der PSVI (*Post Schema Validation Infoset*) eine Liste mit Objekten und Typen und greift darauf zurück. Wird *kein* Schema vereinbart und ist der Typ eines Objektes nicht ersichtlich, so setzt XML einen speziellen Typ ein, nämlich `xs:untypedAtomic`. Das bedeutet, dass damit *jedes* Objekt getypt ist, und sei es nur mit `xs:untypedAtomic`.

1.3.6 Das Typsystem

Das Typsystem von XML ist reich. Dabei spielen viele Typen nur eine Nebenrolle, man sollte allerdings von vielen zumindest gehört haben. Es gibt Basistypen und Verfeinerungen davon.

- *Atomare Haupttypen*:
  - `xs:anyURI` (für eine Webressource)
  - `xs:boolean` (für einen booleschen Wert)
  - `xs:date` (für ein Datum)
  - `xs:dateTime` (für eine Datum und Zeitangabe)
  - `xs:decimal` (für eine Dezimalzahl)
  - `xs:double` (für eine Dezimalzahl in doppelter Präzision)
  - `xs:integer` (für eine ganze Zahl)
  - `xs:QName` (für einen "qualifizierten Namen")
  - `xs:string` (für eine Zeichenkette)
  - `xs:time` (für eine Zeitangabe)
  - `xs:dayTimeDuration` (für ein Zeitinterval)
- *Atomare Nebentypen*
  - `xs:gYear`
  - `xs:gYearMonth`
  - `xs:gMonth`
  - `xs:gMonthDay`
  - `xs:gDay`
  - `xs:duration`
  - `xs:float`
  - `xs:hexBinary`
  - `xs:base64Binary`
  - `xs:NOTATION`
- *Abgeleitete Zahltypen zB*
  - `xs:positiveInteger`
  - `xs:short`
  - `xs:unsignedByte`

- *Abgeleitete Zeichenkettentypen zB*
  - `xs:token`
  - `xs:NCName`
- *Der Allzwecktyp* `xdt:untypedAtomic`.

Die Elemente eines Typs müssen ein bestimmtes Erscheinungsbild haben.

- `xs:decimal` Optional ein Vorzeichen (+ oder -), dann eine Ziffernfolge, ein Dezimalpunkt (!) und eine weitere Ziffernfolge. Eine (und nur eine) dieser Ziffernfolgen darf leer sein.
- `xs:float` Erlaubt zusätzlich auch noch Exponentialschreibweise. Ich gehe daruf nicht näher ein.
- `xs:time` Besteht aus drei Zweierblöcken, getrennt durch Doppelpunkt, wobei der erste die Stunden, der zweite die Minuten, der dritte die Sekunden angibt, zB `13:05:25`. Dazu kann noch ein Zeitzoneninkrement angegeben werden.
- `xs:date` Drei Ziffernfolgen, getrennt durch einen Strich. Die erste gibt das Jahr an, die zweite den Monat, die dritte den Tag, zB `2017-10-31`.Ein Zeitzoneninkrement kann optional angegeben werden.
- `xs:dateTime` Besteht aus einer Datumsangabe (wie in `xs:date`, einem `T`, und einer Zeitangabe (wie in `xs:time`), gefolgt von einem Zeitzoneninkrement, zB `2017-10-31T10:45:00`.

Ich gehe noch gesondert auf die Zeichenkettentypen ein.

- `xs:string` Jede Art Zeichenkette, in der Leerzeichen signifikant sind.
- `xs:normalizedString` Jede Art Zeichenkette, in der verschiedene Arten von Leerzeichen nicht unterschieden werden.
- `xs:token` Eine Folge von Elementen getrennt durch Leerzeichen. (Das Leerzeichen darf dann natürlich nicht Teil der Elementbezeichnungen sein!)
- `xs:language` Eine Bezeichnung von Sprachen und Varianten, zB `de` oder `de-DE` (Sprache plus Bereichskürzel, wichtig für Rechtschreibung und Währungszeichen) usf. Für die Sprachkürzel gilt die Norm ISO 639-1 und für die Länder die Norm ISO 3166-1.

- `xs:NMTOKEN` Eine Folge von Buchstaben, die als *name characters* gelten (dazu gehören Buchstaben, Ziffern, Punkt, Doppelpunkt, Bindestrich, Unterstrich).

- `xs:Name` Ein `NMTOKEN`, das dazu noch mit einem *Beginnzeichen* anfängt, wozu Buchstaben, Unterstrich und Doppelpunkt gehören, aber nicht Zahlen.

- `xs:NCName` Ein `Name`, der keinen Doppelpunkt enthält.

- `xs:ID` Ein `NCName`, Wert von dem Attribut `ID`, der zusätzlich für das gesamte Dokument eindeutig sein muss.

- `xs:IDREF` Ein `NCName`, Wert von dem Attribut `IDREF`, der in dem Dokument auch Wert von `ID` ist.

1.3.7 Typen verwenden in XML Schema

In der Definition von `xs:element` besitzt man zwei Arten, den Typ festzulegen. Die erste sieht so aus:

```
<xs:element name="autor" type="xs:string"/>      (1.33)
```

In diesem Fall greift man auf einen vordefinierten Typ zurück und setzt ihn als Wert des Attributs `type` ein. Das ist die einfachste Art, ein terminales Tag zu definieren, das zum Beispiel nur Text enthalten soll. Ebenso kann man bei Attributen verfahren:

```
<xs:attribute name="telefonnummer" type="xs:string"/>   (1.34)
```

In XML Schema gibt es die Möglichkeit, eigene Typen zu definieren, sowie einem Objekt einen Typ zuzuweisen. Grundsätzlich unterscheiden wir streng zwischen

1. Einfachen Typen: Dies sind Werte von Attributen und einfache Tags, dh Tags, die keine Attribute enthalten *und* unter sich keine weiteren Tags mehr besitzen.

2. Komplexen Typen: alle anderen.

Diese Unterscheidung wird in den Elementen `xs:simpleType` und `xs:complexType` aufgefangen. Es gibt allerdings de facto einen dritten Typ, der unter `xs:complexType` subsumiert wird: falls es keine Untertags gibt, tritt der Fall des sogenannten `xs:simpleContent` ein, auf den ich am Ende dieses Abschnitts eingehen werde.

Achtung. Bei komplexen Typen ist der Default, dass Text und Werte nicht gemischt werden dürfen. Das bedeutet, dass Knoten entweder Attribute haben und Unterknoten, oder aber einen Wert, dh Text, aber nicht beides. Dies kann man ändern. Dazu steht das Attribut `mixed` für `xs:complexType` bereit. Der Default ist `false`, man muss es also nur auf `true` setzen, wenn man sowohl Text wie auch Attribute bzw. Untertags haben will.

Wir haben schon gesehen, wie komplexe Typen definiert werden. Wir haben diese Typen direkt in das `xs:element` eingebaut, es ist aber möglich, sie eigens zu definieren, ihnen einen Namen zu geben und dann zu referenzieren. Das geht dann so. Anstelle von

```
<xs:element name="bibliothek">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="buch"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```
(1.35)

schreibe man

```
<xs:complexType name="btype">
    <xs:sequence>
        <xs:element ref="buch"
            maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="bibliothek" type="btype"/>
```
(1.36)

Dies wird man meistens nicht immer tun wollen, weil es die Definition intransparent macht, kann aber Vorteile haben, zB wenn ein Typ mehrfach gebraucht wird.

Einfache Typen kann man wie folgt definieren:

1. Restriktion. Ein Typ wird als Restriktion eines schon vorhandenen Typs definiert. Dabei darf der Basistyp ein XML-Typ sein wie auch ein im Schema definierter eigener Typ.

```
<xs:simpleType name="meinTyp">
    <xs:restriction base="xs:string">
        <xs:enumeration value="a"/>
        <xs:enumeration value="b"/>
        <xs:enumeration value="c"/>
    </xs:restriction>
</xs:simpleType>
```
(1.37)

Dazu einige Anmerkungen.

- Falls wir keinerlei Restriktionen wollen, also nur den Typ `xs:int` einsetzen wollen, muss man zwar auch `xs:restriction` verwenden, lässt aber die nachfolgenden Elemente weg:

  ```
  <xs:simpleType>
      <xs:restriction base="xs:string"/>
  </xs:simpleType>
  ```
  (1.38)

  Dies mag überflüssig erscheinen, aber es wird später noch gebraucht.

- Eine Alternative zu der Aufzählung ist bei Zahlen die Verwendung von `xs:minInclusive` und `xs:maxInclusive`. (Die Bedeutung ist hoffentlich selbsterklärend; es gibt auch `xs:minExclusive` und `xs:maxExclusive`.) Diese geht wie folgt.

  ```
  <xs:simpleType name="schulnote">
      <xs:restriction base="xs:int">
          <xs:minInclusive value="1">
          <xs:maxInclusive value="6">
      </xs:restriction>
  </xs:simpleType>
  ```
  (1.39)

- Bei Zeichenketten kann man über `xs:whitespace` auch die Kontrolle über die Behandlung von Leerzeichen steuern. Der Wert `preserve` sagt, dass Leerzeichen gleich welcher Art erhalten werden müssen; `replace` sagt,

dass Leerzeichen ersetzt werden sollen (Tab und Newline durch Leerzeichen); `collapse` bedeutet, dass alle Leerzeichen durch das eine Leerzeichen ersetzt werden und schließlich noch am Anfang und Ende alle Leerzeichen gelöscht, und ansonsten benachbarte Leerzeichen durch eins ersetzt werden.

- Einen Typ kann man über `xs:pattern` spezifizieren. Dies sieht dann wie folgt aus.

```
<xs:simpleType name="preis">
    <xs:restriction base="xs:float">
        <xs:pattern value="[0-9]+,[0-9]{2}">      (1.40)
    </xs:restriction>
</xs:simpleType>
```

Ich werde nicht weiter auf reguläre Ausdrücke eingehen. Dazu gibt es eigene Literatur, etwa Friedl 2003. Der angegebene Ausdruck besagt, dass die Zeichenkette, die die reelle Zahl angibt, aus einer nichtleeren Folge von Ziffern besteht, einem Dezimalkomma und zwei obligatorischen Nachkommastellen. Man beachte, dass dies nicht den Wert der Zahl einschränkt als vielmehr die Art ihrer Darstellung. (Man darf also jetzt nicht mehr `3,1` schreiben, sondern es muss `3,10` heißen.)

2. Konstruktion. Typen kann man auch aus anderen Typen konstruieren. Eine Möglichkeit, die man häufiger brauchen wird, ist die Disjunktion. Nehmen wir an, ein Typ ist als `xs:int` vereinbart worden, aber wir wollen den Knoten auch leer lassen. Das geht nicht, ohne dass wir explizit die Möglichkeit vorsehen, dass die leere Zeichenkette eine Instanz des Typs ist. `xs:integer` darf aber nicht leer sein. Hier ist die Lösung: wir konstruieren einen disjunktiven Typ aus `xs:integer` und `xs:string`, wobei letzterer nur leer sein darf. Hier also ein Typ, der die Zahlen

von 0 bis 4 erlaubt und die leere Zeichenkette:

```
<xs:simpleType name="punktTyp">
    <xs:union>
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value=""/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType>
            <xs:restriction base="xs:int">
                <xs:enumeration value="0"/>
                <xs:enumeration value="1"/>
                <xs:enumeration value="2"/>
                <xs:enumeration value="3"/>
                <xs:enumeration value="4"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:union>
</xs:simpleType>
```
(1.41)

Ein anderes Element ist `xs:list`. Damit kann man den Typ einer Liste über einen gegeben Typ definieren. Der Gebrauch ist wie folgt.

```
<xs:list itemType={Typ}>
```
(1.42)

Hierbei ist noch zu beachten, dass die Eingabe der Liste als Zeichenkette erfolgt (wie üblich) und das Leerzeichen als Trennsymbol fungiert, zB `1 2 5 18` für eine Liste von 4 Zahlen oder `der die das` für eine Liste von drei Worten. Zum Beispiel kann es sein, dass wir als Wert eines Attributs nicht eine einzelne Zahl geben wollen sondern eine Liste von Zahlen. Hier ist die entsprechende Definition.

```
<xs:simpleType name="intList">
    <xs:list itemType="xs:int"/>
</xs:simpleType>
<xs:attribute name="zahlliste" type="intList"/>
```
(1.43)

Oder alternativ:

```
<xs:attribute name="zahlliste">
    <xs:simpleType>
        <xs:list itemType="xs:int"/>
    </xs:simpleType>
</xs:attribute>
```
(1.44)

Achtung: Eine einzelne Zahl ist eine Liste von Zahlen, eine leere Liste ist ebenfalls eine Liste von Zahlen. Listen von Objekten eines Typs sind also allgemeiner.

3. Erweiterung. Die dritte Möglichkeit, die man hat, ist die der Erweiterung. Dies betrifft den Fall, wo man einen Knoten hat mit Text und Attributen, und man möchte sowohl den Text wie die Attribute beschränken. Das eine wie das andere ist für sich genommen ein einfacher Typ, aber zusammen wären sie ein komplexer Typ. Dies heißt so etwas auch *simple content complex type*. Die Idee ist, dass man mit dem Textknoten anfängt, diesen typbeschränkt und dann um seine Attribute

erweitert. Ich mache dies am oberen Beispiel vor.

```
<xs:simpleType name="aufgTyp">
    <xs:restriction base="xs:int">
        <xs:minInclusive value="1"/>
        <xs:maxInclusive value="4"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="punktTyp">
    <xs:restriction base="xs:int">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="4"/>
    </xs:restriction>
</xs:simpleType>
<xs:attribute name="nummer" type="aufgTyp"/>
<xs:element name="aufgabe">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="punktTyp">
                <xs:attribute ref="nummer"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
```
(1.45)

Es ist vielleicht interessant zu verstehen, warum dieser Fall gesondert betrachtet wird. Man könnte ja eigentlich eine leere Folge von Untertags hinschreiben, etwa `<xs:sequence/>`. Aber in diesem Fall ist es nicht möglich, das Element selbst, seinen Inhalt zu beschränken. Zunächst handelt es sich um einen einfachen Knoten insofern es keine Töchter gibt. Jedoch muss man nicht nur einen Typ für den Wert des Knotens als solchen angeben, sondern auch je einen Typ für die Attribute, die man hinzufügen will. Die Attributdeklarationen erlauben dies offenkundig. Jedoch fehlt eine Stelle für das Element selbst. Deswegen braucht man also ein Konstrukt braucht, den Typ zu erweitern.

Analog zu `xs:simpleContent` gibt es auch `xs:complexContent`, mit dem es möglich ist, komplexe Typen mittels Restriktion zu definieren. Die folgende

Definition

```
<xs:complexType name="length3">
    <xs:sequence>
        <xs:element name="size"
            type="xs:nonNegativeInteger"/>
        <xs:element name="unit"
            type="xs:NMTOKEN"/>
    </xs:sequence>
</xs:complexType>
```
(1.46)

lässt sich ersetzen durch

```
<xs:complexType name="length2">
    <xs:complexContent>
        <xs:restriction base="xs:anyType">
            <xs:sequence>
                <xs:element name="size"
                    type="xs:nonNegativeInteger"/>
                <xs:element name="unit"
                    type="xs:NMTOKEN"/>
            </xs:sequence>
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>

<xs:element name="depth" type="length2"/>
```
(1.47)

1.4 XSLT

1.4.1 Einführung

Im Folgenden werden wir uns den zwei wichtigen Bestandteilen XSLT und XPath nähern. Zunächst werde ich eine kleine Einführung in XSLT geben. XSLT ist ein Werkzeug, um Dokumente zu *transformieren*. Dies kann zu verschiedenen Zwecken geschehen.

1. Die Information soll verändert werden (Teile sollen gelöscht oder hinzugefügt werden, die Struktur soll sich ändern).

2. Die Information soll in einem Browser angezeigt werden. Dazu muss sie in HTML umgeformt werden.

3. Die Information soll verarbeitet werden (Statistiken sollen geführt werden wie zB bei Excel).

Die Transformation setzt voraus, dass die Daten bereits mit Markup versehen sind. Anything goes, solange wir XML-Markup vorliegen haben. (Es geht auch ohne, aber das ist wesentlich aufwändiger.) In diesem Fall erlaubt uns XSLT die Daten umzuformen, und zwar sowohl in ein anderes Markup-Format (nach HTML oder ein eigenes Format) als auch in ein völlig fremdes Format (zB CSV (= comma-separated values), oder gar pdf).

Grundsätzlich gibt es zwei Möglichkeiten, XSLT einzusetzen: mit Schema und ohne. In dem ersten Fall bekommt der Transformator auch die Informationen über den Typ der Elemente, im zweiten nicht. Im letzten Abschnitt habe ich auf die Vorteile eines Schemas hingewiesen; einer davon ist, dass die Transformation erleichtert wird, weil der Prozessor weiß, welcher Art die Daten sind. Ich weise darauf hin, dass es auch möglich ist, während der Transformation den Typ eines Elements zu beschränken.

Ich gebe ein kommentiertes Beispiel. Die Transformationsdatei bekommt den Namen `kurs.xsl`. Das Suffix `xsl` ist kanonisch festgelegt, der Grundname (hier `kurs`) ist wiederum frei wählbar.

Hier ist ein ganz einfaches Beispiel. Wir erzeugen aus der Datei eine Liste der Teilnehmer eines Kurses in Rohformat. Das Ergebnis kann man sich in einem Editor anzeigen lassen.

```
<?xml version="1.0" encoding="UTF-8"?>
<kurs name="Einf Donaldistik" semester="SS2010">
    <teilnehmer>
        <name>
            <nach>Duck</nach>
            <vor>Donald</vor>
        </name>
        <name>
            <nach>Gans</nach>
            <vor>Gitta</vor>
        </name>
    </teilnehmer>
</kurs>
```
(1.48)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            version="1.0">
<xsl:output method="text"/>
    <xsl:template match="/kurs">
        <xsl:value-of select="@name"/>
        <xsl:text> (</xsl:text><xsl:value-of select="@semester"/>
        <xsl:text>)&#xA;</xsl:text>
        <xsl:text>Teilnehmerliste:&#xA;&#xA;</xsl:text>
        <xsl:for-each select="teilnehmer/name">
            <xsl:value-of select="vor"/>
            <xsl:text> </xsl:text>
            <xsl:value-of select="nach"/>
            <xsl:text>&#xA;</xsl:text>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

(1.49)

Dazu einige Bemerkungen.

- Zeile 1: Der übliche XML-Vorspann.

- Zeile 2: Der XSLT-Vorspann, minimale Version; genauer gesagt ist dies das *Wurzeltag* des Stylesheets. Es ist verpflichtend und darf nur hier verwendet werden. Es legt außerdem Namensräume fest und deren Behandlung bei der Transformation. In diesem Fall ist der Namensraum `xsl`. Der Default der Versionsnummer ist `1.0`, aber ich empfehle, dies mindestens auf `2.0` zu setzen. Wir werden die Funktionen aus der 2.0-Version sehr bald brauchen. (Bei Oxygen muss man den Prozessor manuell anpassen. Saxon6.5.5 reicht nicht aus.) Inzwischen gibt es auch eine Version 3.0, aber wir werden deren Funktionalität nicht benötigen.

- Zeile 3: Das Transformationsschema kennt drei Möglichkeiten der Ausgabe: als XML, als HTML und als Text. Falls wir XML oder HTML wählen, dann tut es einige Dinge vollautomatisch (zum Beispiel den nötigen Vorspann einfügen, damit die Datei nachher auch als XML bzw. HTML erkannt wird). In diesem Fall habe ich als Ausgabeformat `text` gewählt.

- Zeile 4: Grundsätzlich besteht ein Transformationsschema aus einer Reihe von Regeln und einer Angabe, wann diese Regeln „feuern" sollen. Hier gibt es nur eine einzige Regel, und sie soll auf alle Knoten namens `kurs` angewendet werden. Regeln werden durch das Tag `xsl:template` kenntlich gemacht. Wann sie feuern sollen, wird als Wert von `match` angegeben. Der Wert von `match` ist ein XPath-Ausdruck. Wir werden uns noch gründlich mit XPath auseinandersetzen. Für den Moment soll es ausreichen, dass der Schrägstrich sagt: ist Tochter des Dokumentknotens, und dass der Name des Tags hinter dem Schrägstrich folgt. Regeln müssen keinen Namen haben.

- Zeile 5: Mit `xsl:value-of` bekommen wir den Wert eines Ausdrucks. Solche Ausdrücke sind wiederum XPath-Ausdrücke. Der Wert solcher Ausdrücke sind stets Listen von Knoten. Was der Wert eines Knotens ist, wird noch im Einzelnen festgelegt werden. Im vorliegenden Fall ist `@name` ein Attribut, und `xsl:value-of` erfragt den Wert des Attributs. Der Wert eines Attributs ist, sofern nichts anderes vereinbart ist, eine Zeichenkette.

- Zeile 6: Das Tag `<xsl:text>` dient dazu, dass man auszugebenden Text angeben kann. In diesem Fall soll eine öffnende Klammer ausgegeben werden. Diese Ausgabe erfolgt nach der aus Zeile 5, und vor den folgenden Zeilen.

- Zeile 7: `
` ist nicht das, was eigentlich am Ende ausgegeben wird; was wir tatsächlich sehen, ist ein Zeilenumbruch. Wie kommt es dazu? Zunächst einmal ist die Grundlage von XSLT natürlich XML, und das wiederum ist eine Erweiterung von HTML. In HTML hat man sogenannte Entities geschaffen. Das sind Zeichenketten, die links vom Kaufmanns-Und (`&`) und rechts vom Semikolon (`;`) eingerahmt sind. Jedesmal, wenn der Prozessor von links kommend ein Kaufmanns-Und entdeckt, vermutet er ein Sonderzeichen und schaltet in einen anderen Modus. Die folgende Zeichenkette wird bis zum Semikolon gelesen und dann ausgewertet. (Wenn er nichts Passendes findet, dann überlegt er als Alternative, ob das Kaufmanns-Und vielleicht doch ein echtes Symbol ist.) Auf diese Weise kann man mit einer einfachen Tastatur einen riesigen Vorrat an Zeichen ausgeben (zB über Unicode, doch davon später). Auf der anderen Seite erlaubt es uns, Zeichen auszuwerfen, die zwar auf der Tastatur vorhanden sind, die aber eine besondere Bedeutung haben. Dazu gehören: das Kaufmanns-Und und die spitzen Klammern. Leerzeichen und neue Zeile sind irgendwie dazwischen. Ich empfehle, wegen der Optik des XSLT-Schemas (also, was wir im Editor

bzw auf dem Schirm sehen) von dem echten Zeilenumbruch *keinen Gebrauch zu machen* (den es ja als Taste durchaus gibt), sondern stattdessen die entsprechende HTML-Entität zu benutzen, und die ist `
`. XSLT macht also von der HTML-Konvention Gebrauch und normalisiert unseren Text. Das kann man zwar abschalten, aber auch davon rate ich ab.

- Zeile 9: Mit `xsl:for-each` wird eine Schleife definiert. Das Prinzip ist wie bei `xsl:template`. Wir definieren (in XPath) eine Bedingung für die Knoten, auf die die Transformation angewendet werden soll, und sagen, sie soll auf alle solche Knoten angewendet werden. (XPath wirft eine Liste von Knoten aus, die XSLT in der Reihenfolge abarbeitet, in der es sie bekommt.) Im vorliegenden Fall sagt `teilnehmer/name`: alle Namensknoten unterhalb von Teilnehmerknoten unterhalb des aktuellen Knotens.

- Zeile 11: Falls wir ein Leerzeichen wollen, müssen wir das unbedingt sagen. Und das Leerzeichen muss wiederum in `xsl:text` eingeschlossen werden, sonst wird es komplett ignoriert!

(Spaßvögel sollten es sich nicht entgehen lassen, zu überlegen, wie man HTML dazu bringen kann, `
` so anzuzeigen, wie Sie es hier sehen. Bedenken Sie, dass HTML dem Kaufmanns-Und eine besondere Bedeutung zumisst. Das bedeutet: Was Sie sehen, ist garantiert nicht das, was man eintippen muss.)

Das vorige Beispiel modifizieren wir etwas. Wir erzeugen aus der Datei eine Liste der Teilnehmer in HTML Format. Das Ergebnis kann man sich in einem

Browser (was etwas anderes ist als ein Editor) anzeigen lassen.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" version="4.0"
    encoding="UTF-8" indent="yes"/>
    <xsl:template match="/kurs">
        <html>
            <head></head>
            <body bgcolor="moccasin">
            <h2> <xsl:value-of select="@name"/>
                <xsl:text>(</xsl:text>
                <xsl:value-of select="@semester"/>
                <xsl:text>)</xsl:text>
            </h2>
            <h4>Teilnehmerliste</h4>
            <p>
            <ul>
                <xsl:for-each select="teilnehmer/name">
                    <li><xsl:value-of select="vor"/>
                    <xsl:text> </xsl:text>
                    <xsl:value-of select="nach"/></li>
                </xsl:for-each>
            </ul>
        </p>
        </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

(1.50)

Im Prinzip hat sich die Struktur des XSLT-Schemas nicht verändert. Nur zwei Dinge sind zu bemerken.

- Damit wir HTML mit dem nötigen Vorspann bekommen (und auch eine akzeptable Ausgabe), müssen wir `method` auf `html` setzen. Dazu können wir dann noch ein paar Dinge wählen, zB welche Version von HTML (für den Vorspann) oder ob wir automatische Indentierung wollen.

- Wenn wir HTML wählen, ist das Markup fast genau wie in XML. Im Unterschied zu Schema ist jetzt die Mischung von Text und Markup kein Problem. Das heißt, wir können sowohl XSLT-Tags wie auch HTML-Tags mischen. Ein XSLT-Tag ist ja durch das Namenspräfix `xsl` kenntlich gemacht. Tags mit diesem Präfix werden dann als XSLT-Kommandos behandelt, alle anderen werden als Text ausgegeben. Die HTML-Tags also schlicht als Text behandelt, das heißt so, wie sie sind, in die Datei ausgegeben. Der Browser wird sie dann so vorfinden, und dann können sie bei der Anzeige die nötige Wirkung entfalten. Die Markup Tags habe ich rot hervorgehoben.
- Im Gegensatz zu der `text` Methode ist es (jedenfalls im vorliegenden Beispiel) nicht nötig, sich mit dem Zeilenumbruch zu befassen. Das hat zwei Gründe. Zunächst einmal müssen wir uns klarmachen, dass es zwei Zeilenumbrüche gibt, über die wir uns Gedanken machen müssen: die eine Sorte ist die in der auszugebenden Datei (die Quelldatei für den Browser), die andere ist die auf der angezeigten Seite im Browser. Die erste Sorte ist für die Übersicht (also technisch gesehen entbehrlich), die zweite für das angezeigte Ergebnis. Umbrüche in der Datei erledigt der XSLT-Prozessor. Er erzeugt schon eine optisch nette Ausgabe (nur lange Textzeilen können Ausnahmen darstellen). Umbrüche auf der angezeigten Seite hingegen sind die Aufgabe des Browsers. Und die werden jetzt über HTML gesteuert (wozu ich demnächst noch etwas sagen werde). Im vorliegenden Fall geben wir eine Liste aus, und jedes `<li>` Tag erzeugt eine neue Zeile mit einer Kugel davor.

1.4.2 Der Prozessor

Die Kommandos findet man in vielen Büchern beschrieben. Man kann die Einzelheiten bei Bedarf immer nachlesen, die technischen Dokumente sind ja alle online verfügbar. Ich bespreche deswegen auch nicht alle Elemente sondern beschränke mich auf konzeptuelle Klärung. Das Wesentliche daran sollte man sich jedoch gründlich merken.

> [Die Arbeitweise des Prozessors]
> XSLT arbeitet mit einem Fokus, den ich hier nenne. Nehmen wir an, der Fokus sei auf y gesetzt. Wenn XSLT nun mittels `<xsl:for-each>` über das Attribut `select="{xpfad}"` aufgefordert wird, etwas zu tun, so ruft es XPath auf und lässt sich die Liste $L = [x_1; x_2; x_3; \cdots ; x_n]$ der Knoten liefern, die vom Fokus (also y) aus im Bereich des Pfades `xpfad` liegen. (Mittels `<xsl:sort>` kann man diese Liste vor

> der Abarbeitung umsortieren.) Für alle i von 1 bis n führt XSLT die Aktion im Rumpf von der Prozedur für x_i aus. Die Zahl i bekommt man durch Aufruf von `position()`. Ebenso für das Attribute `match`, dessen Wert für einen Knoten x entscheidet, ob die Schablone auf x angewendet wird. Indem Moment, wo die Schablone angewendet wird, ist x der neue Fokus.

Und:

> [Fokus]
> Wir unterscheiden zwischen Operationen, die den Fokus verändern und solchen, die es nicht tun. Falls der Fokus verändert wird, so ändert er sich von y zu x_1 und durchschreitet die Liste L (siehe *Arbeitsweise des Prozessors* oben). Der Fokus (also hier) wechselt dann von y zu den Knoten x_1 bis x_n. Alle Pfade werden dann also relativ zu x_i ausgerechnet. Anschließend fällt der Fokus nach y zurück. Ändert sich der Fokus bei der Operation nicht, so bleibt er also stets bei y. Pfade müssen dann weiterhin relativ zu y angegeben werden.

1.4.3 Baummanipulationen

Mittels XSLT kann man die Baumstruktur verändern. Man kann unter anderem

- Knoten wegnehmen
- Knoten hinzufügen
- Attribute wegnehmen
- Attribute hinzufügen

Das Hinzufügen von Knoten geschieht auf zwei Weisen. Die eine ist, sie einfach als Textknoten einzusetzen, wie wir das schon in HTML getan haben. Solange sie nicht den Namensraum `xsl` tragen, geht das ohne Komplikationen. Daneben existiert noch ein Kommando `<xsl:element>`. Man benötigt ein solches explizites Instrument immer dann, wenn der wirkliche Text (dh der Namensraum oder das Tag) erst noch berechnet werden muss. Ferner stehen zwei Kommandos zur Verfügung: `<xsl:copy>` sowie `<xsl:copy-of>` (ferner auch `<xsl:variable>`, aber davon später).

- <xsl:copy> erzeugt eine Kopie des Knotens ohne Attribute und ohne Unterknoten;
- <xsl:copy-of> erzeugt den gesamten Unterbaum.

Das Kommando <xsl:attribute> kann man dazu benutzen, Attribute einzusetzen. Als Default werden sie dem aktuellen Knoten zugefügt, der gerade erzeugt wird. Man kann die Werte aber auch speichern und später verwenden.

Genauso stellt <xsl:comment> eine Möglichkeit dar, Kommentar zu erzeugen. Die Idee dahinter ist, dass ein Kommentar möglicherweise komplex ist (zB wenn wir eine aktuelle Datumsangabe automatisch erzeugen wollen). Das kann es notwending machen, dass wir die von XSLT bereitgestellten Funktionen auch zur Erzeugung von Kommentar nutzen wollen.

Instruktiv und als Beispiel für viele soll die Funktion <xsl:variable> besprochen werden. Diese wird wie folgt benutzt:

```
<xsl:variable name="varname">
```
(1.51)

Der Name ist dabei obligatorisch. Es gibt zwei optionale Attribute:

1. as, welches den Typ spezifiziert (zum Beispiel as="xs:string"),
2. select, mit Pfaden als Werten (zum Beispiel select="name/vorname").

Dabei darf nicht (1) *und* (2) gleichzeitig gegeben sein. Es geht also entweder

```
<xsl:variable name="n">
    Duck
</xsl:variable>
```
(1.52)

oder

```
<xsl:variable name="n"
    select="/name/nach" />
```
(1.53)

Dies definiert also eine Variable namens n mit Wert Duck. Den Wert der Variablen kann man sich durch den Befehl $n geben lassen (also: ein Dollarzeichen umittelbar gefolgt von dem Namen der Variablen). Beispiel:

```
<xsl:value-of select="$n" />
```
(1.54)

Wenn gleichzeitig der Typ gegeben ist, wird der Wert auch auf seinen Typ geprüft.

Eine Variable muss keineswegs einfach sein. Wir können einer Variablen auch Bäume und Listen zuweisen. Variable sind allerdings eines nicht: variabel. Zwar gibt es in XPath auch Anweisungen der Form for $i in 1 to 5, aber das bedeutet lediglich, dass $i eine Variable ist, während die Variablen mit dem Namen n im obigen Beispiel nicht einfach neu zugewiesen werden kann.

1.4.4 Steuerung

Zunächst einmal genügt die Angabe eines Templates, um eine Transformation zu erzeugen. Jedoch wird man des öfteren merken, dass man die Abarbeitung gezielt steuern will. Die Syntax ist

```
<xsl:apply-templates
    select ? = {xpfad}
    mode ? = {ausdruck}>
    <!-- xsl:sort oder xsl:param Knoten -->
</xsl:apply-templates>
```
(1.55)

Dabei ist `select` dafür zuständig, eine Knotenmenge auszuwählen. (Default ist der Fokus.) Bemerkenswert ist hier das Attribut `mode`. `<xsl:template>` erlaubt ein Attribut namens `mode`. Der Wert kann ein beliebiger Name sein. Wichtig ist, dass, wenn `mode` gesetzt wird, das Template nur dann benutzt wird, wenn `mode` den richtigen Wert hat. Das kann man sich wie folgt zu Nutze machen. Man kann das Verhalten der Transformation über den Modus steuern. Diesen kann man zB als Parameter dem Prozessor übergeben. So kann man in einem einzigen Stylesheet verschiedene Möglichkeiten der Behandlung unterbringen.

Dazu benötigt man lediglich logische Verzweigungen und bedingte Anweisungen, und diese gibt es auch in XSLT. Das einfachste ist `<xsl:if>`. Die Syntax ist

```
<xsl:if
    test = {xtest}>
    <!-- auszuführende Operation -->
</xsl:if>
```
(1.56)

Der Testausdruck wird an XPath weitergereicht (wie Tests aussehen, bespreche

ich im nächsten Abschnitt). Hat man Alternativen, so benutzt man

```
<xsl:choose>
    <xsl:when
        test = {xtest1}>
        <!-- auszuführende Operation1 -->
    </xsl:when>
    <xsl:when
        test = {xtest2}>                          (1.57)
        <!-- auszuführende Operation2 -->
    </xsl:when>
    <xsl:otherwise>
        <!-- auszuführende Operation3 -->
    </xsl:otherwise>
</xsl:choose>
```

Es ist möglich, beliebig viele <xsl:when> Bedingungen unterzubringen. Diese werden in der Reihenfolge abgearbeitet, in der sie stehen. Falls keine Klausel greift, wird <xsl:otherwise> genommen. Das ><xsl:otherwise> Element darf fehlen (dann wird natürlich nichts unternommen).

Grundsätzlich ist <xsl:template> schon ausreichend, um eine Transformation auf alle Knoten durchzuführen. Trotzdem stellt XSLT noch eine weitere Funktion bereit, nämlich <xsl:for-each>. Diese kann auch für beliebige Listen verwendet werden. Beispiel

```
<xsl:for-each select="1 to 5">
    <!-- auszuführende Operation -->          (1.58)
</xsl:for-each>
```

1.4.5 Ausgabeformatierung

Neben der reinen visuellen Formatierung (die der Browser oder Editor übernimmt), und der Berechnung (die XPath zu erledigen hat), gibt es noch Aufgaben, die XSLT selbst zufallen. Dazu gehört die Konversion von Zahlen in Text. Dazu existiert ein Befehl, <xsl:number>. Ferner, und das ist das Entscheidende, gibt es noch Funktionen, nämlich format-number und format-date (sowie format-dateTime, format-time).

Haben wir eine Variable, sagen wir ergebnis, deklariert, so können wir den Wert wie folgt anzeigen lassen:

```
<xsl:value-of select="$ergebnis" />          (1.59)
```

Das genügt schon für gewöhnliche Zwecke, zB wenn wir das Dokument nicht validiert haben. Ist aber der Wert eine Zahl, so müssen wir Folgendes tun:

```
<xsl:number value="$ergebnis" />
```
(1.60)

Dabei können wir die Zahl sowohl als Kardinalzahl wie auch als Ordinalzahl ausgeben (indem man das Ordinalsuffix als Wert von `ordinal` angibt), als Buchstaben oder Zahlenwerte (`letter-value` mit Werten `"alphabetic"` oder `"traditional"`). Als Alternative zu `value` (welche die Zahl selbst angibt) können wir auch das Attribut `select` wählen, welches erlaubt, den Wert des Knotens zu berechnen.

Eine wichtige Anwendung von `number` ist die fortlaufende Nummerierung von Elementen. Dazu gibt es eine XPath Funktion `position()`, welche die Position des Fokus angibt (in einer abzuarbeitenden Liste). Diese kann man sich anzeigen lassen:

```
<xsl:number value="position()" />
```
(1.61)

Lässt man sowohl `value` wie `select` weg, so nimmt XSLT die Position als Default.

Die Funktion `format-number` erwartet zwei Argumente, nämlich eine Zahl und einen sogenannten *picture string* oder *Vorbild(kette)*. Dieser zeigt an, wie die Zahl in der Ausgabe formatiert werden soll. Die Art, wie die Vorbildkette interpretiert werden soll, wird durch `<xsl:decimal-format>` festgelegt. Etwa so:

```
<xsl:decimal-format decimal-separator=","
    grouping-separator="."/>
```
(1.62)

Dies regelt, dass die Gruppen durch einen Punkt abgetrennt werden sowie, dass Dezimalzahlen mit Hilfe eines Kommas und nicht — wie in England oder Amerika üblich — durch einen Punkt ausgegeben werden. Damit sind wir fertig, und wir können Zahlen in unserem Format ausgeben. Zum Beispiel

```
<xsl:number value="format-number($ergebnis,
    '0.000,00')"/>
```
(1.63)

Dies legt fest, dass nur zwei Stellen hinter dem Komma ausgegeben werden sollen und Dreierblöcke durch einen Punkt getrennt werden sollen. Hierbei ist `'0.000,00'` der *picture string* oder *Vorbildkette*. Die Zahlenwerte sind unerheblich; mit Hilfe der Vorkommastellen kann man steuern, wie viele Stellen vor dem Komma erscheinen *müssen*. Steht eine `0`, so muss die entsprechende Stelle durch eine Ziffer

repräsentiert werden. Steht anstelle der `0` das Doppelkreuz `#`, dann darf die Ziffer wegfallen, wenn sie nicht signifikant ist. Das benötigt man, weil das Vorbild ja aus Blöcken bestehen muss. So können wir mittels `#.##0,0#` festlegen, dass der Punkt Dreierblöcke abteilt, aber alle Ziffern außer der Vorkommaziffer wegfallen dürfen. Analog sagt die Kette, dass *eine* Nachkommastelle wenigstens vorhanden sein muss aber nur zwei dastehen dürfen. Sind die Zahlen also größer, als es das Vorbild zeigt, dann werden mehr Ziffern angezeigt. Dann folgt das Komma als Trennsymbol, und dann noch zwei Ziffern. Diese geben die Anzahl der Nachkommastellen wieder. Man beachte, dass im Gegensatz zu den Stellen vor dem Komma, die je nach Größe immer mehr werden, die Stellen nach dem Komma jedoch durch die angegebenen Symbole begrenzt sind. Nur so viele Nachkommastellen werden angezeigt, wie es zusammen Vorkommen von `0` und `#` nach dem Komma gibt. Weitere Einzelheiten finden sich hier.

Anmerkung. Man beachte den Gebrauch von Anführungszeichen. Die Vorbildkette wird deswegen in einfache Anführungszeichen gesetzt, weil wir nicht wollen, dass XSLT denkt, der XPath Ausdruck sei schon zu Ende.

Hier noch einmal eine schöne Anwendung. Angenommen, wir wollen in unsere Ausgabedatei eine Kommentarzeile aufnehmen, die sagt, wann unsere Datei erstellt worden ist. In diesem Fall können wir den String nicht direkt angeben, er muss automatisch erzeugt werden.

```
<xsl:comment>
    <xsl:text> Erstellt am </xsl:text>
    <xsl:value-of select="format-dateTime(current-dateTime(),
'[D].[M].[Y] um [H]:[m]:[s]')"/>
</xsl:comment>
```
(1.64)

Dies erzeugt folgende Ausgabe:

```
<!-- Erstellt am 26.05.2009 um 17:36:25 -->
```
(1.65)

Hierbei ist natürlich das Datum sowie die Uhrzeit die vom Computer registrierte Zeit, zu der wir die Zieldatei mittels XSLT erstellt haben. Hierbei ist `'[D].[M].[Y] um [H]:[m]:[s]'` wieder ein sogenanntes *Vorbild*. Die in eckigen Klammern eingeschlossen Symbole sind von fester Bedeutung,welche in den einschlägigen Handbüchern eingesehen werden kann. Alles andere wird so wiedergegeben wie aufgeschrieben, zum Beispiel die Trennzeichen (der Punkt, der Doppelpunkt) sowie das Wort `um`. Die allgemeinen Formatierungsregeln kann man hier nachlesen.

1.5 XPath

1.5.1 Bäume und Relationen

Eine (2-stellige) *Relation* auf einer Menge M ist eine Teilmenge von $M \times M$, dh eine Menge von Paaren (x, y), wo $x, y \in M$. Es gibt einige wichtige Operationen auf Relationen, die wir im Folgenden benötigen werden.

- $\Delta_M := \{(x, x) : x \in M\}$ (die *Diagonale*)
- $R \circ S := \{(x, z) :$ es existiert $y : (x, y) \in R$ und $(y, z) \in S\}$ (das *Relationenprodukt*)
- $R \cup S := \{(x, y) : (x, y) \in R$ oder $(x, y) \in S\}$
- $R \cap S := \{(x, y) : (x, y) \in R$ und $(x, y) \in S\}$
- $R - S := \{(x, y) : (x, y) \in R$ und $(x, y) \notin S\}$.
- $R^\smile := \{(y, x) : (x, y) \in R\}$ (die zu R *konverse* Relation)
- $R^+ := R \cup R \circ R \cup R \circ R \circ R \cup \cdots$ (die *transitive Hülle* von R)
- $R^* := \Delta_M \cup R^+$ (die *reflexive transitive Hülle* von R)

Zu R ist

- $b(R) := \{x :$ es existiert ein y mit $(x, y) \in R\}$ die Menge der *Beginnpunkte* von R und
- $e(R) := \{y :$ es existiert ein x mit $(x, y) \in R\}$ die Menge der *Endpunkte* von R.

Dann ist $b(R) = e(R^\smile)$ sowie $b(R) = e(R^\smile)$.

Ich gebe noch ein paar Rechenregeln mit Relationen an.

$$\begin{aligned}
\Delta_M^\smile &= \Delta_M \\
R^{\smile\smile} &= R \\
(R \circ S)^\smile &= S^\smile \circ R^\smile \\
(R \cup S)^\smile &= R^\smile \cup S^\smile \\
(R \cap S)^\smile &= R^\smile \cap S^\smile \\
(R^+)^\smile &= (R^\smile)^+ \\
(R^*)^\smile &= (R^\smile)^*
\end{aligned} \tag{1.66}$$

Es folgt daraus, dass die Konversionbildung nach innen geschoben werden kann. Es gibt deswegen in XPath keinen Operator, mit dem man eine Relation umdrehen kann, weil er unnötig ist, weil zu jeder Basisrelation die konverse ebenfalls existiert. Außerdem wird man feststellen, dass sich mit der Konversenbildung nicht leicht rechnen lässt.

Wie funktioniert XPath?

Eine schöne Übersicht findet sich auch auf Selfhtml. In XSLT haben wir schon den Aufruf `<xsl:template match="{Pfad}">` gesehen. Beim Aufruf von `xsl:template` wird XPath gerufen, um den Ausdruck `Pfad` auszuwerten. Dabei muss XPath beantworten, ob das Template anzuwenden ist oder nicht. Dies geschieht anhand des Pfadausdrucks. Dabei dient der Pfadausdruck als Schema, und gefragt ist, ob der Pfad des behandelten Elements unter das Schema fällt. Dabei muss das Schema nur ein *Endstück* des echten Pfades spezifizieren. So fällt der Pfad `/bibliothek/buchliste/buch` unter das Schema `buch`, weil `buch` ein Endstück von `/bibliothek/buchliste/buch` ist. Hingegen geschieht bei `<xsl:for-each select="{Pfad}">` etwas anderes. Hier wird XPath gerufen, um den Pfadausdruck auszuwerten. Das Resultat dieser Auswertung ist eine *Liste* von Knoten. XSLT wird dann den Rumpf der Prozedur für jedes Element dieser Liste ausführen. Die Ausführungsordnung ist durch die Liste vorgegeben. Diese Ordnung kommt ihrerseits aus der Ordnung in der Datei (wie sie im DOM kodiert ist). In `{Pfad}` steht deswegen ein Ausdruck, der eine Relation R definiert. Von dieser Relation gibt uns XPath dann die Liste der Endpunkte. Zum Beispiel bedeutet `select="./teilnehmer"` etwa dies: „Betrachte die Paare (x, y), die ein `teilnehmer` Tag haben, und wo x der momentane Knoten ist. Dann gib mir davon die Liste dieser y." Im Einzelnen bleibt noch zu bestimmen, was der *momentane Knoten* oder *Fokus* ist. Wichtig ist, dass die Ausführung einer Operation den Fokus verschieben kann und man deswegen genau beachten muss, was man wann tut.

Pfadausdrücke gibt es in einer Langfassung und einer Kurzfassung. Es ist besser, die zunächst Langfassung zu betrachten und erst dann die Kurzfassung. Pfadausdrücke entstehen als Relationenprodukt aus Einzelausdrücken. Es gibt die Verkettungssymbole `/` und `//`. (*Achtung:* Die Dokumentwurzel wird auch mit `/` gekennzeichnet, was zu einer gewissen Verwirrung Anlass gibt!) Ein Einzelschritt wiederum hat die Form `Schritt(ausdruck)::Prädikat`. Das bedeutet: der Schrittausdruck steuert die Bewegung, und das Prädikat nimmt die Auswertung vor. Man kann das auch einen *Test* nennen, obwohl kleine Unterschiede bestehen (siehe das nächste Kapitel). Für jetzt soll das Prädikat `node()` ausreichen.

Es trifft auf alle Knoten dh auf alle Elemente von E zu.
Syntax der Relationsausdrücke:

1. `{Schritt}::{Prädikat}`
   Relativer Ausdruck: ergibt die Liste aller Knoten y derart, dass $(x, y) \in$ {`Schritt`} und x der Fokus ist (das „hier") und y erfüllt `Prädikat`.

2. `/{Schritt}::{Prädikat}`
   Absoluter Ausdruck: ergibt die Liste aller Knoten y derart, dass $(r, y) \in$ {`Schritt`} und r die Dokumentwurzel und y erfüllt {`Prädikat`}

Folgendes sind Prädikate:

- `node()` Trifft auf alle Knoten zu.
- `element()`, `attribute()`, `namespace()`
- `comment()`, `text()`, `processing-instruction()` Diese Funktionen erlauben ein Argument. Achtung: Diese Knoten sind außerhalb des DOM, also auch außerhalb der Dokumentordnung!
- Eine Zeichenkette (sofern sie ein Tagname, dh ein QName sein kann).
- * Steht für einen beliebigen Tagnamen. Technisch sind dies QNamen, und diese dürfen auch einen Doppelpunkt enthalten (womit sie eine Referenz auf Namensraum enthalten können).

Kurzformen:

Kurzfassung	Langfassung
`{Prädikat}`	`child::{Prädikat}`
`//`	`/descendant-or-self::node()/`
`@{Attribut}`	`attribute::{Attribut}`
`..`	`parent::node()`
`.`	`self::node()`

Beispiele.

- `/child::bibliothek` Alle Töchterknoten der Wurzel, welche das Tag `bibliothek` haben. Kurzform: `/bibliothek`
- `child::autor` Alle Töchterknoten des Fokus, welche das Tag `autor` besitzen. Kurzform: `autor`

- `attribute::isbn` Der Wert des Attributs `isbn` des gegenwärtigen Fokus. Kurzform: `@isbn`

Zu guter Letzt darf man Pfade auch verketten: Es gibt einen Operator, geschrieben `/`, dessen Interpretation genau die Verkettung der Pfade ist. Das bedeutet, dass die Pfade, die den Ausdruck

$$\texttt{\{Schritt}_1\texttt{\}::\{Prädikat}_1\texttt{\}/\{Schritt}_2\texttt{\}::\{Prädikat}_2\texttt{\}} \tag{1.67}$$

erfüllen, genau die Verkettung der Pfade ist, die $\texttt{\{Schritt}_1\texttt{\}::\{Prädikat}_1\texttt{\}}$ erfüllen mit denen, die $\texttt{\{Schritt}_2\texttt{\}::\{Prädikat}_2\texttt{\}}$ erfüllen. Dazu gibt es noch den Operator `//`, wobei Pfade genau dann

$$\texttt{\{Schritt}_1\texttt{\}::\{Prädikat}_1\texttt{\}//\{Schritt}_2\texttt{\}::\{Prädikat}_2\texttt{\}} \tag{1.68}$$

erfüllen, wenn sie

$$\texttt{\{Schritt}_1\texttt{\}::\{Prädikat}_1\texttt{\}/descendant-or-self::*/}\texttt{\{Schritt}_2\texttt{\}::\{Prädikat}_2\texttt{\}} \tag{1.69}$$

erfüllen. In der Praxis heißt das, dass der zweite Pfad unterhalb des ersten liegt, aber nicht unmittelbar anschließen muss.

Relationen in XPath

Hier ist eine vollständige Liste der Relationen. Ich erinnere an das DOM (E, K, L, f, t). Hierbei ist K die Relation der unmittelbaren Dominanz, und L die Relation *ist unmittelbare linke Schwester von*. Hier ist eine Liste der primitiven Relationen.

Code	Relation
`self`	Δ_E
`child`	K
`parent`	$K^\smile$
`descendant`	K^+
`descendant-or-self`	K^*
`ancestor`	$(K^\smile)^+$
`ancestor-or-self`	$(K^\smile)^*$
`following`	$(K^\smile)^* \circ L^+ \circ K^*$
`following-sibling`	L^+
`preceding`	$(K^\smile)^* \circ (L^\smile)^+ \circ K^*$
`preceding-sibling`	$(L^\smile)^+$
`attribute`	(nicht repräsentiert)
`namespace`	(nicht repräsentiert)

Aus diesen können wir nun komplexe Relationen zusammenbauen. Hier sind Beispiele.

- `self::teilnehmer`
  $\{(x, x) : x$ hat das Tag `teilnehmer`$\}$

- `child::name/attribute::lang`
  $\{(x, y) : y$ hat ein Attribut namens `lang` und ist `name` Tochter von $x\}$

- `ancestor-or-self::node()/following::node()`
  $\{(x, z) : z$ folgt y und y ist Vorfahre oder gleich mit $x\}$

- `attribute::matrikel`
  $\{x : x$ hat das Attribut `matrikel`$\}$

Diese Relationen werden auf den Fokus bezogen. Nehmen wir zB die Relation zu

`child::name/attribute::lang` (1.70)

Ist a der Fokus, so berechnet der Prozessor die Menge

$\{y : y$ hat ein Attribut namens `lang` und ist `name` Tochter von $a\}$ (1.71)

Diese entsteht durch Einsetzen von a für x in der Relation.

Ähnlich ist es etwa, wenn wir die Bedeutung des Wortes *Vorgesetzter* als eine Relation zwischen Personen begreifen (also die Menge aller (a, b), wo a Vorgesetzter von b ist) und dann die Person a fixieren. Dann bekommen wir die Menge aller Personen, deren Vorgesetzter a ist. Die Ausgabe in XPath ist streng genommen aber keine Menge sondern eine Liste, die entweder entsprechend der Reihenfolge im Dokument geordnet ist oder per `xsl:sort` Anweisung, wie in den Übungen besprochen.

Hinweis. Bei `following`, `following-sibling`, `preceding` und `preceding-sibling` bekommen Sie auch die Attributknoten (siehe dazu auch die Definition der Dokumentordnung auf Seite 15). Wollen Sie diese ausschließen, müssen Sie anstelle von `node()` das Prädikat `element()` verwenden. Die Namensraum- und Attributknoten sind jedoch *nicht* Unterknoten. Mit Hilfe von `child`, `parent`, `ancestor` etc. bewegt man sich nur zwischen Elementknoten.

1.5.2 Tests

Wie schon erwähnt, ist XSLT nur für die Ausführung der Transformation zuständig, während das Rechnen Aufgabe von XPath ist. XPath rechnet mit Zahlen, Zeichenketten, Datumsangaben, Booleschen Werten, und eben auch mit Knoten. Das Attribut `select` has als Wert einen XPath Ausdruck, der einen Relation benennt. Mit dem Fokus zusammen ergibt diese Relation eine Eigenschaft von Knoten. XPath berechnet aus diesem Test eine Liste von Knoten und gibt diese an XSLT zurück.

In diesem Abschnitt befasse ich mich ausführlicher mit den Tests. Dazu muss ich die Aussagen über Pfadausdrücke noch einmal beleuchten. Ein Einzelschritt hat die Syntax `{Pfad}::{Prädikat}`. Faktisch hat sich dabei ergeben, dass sowohl `{Pfad}` wie auch `{Prädikat}` relational sind und deswegen eine Liste auswerfen. So ist `descendant::{Name}` die Aufforderung, zunächst die Nachfahren abzuprüfen. `{Name}` wirft die Knoten mit Tag `{Name}` aus. Gekoppelt mit dem Nachfahren-Schritt bekommen wir also eine Liste. Wiederum sollte man sich die Semantik von `::` als relationale Verkettung ($\circ$) vorstellen. Hinzu kommen jetzt noch die sogenannten *Filterausdrücke*. Die volle Syntax ist mithin:

$$\texttt{\{Schritt\}::\{Prädikat\}[\{Filter}_1\texttt{\}][\{Filter}_2\texttt{\}]}\cdots\texttt{[\{Filter}_n\texttt{\}]} \quad (1.72)$$

Der Ausdruck $\texttt{\{Filter}_i\texttt{\}}$ ist vom Typ `xs:boolean` oder positive ganze Zahl. Das bedeutet, dass er eine Bedingung an Knoten angibt. Die Semantik der eckigen Klammern, also $[\cdots]$, ist die des Schnitts, genauer: Die Liste L wird geschnitten mit der Knotenmenge M, die den Ausdruck in den Klammern erfüllt. Das Ergebnis ist dann diejenige Teilliste von L, die dadurch entsteht, das aus L alle Elemente gestrichen werden, die nicht in M vorkommen. Damit ist nun gesagt, was Filter sind. Ich werde nun einige solche Filterausdrücke vorstellen.

Tags und Attribute abfragen

Eine häufige Verwendung von Tests ist sicher die, dass man den Wert eines Attributs abfragen will. Dies geht wie folgt:

```
child::node()[@{Attribut} = {Wert}]          (1.73)
```

(Hierbei sind die Ausdrücke `{Attribut}` und `{Wert}` jeweils Variable für beliebige Namen von Attributen und Werten, also nicht selbst Zeichenketten.) Dies

kann man auch wie folgt abkürzen:

```
*[@{Attribut} = {Wert}]
```
(1.74)

Der Stern wird hier aus syntaktischen Gründen gebraucht. Dieser ist wie bei Unix üblich ein Joker: er matcht jede Zeichenkette. Man nennt so etwas eine *Wildcard*. In XSLT matcht * zunächst einmal jeden Namen; da aber der Name und die Sequenz bestehend aus `child::` und diesem Namen synonym sind, kann man * anstelle von `child::*` verwenden. Ich weise noch darauf hin, dass `node()` durch `element()` ersetzt werden kann.

Die Position abfragen

Wie schon gesagt, kann der Testausdruck auch ein Zahlausdruck eine Zahl sein. In diesem Fall wird dieser Ausdruck als der folgende Test interpretiert

```
position() = {Zahl}
```
(1.75)

Der Filterausdruck `[23]` ist also kurz für `[position () = 23]`. Beachten Sie, dass dies die einzige Interpretation dieser Zahl ist. Sie bezeichnet die Position. So ist also

```
following-sibling::node()[23]
```
(1.76)

dasselbe wie

```
following-sibling::node()[position() = 23]
```
(1.77)

und bedeutet, dass wir unter den folgenden Schwesterknoten den 23ten nehmen. (Wobei hier gemeint sein kann, dass wir den 23. Knoten unter allen Schwesterknoten nehmen. Die Interpretation ist so, dass `following-sibling` eine Liste von Knoten auswirft, die durch `node()` nicht weiter eingeschränkt wird. *Unter diesen* wird der 23. ausgewählt. Das heißt insgesamt, dass wir unter den nachfolgenden Schwesterknoten 23 Schritte weitergehen und diesen (und nur diesen) Knoten auswählen.

Natürlich kann man auch andere Filter mit Positionen formulieren. Diese können allerdings *nicht* wie oben abgekürzt werden!

1. `following-sibling::node()[position() < 23]`
2. `following-sibling::node()[position() != 23]`

3. `following-sibling::node()[position() != last()]`

Diese wählen alle Knoten vor dem 23ten, alle außer dem 23ten, nzw. alle außer dem letzten aus. Der letzte Ausdruck verwendet die Funktion `last()`, welche wahr ist, wenn die Position die letzte in der Liste ist. (Man überlege sich, ob es eine Funktion `first()` geben muss, oder ob diese im Prinzip überflüssig ist.)

Es gibt auch die Möglichkeit zu testen, ob ein Knoten vor einem anderen Knoten ist (mittels <<).

Vergleich

Atomare werte können mittels zweier Typen von Vergleichen verglichen werden.

- *Identität* `eq` und `ne` testen, ob zwei Werte eines gegebenen Typs gleich sind oder nicht.
- *Größe* `lt`, `le`, `gt` und `ge` testen, ob für zwei Werte a und b eines gegebenen Typs $a < b$, $a \leq b$, $a > b$ oder $a \geq b$.

Nichte alle Typen können angeordnet werden. So gibt es keinen Größenvergleich zwischen Werten für `xs:duration`. Für `xs:booean` gilt, dass `false` kleiner ist als `true`. Sinnvoll sind deswegen `3 lt 4`, `true lg false`, `2017-08-02 ge 2016-03-31`.

Siehe das W3 Dokument zu comparisons.

Boolesche Ausdrücke

Es gibt die üblichen drei Ausdrücke: `not`, `and` und `or`. Wie üblich bindet Konjunktion stärker als Disjunktion.

1.5.3 Arithmetik

XPath erlaubt auch zu rechnen. Die Grundfunktionen sind dabei relativ eingeschränkt. Wir haben die Grundrechenarten, also +, -, * und `div`. Ferner gibt es `idiv` (Division von ganzen Zahlen). Dazu kommen die numerischen Vergleiche. Interessant in diesem Zusammenhang ist die Rundung. XPath stellt zwei Funktionen zur Verfügung: `round`, welche eine ganze Zahl rundet, sowie `round-half-to-even`. Diese Funktion ist wie folgt. Das erste Argument ist die zu rundende Zahl, das zweite gibt die Nachkommastellen an. Wir können mit dieser Funktion zum Beispiel auf 2 oder 3 oder eine andere Zahl von Nachkommastellen runden. Dabei

gibt es aber eine Spezialität: anders als bei der normalen Rundung wird im Fall, dass die zu rundende Zahl genau auf der Hälfte liegt, nicht automatisch nach oben gerundet. Sondern es wird zu derjenigen Zahl gerundet, bei der die letzte Ziffer gerade ist. So ist `round-half-to-even(3.1475,3)` deswegen 3.148, aber `round-half-to-even(3.1465,3)` ist 3.146. Diese Funktion hat den Vorteil, dass sie die Statistik nicht verschiebt (insbesondere den Mittelwert). Die Standardrundung, so die Idee, wird nur dann benötigt, wenn man eine Ausgabe haben will. Ansonsten soll man der anderen Funktion den Vorzug geben.

Es gibt ein paar nützliche arithmetische Funktionen auf Listen.

- `sum` benötigt als Argument eine Liste und gibt dafür die Summe der Elemente der Liste zurück.
- `count` benötigt als Argument eine Liste und gibt dafür die Anzahl der Elemente der Liste zurück.
- `avg` benötigt als Argument eine Liste und gibt dafür den Durchschnitt der Elemente der Liste zurück.

Es ist also `avg(t)` dasselbe wie `sum(t)/count(t)`. Damit lassen sich schon jetzt mächtige Anwendungen schreiben. Eine besonders beliebte ist das Scheckbuch. Hier sind die Einträge stets irgendwelche Kontobewegungen, und wir können mit dem Befehl `sum` jeweils Bilanz machen. Über die Listenfilter können wir auch aus den Kontobewegungen bestimmte Elemente herausfiltern und nur diese zusammenrechnen. Hier ist ein Beispiel.

```
<xsl:value-of select="avg(/kurs/student[1 to
                                        10]/note)">   (1.78)
```

Dies bildet den Durchschnitt der Noten bei den ersten zehn Studenten.

Zusätzlich gibt es noch die Rekursion, die ich allerdings erst später besprechen werde. Zusammen mit rekursiv definierten Funktionen kann man letztlich mit Hilfe von XSLT Tabellenkalkulation jeder Art vornehmen.

1.5.4 Zeichenketten

XPath stellt auch Funktionen zur Manipulation von Zeichenketten zur Verfügung.

- `contains`, um zu sehen, ob eine Zeichenkette in einer anderen Zeichenkette vorkommt (so ist etwa `contains("Beethoven", "Beet")` wahr, aber `contains("Beethoven", "Beat")` falsch).

- `substring`, um aus einer Zeichenkette eine andere Zeichenkette zu extrahieren und zwar durch Angabe der Startposition und der Länge. ZB ergibt

  ```
  substring("Wertzeichen", 3, 5)
  ```

  die Kette `"rtzei"`. Sollte der Anfangswert größer sein als die Länge der Kette, so ist der Wert `""`. So ergibt etwa

  ```
  substring("Wertzeichen", 20, 5)
  ```

  den Wert `""`. Werden mehr Zeichen gefordert, als vorhanden, so egibt die Funktion die höchstmögliche Kette; also ergibt

  ```
  substring("Wertzeichen", 10, 5)
  ```

  lediglich `"en"`.

- `substring-after`, um in einer Zeichenkette den Rest herauszusuchen, der hinter der gegeben Zeichenkette steht. Dabei wird das erste mögliche Vorkommen ausgewählt. Sollte es kein Vorkommen geben, bekommen wir die leere Zeichenkette. So ist etwa

  ```
  substring-after("abcdef", "c")
  ```

  gleich `"def"`,

  ```
  substring-after("abcdcef", "c")
  ```

  gleich `"dcef"`, `substring-after("abcdef", "g")` gleich `""`.

- `substring-before`, um in einer Zeichenkette den Rest herauszusuchen, der vor der gegeben Zeichenkette steht, genauer ihrem ersten Vorkommen. Wiederum ist das Ergebnis leer, wenn kein Vorkommen existiert. Etwa ist

  ```
  substring-before("abcdef", "c")
  ```

  genau `"ab"`.

Man beachte, dass das erste Vorkommen des leeren Worts bereits am Anfang einer Kette ist. Also ist

```
substring-after("abcdef", "")
```

gleich `"abcdef"`, und

```
substring-before("abcdef", "")
```

gleich `""`.

1.5.5 Listen

> [Listen in XPath]
> In XPath sind Listen grundsätzlich flach. Die Listenkonkatenation wird mit einem Komma notiert. So ist `1, 2, 3` die Liste bestehend aus den Elementen `1`, `2` und `3`. Eine Gruppierung in `(1, 2), 3` bezeichnet dieselbe Liste. Eine Liste von Listen von Zeichenketten ist also wieder eine Liste von Zeichenketten. Ein einziges Element ist identisch mit der Liste der Länge 1 bestehend aus diesem Element.

Auf der einen Seite ist nun eine Liste der Länge 1 nicht unterscheidbar von dem Element, auf der anderen Seite muss man gelegentlich doch aufpassen. Die Funktion `max` zum Beispiel erwartet ein einziges Argument. Es ist daher unzulässig zu schreiben `max(1,2,5)`. Die Klammern sind lediglich anwesend, um die Argumente der Funktion sichtbar zu machen. Innerhalb des Funktionsausfrufs dient nun das Komma dazu, die Argumente voneinander zu trennen. Deswegen ist es nötig, noch einmal Klammern zu setzen, damit klar ist, dass es sich nicht um drei Argumente handelt sondern um ein einziges, eine Liste:

```
max((1, 2, 5))
```
(1.79)

Es gibt drei Prädikate, mit denen man Listen überprüfen kann.

- `exactly-one(Liste)` : ergibt `Liste`, falls `Liste` genau ein Element enthält, ansonsten bekommen wir eine Fehlermeldung.
- `one-or-more(Liste)` : ergibt `Liste`, falls `Liste` mindestens ein Element enthält, ansonsten bekommen wir eine Fehlermeldung.
- `zero-or-one(Liste)` : ergibt `Liste`, falls `Liste` höchstens ein Element enthält, ansonsten bekommen wir eine Fehlermeldung.

Normalerweise benötigt man diese aber nicht (man hat ja auch noch die Funktion `count`, siehe weiter unten). Allerdings geben diese nicht einen Wahrheitswert aus, sondern die Liste selbst. Ganz anders die Funktion `empty(L)`, welche wahr ist, wenn die Liste leer ist, und ansonsten falsch. Der Wert dieser Funktion ist vom Typ `xs:boolean`.

Erzeugen von Listen

Es gibt mehrere Arten, Listen zu erzeugen. Aus Dokumenten kann man sich mit Hilfe von Pfadausdrücken Listen von Knoten besorgen. Man kann eine Liste auch mit dem Komma hinschreiben: `Katze, Hund, Maus`. Dabei kann man auch Klammern verwenden:

```
(Katze, Hund, Maus)
```
(1.80)

Gruppierungen sind möglich aber technisch irrelevant. Für ganze Zahlen gibt es noch den Konstruktor `to`. Dieser hat folgende Syntax: $\texttt{Zahl}_1$ `to` $\texttt{Zahl}_2$, wo $\texttt{Zahl}_1$ und $\texttt{Zahl}_2$ ganze Zahlen bezeichnen. Das Ergebnis ist die Liste aller ganzen Zahlen, aufsteigend geordnet, beginnend mit $\texttt{Zahl}_1$ und endend mit $\texttt{Zahl}_2$. Ist $\texttt{Zahl}_2$ kleiner als $\texttt{Zahl}_1$, so ist die Liste leer; sind die Zahlen gleich, so erhält man eine Liste mit einer einzigen Zahl. Es ist also `1 to 5` nichts anderes als

```
1, 2, 3, 4, 5
```
(1.81)

An dieser Stelle sei noch erwähnt, dass es in XPath Variable gibt. Jede Zeichenkette kann als Variable gelten. Sie muss nur obligatorisch mit `$` beginnen. So ist zum Beispiel `$wert` eine Variable. Dann kann man schreiben

```
1 to $wert
```
(1.82)

Es gibt zahlreiche Funktionen auf Listen, von denen ich ein paar vorstellen möchte.

- `count({Liste})` : Gibt die Anzahl der Elemente der Liste.
- `max({Liste})` : Gibt das größte Element der Liste.
- `min({Liste})` : Gibt das kleinste Element der Liste.
- `sum({Liste})` : Berechnet die Summe über alle Elemente der Liste.
- `avg({Liste})` : Berechnet den Durchschnitt über alle Elemente der Liste.

Michael Kay bespricht eine interessante Möglichkeit, sich den Listenplatz anzuzeigen. XSLT stellt dazu eine Funktion namens `xsl:number` bereit. So können wir uns über den Befehl `<xsl:number/>` unmittelbar anzeigen lassen, der wievielte Teilnehmerknoten wir sind. Alternativ dazu kann man schreiben

```
count(preceding-sibling::teilnehmer)+1        (1.83)
```

Was man hier tut, ist, sich die Liste aller vorangegangenen Knoten zu besorgen, daraus alle `teilnehmer` -Knoten auszusuchen (was man nicht muss, wenn man sicher ist, dass es nur solche gibt, also könnten wir da auch `node()` schreiben) und schließlich 1 hinzufügen, weil wir ja schon an der nächsten Position stehen.

1.5.6 Iteratoren

XPath 2.0 erlaubt, eine beliebige Funktion auf die Element einer Liste anzuwenden. Ist `{Funktion}` eine Funktion und

$$\texttt{\{Liste\}} = \texttt{\{Element}_1\texttt{\}},\cdots,\texttt{\{Element}_n\texttt{\}} \tag{1.84}$$

eine Liste, so bekommt man mit

```
for $i in {Liste} return {Funktion}($i)       (1.85)
```

die Liste

$$\texttt{\{Funktion\}(\{Element}_1\texttt{\}), \{Funktion\}(\{Element}_2\texttt{\}),} \cdots,\texttt{\{Funktion\}(\{Element}_n\texttt{\})} \tag{1.86}$$

Hier ist ein Beispiel.

```
for $i in 1 to 5 return $i * $i               (1.87)
```

Dies ergibt die Liste

```
1, 4, 9, 16, 25                               (1.88)
```

Die anzuwendende Funktion kann irgendwas sein, sie darf auch Listen auswerfen. In diesem Fall werden aber die Listen aneinandergereiht, und es entsteht eine große Liste:

```
for $i in 0 to 4 return 1 to $i               (1.89)
```

Dies erzeugt folgende Liste.

```
1, 1, 2, 1, 2, 3, 1, 2, 3, 4
```
(1.90)

Man mache sich klar, wie das zustandegekommen ist. Für `$i`=0 bekommt man die leere Liste (der Ausdruck `1 to 0` ist nach dem oben Vereinbarten leer). Für `$i`=1 bekommt man `1`, für `$i`=2 die Liste `1,2`, und so weiter. Mit Hilfe von Klammern sieht das etwa so aus:

```
(), (1), (1, 2), (1, 2, 3), (1, 2, 3, 4)
```
(1.91)

Wichtig! `for`-Ausdrücke versetzen nicht den Fokus. Dazu ein Beispiel. Wir wollen die Klausurergebnisse der Teilnehmer in eine Liste werfen. Der Fokus ist im Knoten `kurs`, welcher unter sich alle Teilnehmerknoten beherbergt. Das Folgende tut nicht das Gewünschte unter der Annahme, dass die `klausur` Knoten jeweils unter dem `teilnehmer` Knoten sitzen.

```
for $i in teilnehmer return klausur
```
(1.92)

Richtig ist vielmehr

```
for $i in teilnehmer return $i/klausur
```
(1.93)

Um zu verstehen, wie der Code funktioniert, noch einmal eine genaue Erklärung. Das Wort `teilnehmer` ist ein Pfadausdruck und wirft eine Liste von Knoten aus, etwa {Knoten$_1$}, {Knoten$_2$},$\cdots$, {Knoten$_n$}. Für diese wird nun der Ausdruck berechnet (Fokus ist gleichgeblieben). Hier kommt nun wieder ein Pfadausdruck, aber dieser ändert sich ständig, weil er ja von dem gewählten Knoten abhängt. Deswegen steht hier die Variable `$i`. Sie gibt uns den Unterknoten, dh sie erlaubt uns, den Fokus zu versetzen. Man sehe sich genau den Gebrauch von Variablen in Pfadausdrücken an.

1.5.7 Quantoren

XPath erlaubt auch, zu quantifizieren. Die Syntax ist

```
some {Variable} in {Liste} satisfies
                                {Boolescher Ausdruck}
```
(1.94)

`{Boolescher Ausdruck}` ist ein Ausdruck, der für die Listenelemente jeweils einen Wahrheitswert ergeben muss. Hierbei wird `{Variable}` in `{Boolescher`

`Ausdruck}` vorkommen, muss es aber nicht. Zum Beispie können wir herausfinden, ob jemand an der Klausur nicht teilgenommen hat:

```
some $i in teilnehmer satisfies $i/klausur eq ''
```
(1.95)

Es wäre in diesem Fall auch möglich, Folgendes zu nehmen:

```
teilnehmer[klausur = '']
```
(1.96)

Dies würde uns eine Liste geben. Diese kann man aber in einen Wahrheitswert umrechnen: Die leere Liste ist falsch, alles andere ist wahr. Das kann man zum Beispiel mit der Funktion `exists` (siehe unten). Damit kommen wir auf das gleiche Ergebnis.

Ebenso gibt es noch den Allquantor

```
every {Variable} in {Liste} satisfies
                                {Boolescher Ausdruck}
```
(1.97)

Es gibt für beide Quantoren eine Kurzform. Und zwar kann man mehrere gleichartige Quantoren zusammenziehen. Anstelle von

```
every $i in L satisfies (every $j in M satisfies B)
```
(1.98)

darf man schreiben

```
every $i in L, $j in M satisfies B
```
(1.99)

Es gibt noch eine weitere Kurzschreibweise. Nämlich mit der Funktion `exists`. Diese braucht als einziges Argument eine Liste und gibt *wahr* aus, wenn diese Liste nicht leer ist.

Es gelten die Gesetze der Logik. Es ist *alle A sind B* äquivalent mit *nicht*: *es existiert A, das nicht B*. Deswegen kann man den Allquantor wie folgt ersetzen:

```
not (some $i in L satisfies not (B))
```
(1.100)

Wichtig! Genau wie `for` versetzen die Quantoren den Fokus *nicht*.

1.5.8 Boolesche Verbindungen

An dieser Stelle sollte ich noch einmal auf die booleschen Junktoren zu sprechen kommen. XPath kennt außer den Quantoren und Iteratoren noch die Junktoren `if·then` (mit optionalem `else`), `and` und `or`. Diese haben die übliche Bindungsstärke: `and` bindet stärker als `or`. Zu den Ausdrücken, die booleschwertig sind gehören numerische Vergleiche, Knotenvergleiche (zB Abfragen, ob der Wert eines Knotens eine bestimmte Zahl ist) und so weiter. Pfadausdrücke sind jedoch von anderer Natur! Ihr Wert ist ja eine Liste von Knoten, deswegen sind sie nicht als boolesche Werte zu betrachten. Hier muss man stattdessen die folgenden Operatoren wählen:

- `union` oder `|` für *Vereinigung*,
- `intersect` für *Schnitt*,
- `except` für die *Differenz*.

Man beachte, dass Mengen eigentlich Listen ohne Duplikate sind. In der Tat werden bei der Vereinigung Duplikate eliminiert.

1.6 Verschiedenes

1.6.1 Variable und Parameter

Ein Variable wird wie folgt vereinbart.

```
<xsl:variable name = {QName}
    select? = XPath-Ausdruck
    as? = {Typ}>                                    (1.101)
    <!-- {Wert} -->
</xsl:variable>
```

Der Name ist der Name der Variablen. Dies hat den Typ eines QName, das heißt ein *qualified name*. Für unsere Zwecke reicht es zu sagen, dass dies eine Zeichenkette ist. Haben wir eine Variable namens `theo` definiert, so bekommen wir den Wert der Variablen durch `$theo`.

Das Attribut `as` erlaubt, den Typ zu spezifizieren. Man kann Variable von jedem Typ haben; der Default ist allerdings `string`. Man ist gut beraten, den Typ anzugeben.

Der Wert, der der Variablen zugewiesen wird, ist entweder der Wert des in dem Tag eingeschlossenen Knotens, also zB

```
<xsl:variable
    as ="xs:int">
    137
</xsl:variable>
```
(1.102)

Oder der Wert bestimmt sich durch Auswertung des `select` Ausdrucks. Es darf deswegen nicht gleichzeitig ein `select` Attribut vorhanden sein und der Knoten nicht leer. Denn dann gibt es einen Konflikt zwischen zwei Zuweisungen. Allerdings dürfen beide leer sein. In diesem Fall hat die Variable den Wert einer leeren Zeichenkette.

Es kommt häufig bei Zuweisung von Variablen, wie sie in (1.102) definiert sind zu einem Problem: die Leerzeichen vor und nach dem Wert werden nicht völlig ignoriert, wenn es sich nicht um einen Typ handelt, bei dem dies egal ist, wie etwa Zahlen. Sie werden auf ein einziges Leerzeichen gekürzt. Dieses ist nicht immer erwünscht. In diesem Fall können wir mit den Toplevel Anweisungen `<xsl:preserve-space elements={ Liste }>` und `<xsl:strip-space elements={ Liste }>` für Elemente gewissen Namens von vornherein festelegen, dass Weißzeichen wichtig bzw. unwichtig sind.

Man kann auch in XPath die Funktion `normalize-space()` aufrufen. Dies ist immer eine gute Idee. Diese Funktion eliminiert alle Leerzeichen vor und nach dem String. Stellen wir uns folgenden Ausschnitt in der XML-Datei vor.

```
<kundennummer>
    7841329
</kundennummer>
```
(1.103)

Wir weisen nun einer Variablen einen Wert zu

```
<xsl:variable name="kdr" select="kundennummer">
```
(1.104)

Dann führt der folgende Test zu einem negativen Ergebnis.

```
$kdr = '7841329'
```
(1.105)

Denn die Variable enthält außer der Zeichenkette noch einen Zeilenumbruch und Leerzeichen. Um dies zu vermeiden, sollte die Zuweisung wie folgt aussehen.

```
<xsl:variable name="kdr"
    select="normalize-space(kundennummer)">
```
(1.106)

Ich gebe hier ein instruktives Beispiel. Um die im Prozessor verankerten Alphabetreihenfolgen zu tabellieren, habe ich eine Datei geschrieben, die einfach nur die Buchstaben in irgendeiner Reihenfolge enthält, etwa so.

```
<?xml version="1.0">
<alphabet>
    <wort>c</wort>
    <wort>b</wort>
    <wort>q</wort>
</alphabet>
```
(1.107)

Das eigentlich Spannende ist die Transformationsdatei.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="2.0">
    <xsl:variable name="orig"
        select="/alphabet/wort" as="xs:string*"/>
    <xsl:variable name="ger"
        as="xs:string*">
        <xsl:perform-sort select="$orig">
            <xsl:sort lang="de-DE"/>
        </xsl:perform-sort>
    </xsl:variable>
    <xsl:variable name="hun" as="xs:string*">
        <xsl:perform-sort select="$orig">
            <xsl:sort lang="hu-HU"/>
        </xsl:perform-sort>
    </xsl:variable>
    <xsl:template match="/alphabet">
        <html>
            <head/>
            <body bgcolor="moccasin">
                <h3>Das Alphabet</h3>
                <table border="3">
                    <tr><td>Pos.</td><td>Original</td>
                    <td>de-De</td><td>hu-HU</td></tr>
```

```
                    <xsl:for-each select="$orig">
                        <xsl:variable name="i" select="position()"/>
                        <tr>
                          <td>
                            <xsl:number value="$i" format="1."/>
                          </td>
                          <td>
                            <xsl:value-of select="$orig[$i]"/>
                          </td>
                          <td>
                            <xsl:value-of select="$ger[$i]"/>
                          </td>
                          <td>
                            <xsl:value-of select="$hun[$i]"/>
                          </td>
                        </tr>
                    </xsl:for-each>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Die Testdatei ordnet auch nach finnischer Alphabetordnung. Ich habe das mit Saxon 9.1.0.5 getestet und es hat funktioniert. Als Ergebnis bekommt man die Buchstaben zunächst unsortiert, dann nach verschiedenen Länderkonventionen sortiert ausgegeben. Der Clou ist, dass die Ausgabe senkrecht erfolgt. Damit das geht, habe ich von `<xsl:perform-sort>` Gebrauch gemacht. Dies erlaubt, eine Liste zu ordnen. Das Ergebnis ist wieder eine Liste, und diese weise ich einer Variablen zu. Das Attribut `as` mit Wert `xs:string*` darf hier nicht fehlen, sonst denkt das Programm, ich will nur eine Zeichenkette. (Der Stern * ist ein sogenannter Typkonstruktor. `{Typ}*` bezeichnet Listen von Objekten vom Typ `Typ`.)

Anschließend gehe ich synchron durch alle Listen durch. Dabei wird eine Variable `i` lokal vereinbart und mit dem Wert `position()` versehen. Der Wert wird dann ausgegeben und anschließend jeder Liste witergegeben. (Auch hier wieder: ohne die Variable geht's nicht, das habe ich probiert.) Man kann auch noch die Möglichkeiten einer Formatierung von Zahlausgaben anschauen.

Hat man eine Variable, sagen wir `x7`, deklariert und ihr einen Wert zugewiesen, etwa 123, so kann man auf diesen Wert anschließend zurückgreifen. Den Wert bekommt man mittels `$x7`. Klarerweise ist der Wert der Variable nicht verfügbar, bevor sie definiert wurde (es sei denn, sie ist global). Jedoch ist die Variable nicht unbegrenzt danach existent sondern besitzt eine gewisse Lebensspanne, den sogenannten *Skopus*. In XSLT ist der Skopus einer Variable durch das Tag begrenzt, in dem die Variable definiert worden ist. Das bedeutet: ist die Variable unmittelbar nach einem Tag namens `<tag>` deklariert worden, dann endet der Skopus beim nächsten Auftreten von `</tag>`. Innerhalb ihres Skopus *darf* die Variable nicht erneut definiert werden, das ergibt eine Fehlermeldung.

In XSLT kann man den Wert einer Variablen nicht ändern, deswegen ist der Skopus ein wichtiges Instrument, um eine Variable tatsächlich variabel zu halten. Hier ist die Idee. Gesetzt, wir wollen eine Tabelle in HTML Format ausgeben. Wir wollen einen gewissen Wert, sagen wir die Zeilennummer, in einer Zeile verfügbar machen. Da sich die Zeilennummer ständig ändert, darf man die Variable nicht global vereinbaren. Denn dann wird ihr ein Wert zugewiesen, der nicht mehr zu ändern ist. Die Lösung besteht hier darin, die Variable erst *nach* dem Tag `<xsl:for-each>` zu vereinbaren. In diesem Fall wird ihr Wert für die gesamte Zeile (also bis zum nächsten `</xsl:for-each>`) zur Verfügung stehen. Man kann das in der oberen Datei sehr schön sehen. Innerhalb von `<xsl:for-each>` wird der Variable `i` der Wert von `position()` zugewiesen. Endet das Tag, dann wird ein neuer Knoten bearbeitet, die Variable `i` wird dann aber nicht neu zugewiesen: es ist lediglich so, dass mit Ende des ersten Zyklus die Variable `i` verschwindet und mit Beginn des neuen Zyklus eine neue Variable `i` entsteht. Ihr Wert ist dann der Wert von `position()`, welcher nunmehr um eins erhöht ist gegenüber dem vorigen Wert.

Parameter sind fast dasselbe wie Variable. Sie haben allerdings die Eigenschaft, dass sie ihre Werte anders bekommen (können). Es gibt drei Sorten Parameter. Der einfachste Typ ist der eines Funktionsparameters; ich bespreche ihn allerdings zusammen mit Funktionen. Stylesheet parameter sind Größen, die man dem Stylesheet zum Zeitpunkt des Aufrufs übergeben kann. Der Wert steht also nicht in dem Stylesheet sondern darf zum Zeitpunkt des Aufrufs festgelegt wer-

den. (oxygen erlaubt natürlich, diese Werte festzulegen.) Hier ist die Syntax.

```
<xsl:param
    name = {QName}
    select? = XPath-Ausdruck
    as? = {Typ}                                    (1.108)
    required? = "yes" | "no">
    <!-- {Wert} -->
</xsl:param>
```

Der einzige Unterschied zu <xsl:variable> ist die Möglichkeit festzulegen, ob der Parameter einen Wert haben *muss*. Es gelten wiederum die Regeln wie bei Variablen: es darf nicht gleichzeitig das Tag nichtleer sein und `select` anwesend. Allerdings dürfen diesmal beide leer sein, da der Wert ja noch nicht festliegen muss.

Bei Templates haben wir noch eine Option mehr:

```
<xsl:param
    name = {QName}
    select = XPath-Ausdruck
    as? = {Typ}
    required? = "yes" | "no"                       (1.109)
    tunnel? = "yes" | "no">
    <!-- {Wert} -->
</xsl:param>
```

Falls `tunnel` auf `yes` gesetzt wird, dann wird der Parameter an alle untergeordneten Prozeduren weitergereicht. Das ist sehr nützlich, wenn man eine Prozedur schreiben will, die verschiedene Stile der Ausgabe hat.

1.6.2 Modi

Modi dienen zur Auswahl von Templates. Wir können Templates zu gewissen Modi gruppieren um dann beim Aufruf <xsl:apply-templates> einen Modus anzugeben. Wenn wir also beim Aufruf den Modus spezifizieren, werden nur Templates verwendet, die zu diesem Modus gehören. Zusätzlich zu den frei verfügbaren Namen existieren drei spezielle "Modi":

- `#current`,

- `#default`,
- `#all`.

Die Befehle <xsl:apply-templates> und <xsl:template> besitzen je ein Attribut namens `mode`. Der Wert von `mode` in <xsl:template> darf eine Liste von Modi sein (darunter `#default`) oder `#all`. Der Wert von `mode` in <xsl:apply-templates> hingegen darf nur ein einziger Modus sein oder `#current`. Man beachte also, dass Templates in mehrere Modi gruppiert werden können, aber der Aufruf nur einen einzigen Modus verwendet. Man mag sich überlegen, dass es nicht nötig ist, beim Aufruf die Angabe von Listen von Modi anzugeben. Den Effekt kann man durch die Einführung eines neuen Modus erreichen, der alle in der Liste gegebenen Modi zusammenfasst. So zum Beispiel ist `#all` ein alles umfassender Modus.

Ist der Modus beim Template nicht gesetzt, wird der Wert `#default` eingesetzt. Das bedeutet, dass dieses Template immer dann verwendet wird, wenn in <xsl:apply-templates> *kein* Modus angegeben ist. Ist hingegen der Modus gesetzt und enthält er nicht `#default`, dann kann es nur dann werwendet werden, wenn <xsl:apply-templates> einen Modus aufruft, der in der Liste vorkommt. Sollte ein Template einen Modus haben, der niemals aufgerufen wird, so ist das kein Fehler. Es bedeutet lediglich, dass das Template niemals angewendet wird (eine gute Möglichkeit, um Templates zeitweilig zu inaktivieren, ohne den Code stark zu verändern).

Man sollte wissen, dass der Aufruf von <xsl:apply-templates> dazu führt, dass der XSLT-Prozessor versucht, ein geeignetes Template zu finden. Dabei gibt es mehrere Sonderfälle:

- der erste ist, wenn es mehr als ein Template gibt, das man anwenden kann. Dann wird mittels eines Verfahrens, das ich nicht schildern möchte, ein Kandidat ausgesucht. Ich empfehle nämlich schlicht, solch eine Situation zu vermeiden.
- der zweite Sonderfall ist, wenn es kein Template gibt. In diesem Fall geschieht nicht etwa nichts, sondern in Abhängigkeit vom Knoten bekommen wir folgendes Ergebnis.
  - *Dokument-* oder *Elementknoten*. <xsl:apply-templates> wird auf alle Kinder angewendet, Parameter werden nach unten weitergreicht.
  - *Text-* oder *Attributknoten*. Die Zeichenkette wird wiedergegeben (als wenn das Template wie folgt gewesen wäre: <xsl:value-of select="string(.)"/>

Auch hier gilt die Regel, dass man diesen Fall am besten vermeidet. Des öfteren wird man sich wundern, warum man Text ausgegeben bekommt, wo man keinen bestellt hat. Dies ist meist der Fall, weil man irgendwo <xsl:apply-templates> aufgerufen hat, aber der Prozessor bei diversen Tags nichts finden kann.

1.6.3 Sortieren und Gruppieren

Innerhalb von xsl:apply-templates und xsl:for-each ist es möglich, die Liste der Knoten zu sortieren. Dazu ist es nötig, unmittelbar nach dem Aufruf die Sortierordnung festzulegen. Das geschieht mit xsl:sort.

```
<xsl:sort
    select? = XPath-Ausdruck
    lang? = { nmtoken }
    order? = { "ascending" | "descending" }
    collation? = { uri }
    stable? = { "yes" | "no" }                          (1.110)
    case-order? = { "upper-first" | "lower-first" }
    data-type? = { "text" | "number"
        | qname-but-not-ncname}>
    <!-- Inhalt: Folgen-Konstruktor -->
</xsl:sort>
```

Der Folgen-Konstruktor bestimmt, nach welchem Sortierschlüssel sortiert werden soll; alternativ dazu wird dieser durch das Attribut select angegeben. Wird nichts gesagt, wird die Eingabe wie select="." behandelt.

Hier ist ein Beispiel. Es sollen die Kunden aufgelistet werden und zwar mit dem Vornamen zuerst und dann dem Nachnamen. Dabei sollen sie so sortiert werden, dass logisch gesehen zuerst die Nachnamen sortiert werden, und erst wenn zwei Personen den gleichen Nachnamen haben, wird nach dem Vornamen sortiert.

(Es werden Personen mit dem gleichen Namen übrigens zweimal ausgegeben.)

```
<ul>
    <xsl:for-each select="./kunde">
        <xsl:sort lang="de" select="./name/nach"/>
        <xsl:sort lang="de" select="./name/vor"/>
        <li>
            <xsl:value-of select="./name/vor"/>          (1.111)
            <xsl:text> </xsl:text>
            <xsl:value-of select="./name/nach"/>
        </li>
    </xsl:for-each>
</ul>
```

Man beachte, dass wir *zwei* Sortierbefehle gegeben haben. Der erste legt den sogenannten Primärschlüssel fest; nach diesem wird vorrangig sortiert. Der zweite legt den Sekundärschlüssel fest. Nach diesem wird nachrangig sortiert; das bedeutet, dass er erst dann verwendet wird, wenn der Primärschlüssel keine Ordnung festlegt. So sind Personen mit dem gleichen Nachnamen nicht wechselseitig nach dem Primärschlüssel geordnet. In diesem Fall legen wir mit dem Sekundärschlüssel fest, dass der Vorname entscheidet. Es dürfen beliebig viele Sortierbefehle hintereinandergesetzt werden.

Es ist möglich, aufsteigend wie absteigend zu sortieren; dafür dient das Attribut `order`. Ferner kann die Sortierung der Sprache angepasst werden. Dazu gibt es das Attribut `lang`. Siehe dazu die Erläuterungen auf Seite 28. Ich habe oben bereits angedeutet, dass Sprachen verschiedene alphabetische Reihenfolgen festlegen. Im Englischen werden Umlaute an das Ende gesetzt; dort ist „ä" hinter „z", zumindest sieht die Implementierung das so vor. Deswegen ist es unbedingt erforderlich, dass man die Sprache auf `de` beziehungsweise `de-DE` festlegt. Das Attribut `collation` hat als Wert eine Internetaddresse (URI), die festlegt, nach welchen Regeln vorgegangen werden soll. Kollationen bestimmen, welche Zeichenfolgen als identisch angesehen werden sollen (etwa „ä" und „ae"). Für unsere Zwecke ist dies nicht weiter wichtig.

Falls wir die Kunden jedoch nach der Kundennummer sortieren wollen, so

sieht die Transformationsdatei wie folgt aus.

```
<ul>
    <xsl:for-each select="./kunde">
        <xsl:sort select="./kundennumer"/>
        <li>
            <xsl:value-of select="./name/vor"/>
            <xsl:text> </xsl:text>
            <xsl:value-of select="./name/nach"/>
        </li>
    </xsl:for-each>
</ul>
```
(1.112)

Spannend wird es, wenn wir eine Liste erstellen wollen, die auf eine gewisse Weise geordnet ist, ohne sie gleich zu verarbeiten. In diesem Fall gibt es das Element `<xsl:perform-sort>`.

```
<xsl:variable name="vergeben">
    <xsl:perform-sort select="./kunde/kundennummer">
        <xsl:sort select="."/>
    </xsl:perform-sort>
</xsl:variable>
```
(1.113)

Wen es stört, dass Zahlen doppelt auftreten, der kann die Funktion `distinct-values()` aufrufen. Diese braucht ein einziges Argument, welches eine Liste ist, und eliminiert mehrfache Vorkommen. Diese kann eine Liste von Knoten bekommen, sie liefert allerdings lediglich eine Liste von Werten ab, weswegen der Gebrauch etwas trickreich sein kann. Ich biete hier die folgende Variante an: `$vergeben` ist bereits eine Liste, wir müssen nur noch die Funktion darauf loslassen und einer Variable übergeben, die dann, wohlgemerkt, eine Liste von Werten ist.

```
<xsl:variable name="vergeben2"
    select="distinct-values($vergeben)"/>
```
(1.114)

Die Gruppierung von Elementen nach gewissen Schlüsseln ist eine weitere Mög-

lichkeit in XSLT.

```
<xsl:for-each-group
    select = { XPath-Ausdruck }
    group-by? = { XPath-Ausdruck }
    group-adjacent? = { XPath-Ausdruck }
    group-starting-with? = { Pattern }                    (1.115)
    group-ending-with? = { Pattern }
    collation? = { uri }>
    <!-- Inhalt: (xsl:sort*, Folgen-Konstruktor) -->
</xsl:for-each-group>
```

Das `select` Attribut definiert die Knotenmenge, die gruppiert werden soll (im Jargon Population genannt). Diese wird durch den Befehl in Gruppen geteilt und dann gruppenweise bearbeitet. Ein anschließender `xsl:sort`-Befehl sortiert die Gruppen untereinander. Wird kein solcher Befehl gegeben, gilt die Sortierung nach den ersten Elementen in jeder Gruppe. Der anschließende Folgen-Konstruktor legt fest, was mit jeder Gruppe (nicht: Gruppenelement) geschehen soll.

Die folgenden Attribute schließen sich gegenseitig aus:

- `group-by`,
- `group-adjacent`,
- `group-starting-with`
- `group-ending-with`

Ich diskutiere nur `group-by`. Der Wert von `group-by` legt fest, welche Elemente in eine gemeinsame Gruppe kommen. Dieser Ausdruck wird für jedes Element ausgewertet; eine Gruppe besteht jeweils aus allen Elementen, bei denen dieser Ausdruck denselben Wert besitzt. Zum Beispiel kann dies die Note in einer Klausur sein oder der Geburtstag. Die Prozedur wird nun für jede Gruppe getrennt ausgeführt.

Schauen wir uns das Beispiel aus Dokument des WWW-Konsortiums an.

```
<staedte>
  <stadt name="Milano" land="Italien"
      bev="5"/>
  <stadt name="Paris" land="Frankreich"
      bev="7"/>
  <stadt name="München" land="Deutschland"
      bev="4"/>
  <stadt name="Lyon" land="Frankreich"
      bev="2"/>
  <stadt name="Venezia" land="Italien"
      bev="1"/>
</staedte>
```
(1.116)

Die Vorgabe ist, die Städte nach ihren Ländern zu gruppieren.

```
<table xsl:version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <tr>
    <th>Position</th>
      <th>Land</th>
    <th>Städteliste List</th>
    <th>Bevölkerung</th>
  </tr>
  <xsl:for-each-group select="staedte/stadt"
    group-by="@land">
    <tr>
      <td><xsl:value-of select="position()"/></td>
      <td><xsl:value-of select="@land"/></td>
      <td>
        <xsl:value-of select="current-group()/@name"
          separator=", "/>
      </td>
      <td><xsl:value-of
        select="sum(current-group()/@bev)"/></td>
    </tr>
  </xsl:for-each-group>
</table>
```
(1.117)

Die Anordnung der Gruppen untereinander wird durch `xsl:sort` unmittelbar im Anschluss an `xsl:for-each-group` festgelegt, genau wie dies bei `xsl:for-each` geschieht. Die Sortierung der Elemente innerhalb einer Gruppe muss getrennt festgelegt werden. Gibt man kein `xsl:sort` an, so werden die Gruppen in der Dokumentreihenfolge ausgegeben; diese wird aus der relativen Ordnung der ersten Elemente jeder Gruppe bestimmt. Die Gruppe wird als Liste ausgegeben, in der Dokumentordnung, wobei lediglich ein Trennzeichen festgelegt ist.

Es gibt es noch sehr viel mehr Möglichkeiten. Ich möchte hier nur darauf hinweisen, dass man auch Gruppen weiter in Gruppen unterteilen kann. Weitere Beispiele finden sich in dem Dokument des W3-Konsortiums.

1.6.4 Ein- und Ausgabe

Bisher haben wir nur Fälle angeschaut, wo XSLT aus einer einzelnen Eingabedatei eine einzige Ausgabedatei erzeugt. Gewiss ist es möglich, mit Hilfe von Modi oder Parametern die Struktur und Art der Ausgabe zu steuern, aber in einem einzigen Lauf konnten wir bisher nur eine einzige Datei erzeugen und auch nur eine einzige Datei einlesen. Beides sind Einschränkungen, die man früher oder später aufheben möchte.

Mehrfache Ausgabe In XSLT gibt es das Element <`xsl:result-document`>. Es gibt dazu viele Optionen, von denen ich nur einige bespreche.

- `format` spezifiziert das Ausgabeformat.
- `href` spezifiziert die Ausgabedatei.
- `method` spezifiziert die Ausgabemethode (`html`, `xhtml`, `text`, `xml`).
- `version` gibt die Versionsnummer des Ausgabeformats an (wie auch das `version` Attribut von <`xsl:output`>).

Man kann in <`xsl:result-document`> die Spezifikation eines gesamten Ausgabedokuments packen. Dies wird dann in die Datei ausgegeben, die mit Hilfe von `href` benannt wurde. Dabei kann man so viele Dateien schaffen, wie man möchte. Zum anderen kann man Dateinamen auch erzeugen, weil ja `href` auch Spezifikationen mit Hilfe von Variablen erlaubt; daz umüssen wir nur Ausdrücke wie `{$Zahl}` einbauen, wo `Zahl` eine Variable ist, die natürlich vorher deklariert worden ist.

Mehrfache Eingabe Die Eingabe einer weiteren Datei erfolgt nicht, wie man jetzt erwarten würde, über einen analogen Befehl in XSLT. Sondern es existiert eine Funktion `document()`, mit deren Hilfe man eine Datei einlesen kann. Zum Beispiel hat das Kommando `document('data.xml')` den Effekt, dass die Datei mit Namen `data.xml` einlesen wird. Der Baum steht dann zur Verarbeitung zur Verfügung. Die Adressierung erfolgt dann wie folgt: Die Adresse des Knotens erhält man durch Voranstellen des Befehls `document('...')`. Nur das sichert nämlich, dass XSLT in dem angegebenen Dokument sucht und nicht in dem Baum des Hauptdokuments.

```
document('data.xml')/···
```
(1.118)

Man kann ihn zum Beispiel einer Variable zuweisen:

```
<xsl:variable name="baum"
    select="document('data.xml')"/>
```
(1.119)

In diesem Fall funktioniert die Adressierung von Knoten innerhalb der XML-Datei `data` genauso, wie sie bei XML-Strukturen auch funktioniert.

Wenn man Text „roh" einlesen will, also ohne den XML-Parser, dann kann man dies mit dem Kommando `unparsed-text(arg1?, arg2?)` tun. Das erste Argument ist eine URI, das heißt, sie benennt eine Internetressource. Das zweite spezifiziert die Kodierung (Default ist UTF-8). Hier ist ein Beispiel aus dem W3-Handbuch.

```
<xsl:output method="html"/>

<xsl:template match="/">
    <xsl:value-of select="unparsed-text('header.html',
        'iso-8859-1')"
        disable-output-escaping="yes"/>
    <xsl:apply-templates/>
    <xsl:value-of select="unparsed-text('footer.html',
        'iso-8859-1')"
        disable-output-escaping="yes"/>
</xsl:template>
```
(1.120)

1.6.5 Schlüssel

Schlüssel sind ein Instrument, um analog wie bei IDs, Assoziationen zwischen Elementen und Werten herzustellen. Nehmen wir an, wir haben ein Telefonbuch mit Einträgen der folgenden Form.

```
<eintrag>
    <name>
        <nach>Mustermann</nach>
        <vor>Erwin</vor>
    </name>
    <nummer>123456</nummer>
</eintrag>
```
(1.121)

Nehmen wir nun an, wir wollen jedem Eintrag die in ihm enthaltene Telefonnummer zuordnen. Der Befehl dazu lautet

```
<xsl:key name="telnr"
    match="//eintrag"
    use="nummer" />
```
(1.122)

Der Befehl <xsl:key> hat die folgende Syntax.

```
<xsl:key
    name="{Name}"
    match="{Muster}"
    use?="{Ausdruck}"/>
```
(1.123)

Nur die ersten beiden Attribute müssen vorhanden sein. Ist use nicht vorhanden, so muss stattdessen dem Tag xsl:key ein Wert gegeben werden, ähnlich wie bei Variablen. Effektiv sind also drei Objekte vorhanden.

- name bekommt als Wert eine Zeichenkette. Diese ist der *Name* des Schlüssels. Man kann ihn frei wählen.
- match bekommt als Wert ein Muster, welches eine Sorte von Knoten definiert. Diese ist zwar als Liste implementiert, wird aber als Menge behandelt (Mehrfachnennungen werden zB ignoriert).
- use bekommt als Wert einen Ausdruck. Dieser darf irgendein XPath-Ausdruck sein, der, gegeben einen Knoten, der das Muster matcht, einen oder mehrere Werte bekommt. Im Gegensatz zu IDs dürfen Schlüssel einem Knoten mehrere Werte zuordnen.

Wie benutzt man nun Schlüssel? Zunächst einmal ist die obige XSLT Deklaration nur dazu da, einen Schlüssel zu definieren. Hat man dies getan, muss man noch die XPath-Funktion `key(·, ·)` verwenden, um den Schlüssel zu verwenden. Hierbei ist das erste Argument der Name des Schlüssels, das zweite der Wert. `key(telnr,b)` ruft alle Knoten auf, die mittels des Schlüssels namens `telnr` dem Wert `b` zugeordnet worden sind. Hier ist ein Beispiel.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:key name="telnr" match="//eintrag"
     use="nummer"/>
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="key('telnr','123456')">
          <p>
            Nummer: <xsl:value-of select="nummer"/>
              <br />
            Name: <xsl:value-of select="name/nach"/>,
            <xsl:value-of select="name/vor"/>.
            </p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

(1.124)

Angewendet auf die Telefondatei bekommen wir damit eine Liste von Paaren, die jeweils die Telefonnummer 123456 sowie den dazu passenden Namen enthalten. Man beachte, dass der Schlüssel dazu dient, Knoten auszuwählen. Was wir davon ausgeben wollen, können wir frei bestimmen.

Wie nun können wir die Datei anwenden? Nehmen wir an, wir haben zwei Dateien erzeugt:

- `telbuch.xml`: enthält das Telefonverzeichnis.
- `suche-nummer.xsl`: enthält die Transformation.

Der Zeilenbefehl lautet dann (auf eine Zeile zu schreiben):

```
java -jar /usr/share/java/saxon9he.jar
    -xsl:suche-nummer.xsl
    -o:ergebnis.html telbuch.xml
```
(1.125)

Hierbei ist `/usr/share/java` der Pfad auf meinem Rechner zum Java-Archiv. Der kann also auf anderen Rechnern anders aussehen. Bitte erst prüfen! Ebenso ist `saxon9he.jar` die Version 9 von Saxon Home Edition (=HE).

In vielen praktischen Anwendungen möchte man aber das Ergebnis schnell per Zeilenbefehl bekommen und sich nicht vorher festlegen, nach welcher Telefonnummer man suchen will. Da bietet es sich an, die zu suchende Nummer als Parameter zu deklarieren:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:key name="telnr" match="//eintrag"
     use="nummer"/>
  <xsl:param name="tnr" required="yes" />
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="key('telnr', $tnr)">
          <p>
            Nummer: <xsl:value-of select="nummer"/>
              <br />
              Name: <xsl:value-of select="name/nach"/>,
            <xsl:value-of select="name/vor"/>.
          </p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
(1.126)

Ich habe eine Datei namens `schnellsuche.xsl` geschrieben, die dies mittels Eingabeparametern erledigt.

Hier ist der Zeilenbefehl für Xalan (auf eine Zeile zu schreiben):

```
xalan -in telbuch.xml -xsl schnellsuche.xsl
    -param tnr 123456 -out ergebnis.html
```
(1.127)

Und hier der Zeilenbefehl für Saxon. Bei Saxon müssen die Parameter am Ende stehen also hinter der XML-Datei (und hinter allen Saxon-Parametern), bei Xalan kommen sie vor die XML-Datei.

```
java -jar /usr/share/java/saxon9he.jar telbuch.xml
    -o:nummer.html schnellsuche.xsl
    tnr="123456"
```
(1.128)

1.6.6 Zeichenketten

Es soll hier nur kurz auf Möglichkeiten eingegangen werden, Zeichenketten zu bearbeiten. Dazu stehen unter anderem die Funktionen <xsl:analyze-string>, <xsl:matching-substring> und <xsl:non-matching-substring> bereit.

xsl:analyze-string (1.129)

Die Syntax ist

```
<xsl:analyze-string
    select = {Ausdruck}
    regex = {Zeichenkette}
    flags? = {Zeichenkette}>
    <!- {Inhalt:}
        xsl:matching-substring
        xsl:non-matching-substring
        xsl:fallback*->
</xsl:analyze-string>
```
(1.130)

Der Effekt ist wie folgt: Die in `select` genannten Knoten oder Ausdrücke werden ausgewertet und in eine Zeichenkette umgewandelt, auf die der reguläre Ausdruck angewendet wird. Der reguläre Ausdruck R wird durch `regex` angegeben. Die Zeichenkette zerfällt in Abschnitte, auf die R zutrifft und solche, auf die R nicht zutrifft. Diese liegen alternierend nebeneinander. (Dies wird dadurch hergestellt, dass das Programm die erste matchende Zeichenkette samt dem ihr vorausgehenden Teil abtrennt, und dann die Funktion auf den Rest anwendet (etwa wie bei

substring-after.) Für die Zeichenketten, auf die der reguläre Ausdruck zutrifft, wird der in <xsl:matching-substring> genannte Ausdruck eingesetzt, auf die anderen der in <xsl:non-matching-substring> gegebene Ausdruck. Dabei kann man in dem Flag-Ausdruck die Art des Matchens modifizieren. Der Wert von flags ist eine Zeichenkette, bestehend aus den Buchstaben i, m, s und x, in beliebiger Reihenfolge. Ist der jeweilige Buchstabe vorhanden, so gilt das Flag als gesetzt.

- i macht das Matching unsensibel gegenüber Groß- und Kleinschreibung
- m setzt den Zeilenmodus um. Ohne diese Option matcht ^ den Anfang der gesamten Kette, $ das Ende. Mit der Option matcht ^ den Zeilenanfang, $ das Zeilenende. (Dies ist wichtig bei Dateien, die explizite Zeilenendmarker enthalten.)
- s Ist diese Option gesetzt so matcht der Punkt auch den Zeilenumbruch (x0A). Ansonsten matcht er nur echte Zeichen.
- x sorgt dafür, dass leere Zeichen unwesentlich werden im regulären Ausdruck und deswegen als Trennsymbole gebraucht werden können.

Achtung: Der reguläre Ausdruck darf nicht durch die leere Zeichenkette gematcht werden können. So ist zum Beispiel [0-9]* ausgeschlossen (da es die leere Zeichenkette matcht), [0-9]+ ist dagegen erlaubt.

Achtung: Reguläre Ausdrücke dürfen auch dynamisch erzeugt werden. Deswegen müssen allerdings geschweifte Klammern gedoppelt werden. Zum Beispiel ist regex="[0-9]{{3}}" ein Ausdruck, der drei aufeinanderfolgende Ziffern matcht. regex="[0-9]{3}" ist dagegen nicht wohlgeformt.

Für die Einteilung in matchende und nicht-matchende Zeichenketten gilt folgende Vereinbarung.

- * und + sind *gierig*. Sie verlangen so viele Zeichen wie möglich, um konsistent mit dem Gesamtmatching zu sein. Somit matcht [0-9]+ nur die gesamte Ziffernfolge, nicht irgendwelche Teile davon. In [0-9]+[0-9] dagegen matcht die erste Gruppe lediglich die Ziffernfolge ohne die letzte Ziffer, sonst wäre ein Matching erst gar nicht möglich. Äquivalent dazu ist [0-9]+[0-9]+. Hier versucht die erste Gruppe, so viele Ziffern wie möglich auf Kosten der zweiten an sich zu reißen, sodass sie für die zweite Gruppe lediglich eine einzige Ziffer übriglässt.

- `*?` und `+?` sind *nicht gierig*. Sie verlangen so wenig Zeichen, wie möglich, um konsistent mit dem Gesamtmatching zu sein. Somit matcht `[0-9]+?` nur die erste Ziffer einer Ziffernfolge. Ebenso matcht in `[0-9]+?[0-9]+` die erste Gruppe lediglich eine einzige Ziffer, die zweite dagegen bekommt den Rest.

- Falls es Alternativen gibt, die an derselben Position matchen, so wird die erste genommen.

Atome

Atome sind Ausdrücke, die Einzelzeichen matchen.

- Alle Zeichen außer `.\?*+|^$()[]` matchen Einzelzeichen (wobei gelegentlich auch noch Sonderregeln gelten, wie für den Bindestrich in `[0-9]`)

- Eine in eckigen Klammern eingeschlossene Kette von Zeichen ist ein *Bereich*. Dieser kann eine einfache Aufählung sein, etwa `[axf123]`. In diesem Fall matcht die Gruppe eines der vorkommenden Zeichen (also `a`, `x`, `f`, oder die Ziffern `1`, `2` oder `2`), aber nur ein einzelnes Vorkommen davon; oder es ist ein Bereich wie etwa `[a-z]` (alle Kleinbuchstaben), `[A-Z]`) (alle Großbuchstaben), oder es ist eine Kombination davon: `[0-9X]` (alle Ziffern oder der Buchstabe `X`).

- `^` matcht den Zeilenanfang, `$` das Zeilenende (siehe dazu die Option `m` weiter oben).

- Der Punkt matcht ein Einzelzeichen (siehe dazu die Option `x` weiter oben).

- Es gibt eine große Zahl von sogenannten Zeichenklassen. Diese fassen (auch sprachabhängig) gewisse Zeichen zu Gruppen zusammen (alphabetisches Symbol, Interpunktion usf.)

Gruppen

Der reguläre Ausdruck wird durch runde Klammern (also `(` und `)`) in Gruppen unterteilt. Diese dürfen geschachtelt sein; diesen Fall werde ich nicht behandeln. Sind sie es nicht, werden sie von links nach rechts durchgezählt. Beim Matchen merkt sich der Algorithmus nicht nur den Wert des Gesamtausdrucks sondern auch den der Gruppen. Bei der Ersetzung von Zeichenketten kann man auf

Matching-Gruppen Bezug nehmen. Dazu stellt XSLT die Funktion `regex-group()` bereit. Diese erwartet als Argument eine Zahl *n* und gibt als Wert eine Zeichenkette, und zwar ist dies die Zeichenkette der *n*ten Gruppe im Matching, vorausgesetzt, es gibt für die Zahl *n* eine Klammergruppe.

Auch beim matchen selbst kann man auf Gruppen Bezug nehmen, und zwar durch `\\1`, `\\2` usf. Dabei bezeichnet also `\\1` die zuletzt gematchte Gruppe. Diese sind Rückwärtsreferenzen und bezeichnen die Klammergruppen aus dem gerade aktuell erstellten Match. Die Gruppen werden von links gehend aufgebaut und die Gruppen von 1 bis *n* liegen fest, wenn die Gruppe $n+1$ berechnet wird. So matcht der Ausdruck `(['"])\.*\\1` die Zeichenketten `'Hello'` und `"Hello"`, aber nicht die Ketten `'Hello"` und `"Hello'`, da bei ihnen die Kette, welche am Anfang `['"]` matcht, dies auch am Ende tun muss.

Beispiel

Das folgende Beispiel ist dem Buch von Michael Kay über XSLT 2.0 entnommen. Es geht um die Ausgabe des Datums in der Form 22^{nd} `March 2009`. (Achtung: dies ist für die englische Sprache entworfen, lässt sich also nicht ohne Weiteres ausprobieren. Wir müssen (falls es nicht der Default ist) `format-date()` noch ein weiteres Argument geben, nämlich `en-GB` eingeben.) Die Funktion `format-date()` erlaubt einem, die Ausgabe zumindest teilweise zu steuern: man kann die Zahlen ohne führende Nullen bekommen sowie als Ordinalzahlen (die Option `[D1o]` tut genau dies). Ebenso spezifiert `Nn`, dass der Name in Buchstaben ausgeschrieben wird, mit einem Großbuchstaben beginnend. Das Doppelkreuz ist das Trennzeichen. Das würde dann folgendes Ergebnis geben: `22nd March 2009`. Die Hochstellung des Ordinalsuffixes sowie der Schrägdruck ist jetzt unsere Sache, und hier

müssen wir Hand anlegen:

```
<xsl:analyze-string
        select="format-date(current-date(),
        '#[D1o]#[MNn]#[Y]')"
        regex="^#([0-9]+)([a-z]+)#([A-Z][a-z]+)#(.*)$">
    <xsl:matching-substring>
        <xsl:value-of select="regex-group(1)"/>
            <sup><i><xsl:value-of
                select="regex-group(2)"/></i></sup>
        <xsl:value-of select="regex-group(3)"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="regex-group(4)"/>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
        <xsl:value-of select="."/>
    </xsl:non-matching-substring>
</xsl:analyze-string>
```

(1.131)

Hier ist noch ein eigenes Beispiel. Das Formatieren von Datumsangaben kann manchmal trickreich sein. Insbesondere `format-dateTime()` ist der Transformationssoftware nicht immer bekannt, und auch das Sortieren ist, wie wir gesehen haben, etwas mühselig. Zum Glück kann man sich hier selber helfen. Man verabrede, dass Datum und Zeit kombiniert schlicht Zeichenketten sind, und zwar in der ohnehin in XSLT verwendeten Standardform: `YYYY-MM-DDTHH:mm:ss`. Dies erlaubt dann, den zeitlichen Vergleich von Datumsangaben schlicht über den lexikographischen Vergleich zu erledigen. Datum A ist genau dann zeitlich vor Datum B, wenn die Zeichenkette A lexikographisch vor der Zeichenkette B ist. Lediglich

die Ausgabe muss man noch selber machen.

```
<xsl:analyze-string
    select="."
    regex="([0-9]{{4}})-([0-9][0-9])-([0-9][0-9])
        T([:0-9]{{8}})">
    <xsl:matching-substring>
        <xsl:value-of select="concat(regex-group(3),
        '.', regex-group(2), '.',
        regex-group(1)', ' um ', regex-group(4))"/>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
        <xsl:value-of select="."/>
    </xsl:non-matching-substring>
</xsl:analyze-string>
```
(1.132)

Dies konvertiert `2012-06-12T11:34:07` zu `12.06.2012 um 11:34:07`.

1.6.7 Funktionen

Definition

XSLT 2.0 stellt die Möglichkeit bereit, Funktionen zu definieren. Funktionen müssen unterhalb des Stylesheet-Knotens definiert werden. Die Syntax ist wie folgt.

```
<xsl:function name={Funktionsname} as?={Typ}>
    <xsl:param name={Parameter_1}
        as?={Typ_1}>
    <xsl:param name={Parameter_2}
        as?={Typ_2}>
    usw.
    <xsl:param name=}Parameter_n"
        as?={Typ_n}>
    <!-- {Prozedur} -->
</xsl:function>
```
(1.133)

Man beachte, dass die Argumente der Funktion hier über `xsl:param` gegeben werden und nicht mit `xsl:variable`. Dies folgt der Logik von XSLT, derzufolge Variable sofort mit einem Wert versehen müssen, Parameter hingegen nicht. Da

die Argumente einer Funktion zum Zeitpunkt der Funktionsdeklaration keinen Wert haben *sollen*, müssen sie in XSLT Parameter genannt werden.

Die Typdeklaration ist, wie das Fragezeichen andeutet, nicht verpflichtend. Eine Funktion braucht aber ein Namensraum-Präfix. Ansonsten gibt es keine Beschränkungen. Ist der Name der Funktion etwa `x:fkt` (mit Namensraum `x`) und heißen die Parameter `p`, `q` und `r`, so haben wir die Funktion `x:fkt(p,q,r)` erklärt. Dabei darf der Name der Funktion im Rumpf der Definition wieder auftauchen. Das erlaubt, Funktionen rekursiv zu definieren. Hier ist ein Beispiel.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:w="http://wwwhomes.uni-bielefeld.de/mkracht"
  exclude-result-prefixes="xs"
  version="2.0">
  <xsl:function name="w:dreieck">
    <xsl:param name="i"/>
    <xsl:choose>
      <xsl:when test="$i=0">0</xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="sum(($i,
          w:dreieck($i -1 )))"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>
  <xsl:template match="/wert">
    <xsl:variable name="p" select="/wert"/>
    <xsl:value-of select="w:dreieck($p)"/>
  </xsl:template>
</xsl:stylesheet>
```
(1.134)

Ein paar Anmerkungen zu dieser Datei. Zunächst einmal enthält sie die Definition eines Namensraums. Denn eine Funktionsdeklaration muss in einem deklarierten Namensraum erfolgen. Ich habe dazu einfach den Namensraum `w` mit der URI `http://wwwhomes.uni-bielefeld.de/mkracht` genommen. Dies hat keinen besonderen Effekt außer den, dass kein unbeabsichtigter Namensraum genommen wird. Ansonsten befindet sich auf dieser Webseite nichts für XSLT Verwertbares.

Die Funktion berechnet zu der Eingabe für i die Summe aller Zahlen von 0 bis i. Dies erfolgt durch eine rekursive Definition. Ist $i = 0$, dann ist das Ergebnis 0, ansonsten ist es `dreieck`(i) = i + `dreieck`$(i - 1)$. (Das gilt nur für positive Zahlen.) In der XSLT Definition sieht das etwas anders aus. Dies hat zwei Gründe. Der erste ist, dass der Wert einer Variablen durch ein vorangestelltes `$` geholt wird. Der zweite ist, dass die Funktionsdefinition in XSLT erfolgt, aber der Ausdruck in XPath berechnet wird. Der Prozess springt also permanent zwischen XPath und XSLT hin und her.

Die Parameter sind die Argumente, von denen die Funktion abhängt. Diese müssen erklärt werden, damit klar ist, von welchen Zahlen die Funktion abhängen soll. Die Parameter besitzen außer dem Namen nur noch (optional) einen Typ.

Achtung Die Reihenfolge, in denen die Parameter deklariert werden, bestimmt die Reihenfolge, die sie in dem Funktionsaufruf haben. Hängt eine Funktion von drei Argumenten ab, so muss das erste Argument der Funktion zuerst deklariert werden, danach das zweite und erst zuletzt das dritte. (Denn bei der Definition der Funktion tauchen die Parameter nicht auf. Es ist also nicht klar, welcher der Parameter welchem Argument der Funktion zugeordnet werden muss. Deswegen die Konvention.)

Und hier ist die XML-Datei, auf die diese Transformation angewendet wird:

```
<?xml version="1.0" encoding="UTF-8"?>
<wert>10</wert>
```
(1.135)

Die Ausgabe ist eine relativ schmucklose Datei, die nur die Zahl 55 enthält. Aber das ist genau das, was wir erwartet haben.

Man kann Funktionen auch anstelle von `xsl:key` einsetzen, wobei der eigentliche Effekt, nämlich die Präkompilierung, dabei verlorengeht. Bei kleineren Dateien ist dies allerdings kein Problem. Um jetzt die Funktionalität von Schlüsseln einzusetzen, muss man jedoch eines beachten. Eine Funktionsdeklaration definiert keinen Kontext; der Fokus ist deswegen leer. Man kann also nicht mit Pfadausrücken arbeiten, weil diese *immer* in Bezug auf einen Kontext ausgewertet werden, auch wenn dieser nicht nötig wäre. Ein Ausweg ist, den Kontext explizit als Argument aufzunehmen. Hier ist ein Beispiel. Wir wollen eine Funktion definieren, die zu einer Kundennummer den Nachnamen des Kontoinhabers ausgibt. Der Inhaber wird unter dem Tag `name` des Kontos geführt. Intuitiv würde man dies wie

folgt machen.

```
<xsl:function name="w:kdr">
    <xsl:param name="k"/>
    <xsl:value-of select="/konten/konto[@id =
        $k]/name/@nachname"/>
</xsl:function>
```
(1.136)

Anschließend würde man die Funktion wie folgt aufrufen, um den Inhaber des Kontos mit der Nummer 125 zu bekommen.

```
w:kdr('125')
```
(1.137)

Dies führt jedoch zu einer Fehlermeldung, die besagt, dass kein Kontext definiert sei. Den Kontext führen wir in Gestalt eines neuen Arguments, hier `c` ein.

```
<xsl:function name="w:kdr">
    <xsl:param name="k"/>
    <xsl:param name="c"/>
    <xsl:value-of select="$c/konten/konto[@id =
        $k]/name/@nachname"/>
</xsl:function>
```
(1.138)

Um nun die Funktion aufzurufen, muss man angeben:

```
w:kdr('125',/.)
```
(1.139)

Das zweite Argument ist der Kontext. Dieser ist auf den Wurzelknoten gesetzt.

1.6.8 Nummerierung

Um sich Zahlen ausgeben zu lassen, steht außer dem Konstruktor `<xsl:value-of>` auch noch der Konstruktor `<xsl:number>` zur Verfügung. Der Letzte hat allerdings völlig andere Eigenschaften, weswegen es sich lohnt, ihn gesondert zu betrachten. Vorweg sei gesagt, dass er erstens zur automatischen Nummerierung von Items benutzt werden kann und dass zweitens die Formatierung der Ausgabe viel flexibler ist, schon weil die Werte, die ausgegeben werden können, lediglich natürliche Zahlen bzw. Sequenzen von solchen sind. Hier ist die vollständige Syntax.

```
<xsl:number
    value? = { XPath-Ausdruck }
    select? = { XPath-Ausdruck }
    level? = "single" | "any" | "multiple"
    count? = { Pattern }
    from? = { Pattern }
    format? = { Zeichenkette }
    language? = { NMtoken }
    letter-value? = "alphabetic" | "traditional"
    ordinal? = { Zeichenkette }
    grouping-size? = { Zahl }
/>
```
(1.140)

<xsl:number> ist eine Instruktion. Sie erzeugt eine Zeichenkette. Sie darf nur innerhalb eines Sequenzkonstruktors stehen. Dies ist deswegen so, weil <xsl:number> von sich aus zählt, an welcher Stelle man ist und diese dann gemäß der Formatierungsanweisung ausgibt. Zum Beispiel kann es eingesetzt werden, um eine Zeilennummerierung zu generieren. Die Nummer der Zeile ist wird dabei automatisch erzeugt, das ist Teil von <xsl:number>.

Die Attribute zerfallen in zwei Gruppen. Die erste Gruppe betrifft die Frage, welcher Wert eigentlich erzeugt wird, die zweite, wie er ausgegeben wird.

1. `value` erlaubt, eine Zahl explizit anzugeben. In diesem Fall wird <xsl:number> nur zur Formatierung eingesetzt.

2. `select` erlaubt, die Knoten festzulegen, deren Ordnungsnummen ausgegeben werden soll. Ist dieses Attribut nicht gegeben, wird der Kontextknoten gewählt.

3. `level` bestimmt die Art, wie gezählt wird und welche Niveaus erfasst und ausgegeben werden (ich komme darauf zurück).

4. `count` bestimmt, welche Knoten gezählt werden sollen.

5. `from` bestimmt, ab wann jeweils neu gezählt werden soll.

Die Werte von `level` sind entweder `single`, `any` oder `multiple`. Beginnen wir mit `single`. Hier zählen wir eine bestimmte Sorte von Knoten, die alle von einem einzigen Knoten abhängen (also Schwesterknoten sind). Mit `any` erlauben wir, in beliebiger Schachtelungstiefe zu zählen. In diesem Fall wird das Attribut

from interessant, weil es zu definieren erlaubt, wann die Zählung erneut beginnen soll. Man denke etwa an Fußnoten, deren Zählung sehr oft mit jedem Kapitel neu beginnt.

```
<xsl:template match="footnote">
    <xsl:number level="any" from="kapitel"/>
    <!- weitere Anweisungen ->
</xsl:template>
```
(1.141)

Man beachte an diesem Beispiel, warum die Attribute level und from hier notwendig sind. Die Anweisung, die Nummer zu erzeugen, steht innerhalb eines Templates, ist also strukturell nicht dort, wo die auszugebende Zahl entsteht. Auf der anderen Seite wird das Template an sehr vielen verschiedenen Knoten zur Anwendung kommen, und dann entsteht die Frage, wie der Prozessor die verschiedenen Vorkommen zählen soll. Bei single würde die Zählung sofort abbrechen, wenn der letzte Schwesterknoten erreicht ist (from erübrigt sich dann). Bei any ist dies nicht so. Sollte der zu zählenden Knoten verschieden sein von dem Kontextknoten, so bietet das Attribut count die Möglichkeit, einen anderen Knoten einzusetzen. Allerdings kann man auch select verwenden.

Am kniffligsten ist allerdings die Variante multiple als Wert für level.

```
<xsl:template match="paragraph">
    <xsl:number
        format="1.1.1. "
        level="multiple"
        count="kapitel | abschnitt | paragraph"
        <!- weitere Anweisungen ->
    />
</xsl:template>
```
(1.142)

Dabei setzen wir voraus, dass <paragraph> innerhalb von <abschnitt> auftritt, und dieses innerhalb von <kapitel>. Auf diesem Wege werden Paragraphen innerhalb von Abschnitten innerhalb von Kapiteln gezählt. Das Attribut multiple sorgt aber dafür, dass alle drei Knoten für sich gezählt werden und ihre Ordnungsnummer so ausgegeben wird, dass die Ordnungsnummer des oberen Knotens jeweils zuerst erscheint. Die Schachtelungstiefe ist 3, wie die Formatierungsanweisung klarstellt.

Alternativ könnte man dies auch wie folgt bekommen:

```
<xsl:number count="kapitel"> <xsl:text>.</xsl:text>
<xsl:number count="abschnitt"/> <xsl:text>.</xsl:text>
<xsl:number count="paragraph"/> <xsl:text>.</xsl:text>
```
(1.143)

(Achtung: `level` hat als Default `single`, deswegen werden die Paragaphen innerhalb der Abschnitte, die Abschnitte innerhalb der Kapitel gezählt.)

Kommen wir nun zur Formatierung. Das wichtigste Element ist hier `format`. Der Wert ist eine Zeichenkette, wie etwa oben `1.1.1.`. Diese wird in eine Sequenz von alphanumerischen Symbolen und Interpunktionszeichen zerlegt. Alles, was nicht alphanumerisch ist, wird ausgegeben. Die alphanumerischen Symbole hingegen stehen für Ausgabeweisen einer Zahl. So wird also der Punkt als Punkt geschrieben, `1` hingegen bedeutet, dass die Zahl als Ziffernfolge ausgegeben werden soll. Die Folge `6, 3, 2` wird also wie folgt ausgegeben: `6.3.2.`. (Das Leerzeichen in der obigen Zeichenkette wird ebenfalls ausgegeben, ich habe es hier aus ästhetischen Gründen weggelassen.) Hier ist eine Übersicht der Optionen:

1. `1` Mindestens eine Ziffer, aber so viele wie nötig.
2. `01` Mindestens zwei Ziffern, aber so viele wie nötig.
3. `a` Mindestens 1 Kleinbuchstabe, aber so viele wie nötig. (Die Ordnung ist wie folgt: eine 1-buchstabige Sequenz ist vor einer 2-buchstabigen, diese vor den 3-buchstabigen, und so weiter. Ansonsten wird lexikographisch geordnet.)
4. `A` Wie `a`, nur dass Großbuchstaben gewählt werden.
5. `i` Römische Zahlen, in Kleinbuchstaben.
6. `I` Römische Zahlen, in Großbuchstaben.
7. `w` Die Zahl als Wort (in der gewählten Sprache), alles in Kleinbuchstaben. (ZB `eins`.)
8. `W` Die Zahl als Wort (in der gewählten Sprache), alles in Großbuchstaben. (ZB `EINS`.)
9. `Ww` Die Zahl als Wort (in der gewählten Sprache), ein Großbuchstabe gefolgt von Kleinbuchstaben. (ZB `Eins`.)

Die anderen Attribute sind wie folgt.

1. `lang` wählt die Sprache aus (wie bereits besprochen).

2. `letter-value` ist für die meisten Sprachen irrelevant.

3. `ordinal` mit Wert Zeichenkette. Ist diese leer oder das Attribut abwesend, so wird eine Kardinalzahl ausgegeben. Ansonsten kann man hier die Ordinalendung eintragen (etwa `ter`, `te` usw.). Ich habe nicht geprüft, was bei kleinen Ordinalzahlen passiert, die abweichend gebildet werden. Es muss ja zum Beispiel `erster` heißen und nicht `einster`.

4. `grouping-separator` Für große Zahlen: sagt, wie die Zifferngruppen abgetrennt werden sollen.

5. `grouping-size` Sagt, wie groß die Zifferngruppen sein sollen.

1.7 XSLT 3.0

1.7.1 Iteratoren auf Funktionen

In diesem Abschnitt soll es um eine Weiterentwicklung von XSLT gehen, nämlich XSLT 3.0. Diese ist in dem verlinkten Text ausführlich beschrieben, jedoch möchte ich einige Punkte herausgreifen. Das Dokument ist vom 2. Oktober 2014, also recht neu. Ein wichtiger Aspekt, den ich nicht besprechen werde, weil er rein technischer Natur ist, ist die Möglichkeit des Streamens. Dies wird erst bei großen Datenmengen relevant.

Im Zusammenhang mit XSLT wurde auch XPath erweitert. Es gibt jetzt XPath 3.1. Hier gibt es zwei wichtige Neuerungen. Die erste ist die Möglichkeit, Variable innerhalb von XPath zuzuweisen (mittels `let`). Die Syntax ist wie folgt. Eine Variable wird erklärt, indem man mit `let` beginnt, Variable durch `:=` zuweist und schließlich mit `return` das Ergebnis ausgeben lässt (siehe auch die Syntax von XQuery in Teil 2). Zuweisungen sind in beliebiger Zahl möglich und dürfen vorherige Variable verwenden:

```
let $i := doc('a.xml')/*, $y := $x//*
    return $y[@value gt $/@min]
```
(1.144)

Eine andere ist die Einführung von Variable für Funktionen und die höherstufigen Operationen `fold-left` und `fold-right`, sodass nicht nur, wie im Dokument erwähnt, XSLT eine eigene Progammiersprache geworden ist, sondern auch XPath den Überbau von XSLT nicht wirklich mehr benötigt.

Bevor ich diese beiden Operationen bespreche, werde ich ein neues Werkzeug in XSLT 3.0 vorführen, nämlich `xsl:iterate`. Dies erlaubt eine iterative Ausführung, bei der in jedem Zyklus die Parameter neu gesetzt werden können. Dies

ist im Prinzip nichts anderes als eine rekursive Vorschrift.

```
<xsl:stream href="employees.xml">
  <xsl:iterate select="employees/employee">
    <xsl:param name="highest" as="element(employee)*"/>
    <xsl:param name="lowest" as="element(employee)*"/>
    <xsl:on-completion>
      <highest-paid-employees>
        <xsl:value-of select="$highest/name"/>
      </highest-paid-employees>
      <lowest-paid-employees>
        <xsl:value-of select="$lowest/name"/>
      </lowest-paid-employees>
    </xsl:on-completion>
    <xsl:variable name="is-new-highest" as="xs:boolean"
      select="empty($highest[@salary ge current()/@salary])"/>
    <xsl:variable name="is-equal-highest" as="xs:boolean"
      select="exists($highest[@salary eq current()/@salary])"/>
    <xsl:variable name="is-new-lowest" as="xs:boolean"
      select="empty($lowest[@salary le current()/@salary])"/>
    <xsl:variable name="is-equal-lowest" as="xs:boolean"
      select="exists($lowest[@salary eq current()/@salary])"/>
    <xsl:variable name="new-highest-set" as="element(employee)*"
      select="if ($is-new-highest) then .
      else if ($is-equal-highest) then ($highest, .)
      else $highest"/>
    <xsl:variable name="new-lowest-set" as="element(employee)*"
      select="if ($is-new-lowest) then .
      else if ($is-equal-lowest) then ($lowest, .)
      else $lowest"/>
    <xsl:next-iteration>
      <xsl:with-param name="highest"
      select="$new-highest-set"/>
      <xsl:with-param name="lowest"
      select="$new-lowest-set"/>
    </xsl:next-iteration>
  </xsl:iterate>
</xsl:stream>
```

(1.145)

Um dies besser zu verstehen, sei die Wirkungsweise besprochen. Es geht um eine Liste von Angestellten, die einmal durchgesehen werden soll, um am Ende eine Liste der *B* bestbezahlten und eine Liste *S* der am schlechtesten bezahlten Angestellten zu bekommen. Die beiden Listen sind zunächst leer. Bei jedem Mal, wo ein Knoten besichtigt wird, fragt der Algorithmus, ob der Lohn des neuen Angestellten *a* jetzt höher liegt als der Lohn eines Angestellten in *B*. Wenn ja, wird *B* durch die Einerliste (*a*) ersetzt. Wenn der Lohn gleich ist, wird *a* zu *B* hinzugefügt. Wenn der Lohn kleiner ist, so wird die Liste unverändert gelassen. Ähnlich verfährt man mit der Liste *S*. Auf diese Weise läuft man einmal durch die Liste der Angestellten durch und bekommt die zwei Listen von Knoten, *B* und *S*.

Am Ende jedes Iterationsschritts wird in `xsl:next-iteration` festgelegt, welchen Wert die Parameter `highest` und `lowest` haben sollen. Normalerweise darf ja der Parameter nicht verändert werden. Jedoch erlaubt das Konstrukt `xsl:next-iteration` festzulegen, mit welchen Werten die Funktion erneut aufgerufen wird. Die Parameter dürfen ja durchaus mit jedem Funktionsaufruf beliebig gesetzt werden. Dies geschieht mit Hilfe des Befehls `xsl:with-param`. Die dort vereinbarten Werte werden der Funktion gegeben, wenn es einen weiteren Knoten gibt, auf den sie angewendet werden soll.

Ist die Funktion fertig, dh gibt es keine Knoten mehr, so tritt die Anweisung `xsl:on-completion` in Kraft. In diesem Fall werden in dem Tag `highest-paid-employee` die Namen aller derjenigen Angestellten ausgegeben, die in *B* stehen, sowie in `lowest-paid-employee` die Namen aller derjenigen, die in *S* stehen. (Beachten Sie, dass *B* und *S* Listen von Knoten sind.)

Eine weitere Neuerung ist `xsl:evaluate`. Es erlaubt, einen Ausdruck dynamisch zu erzeugen und dann auszuführen. Diese bedeutet also, dass man einen XPfad Ausdruck nicht konkret kennen muss (zum Beispiel, weil er noch unbekannte Funktionsnamen enthält), sondern dass man ihn in dem Moment erzeugen kann, wo diese Daten bekannt sind. Dies ist ein sehr mächtiges Werkzeug.

Vergleichen wir dies mit einem neuen Konstruktor in XPath. Wie schon gesagt, gibt es XPath 3.0. Das verlinkte Dokument beschreibt jedoch nicht alle Neuerungen. Zum Beispiel muss man sich schon das Dokument XPath Funktionen um Bescheid zu wissen. Dort finden wir eine Beschreibung der Funktionen `fn:fold-right` und `fn:fold-left`. Die Synopse ist wie folgt. `fn:fold-left` hat drei Argumente. Die ersten beiden Argument sind Listen, und das dritte eine Funktion von Paaren von Listen nach Listen. (Da Listen auch Einzelargumente sein können, ist die die allgemeinste Art von Funktion.) Betrachten wir das Bei-

spiel

```
fn:fold-left(1 to 5, 0, function($a, $b) { $a + $b }   (1.146)
```

Dabei ist `function($a, $b){ $a + $b}` derjenige Ausdruck, der bestimmt, wie die Funktion zu berechnen ist. (Dies wird nicht weiter ausgeführt.) Diese Funktion ist nun wie folgt anzuwenden. Der Anfangswert ist 0 (erstes Argument). Dies wird `$a` zugewiesen. Anschließend wird die Funktion auf alle Werte aus `1 to 5`, also `(1,2,3,4,5)` angewendet, wobei diese jeweils der Funktion als zweites Argument gegeben werden, während das erste jeweils der kumulierte Wert ist: wir bekommen damit folgende Funktionsaufrufe.

1. `0 + 1`, mit Ergebnis 1.
2. `1 + 2`, mit Ergebnis 3.
3. `3 + 3`, mit Ergebnis 6.
4. `6 + 4`, mit Ergebnis 10.
5. `10 + 5`, mit Ergebnis 15.

Damit ist das Ergebnis der Faltung 10. Hierbei ist eigentlich kein Unterschied zu `fn:fold-right` welches sich nur darin unterscheidet, welches das kumulierte Argument ist. Augenfällig wird dies bei folgendem Aufruf:

```
fn:fold-left(1 to 5, (), function($a, $b) { ($a, $b) }  (1.147)
```

Wir bekommen:

1. `((), 1)`, mit Ergebnis `(1)`.
2. `((1), 2)`, mit Ergebnis `(1,2)`.
3. `((1,2), 3)`, mit Ergebnis `(1,2,3)`.
4. `((1,2,3), 4)`, mit Ergebnis `(1,2,3,4)`.
5. `((1,2,3,4), 5)`, mit Ergebnis `(1,2,3,4,5)`.

```
fn:fold-right(1 to 5, (), function($a, $b) {($a, $b)}  (1.148)
```

Wir bekommen:

1. `(1, ())`, mit Ergebnis `(1)`.
2. `(2, (1))`, mit Ergebnis `(2,1)`.
3. `(3, (2,1))`, mit Ergebnis `(3,2,1)`.
4. `(4, (3,2,1))`, mit Ergebnis `(4,3,2,1)`.
5. `(5, (4,3,2,1))`, mit Ergebnis `(5,4,3,2,1)`.

1.7.2 Maps und Arrays

Um nun Funktionen selber als Objekte verwenden zu können (und damit letztlich alle höherwertigen Datentypen zu integrieren), wurden die allgemeinen Konstrukte `map` und `array`s geschaffen. Maps sind Abbildungen als Objekte; arrays sind Vektoren, also Funktionen von strikt positiven ganzen Zahlen in Objekte. Ich bespreche hier nur (kurz) die Idee hinter `map`.

Kapitel 2

FO und XQuery

2.1 FO

2.1.1 XSL und Layout

Die Transformationsspache XSLT ist eigentlich nur ein Teil von einem Trio. Der andere Teil ist XML Schema, welches wir zusammen mit XSLT in dem ersten Teil der Veranstaltung besprochen haben. Hier will ich nun einen Überblick über XSL-FO geben. FO steht hier für *Formatting Objects*. Die Idee hinter XSL-FO ist, dass die (zumeist visuelle) Ausgabe durch eine spezielle Sprache geregelt wird. Denn die reine Transformation mittels XSLT sorgt noch nicht dafür, dass die Information optisch angezeigt werden kann. Dazu muss sie nämlich noch auf das Papier "verteilt" werden.

Wir bekommen insgesamt eine Kaskade: angefangen mit einer XML-Datei können wir sie mit Hilfe von XSLT strukturell verändern. Anschließend können wir das Ergebnis formatieren und anzeigen. Am einfachsten wäre dies natürlich mit einem HTML Browser, aber Browser sind erstens ziemlich schlecht zu steuern, und zweitens möchte man des öfteren doch eine Papierversion haben, beziehungsweise, was irgendwie dasselbe ist, eine PDF Datei. PDF ist aber sehr schwierig selber zu erzeugen. Diese Arbeit nimmt einem XSL-FO ab.

Bevor ich die Wirkungsweise von FO näher erläutere, hier ein paar praktische Bemerkungen. Es gibt mehrere Prozessoren für FO, zum Beispiel FOP von Apache. Diesen kann man kostenfrei herunterladen, und er eignet sich auch für Kommandozeilen. (Alternativ kann man ihn auch über Oxygen benutzen.) Der Befehl lautet schlicht `fop`. Argumente kann man mit `-xsl`, `-fo`, `-xml` und `-pdf` angeben. Diese entsprechen in der angegebenen Reihenfolge: eine Transforma-

tionsdatei (mit Suffix .xsl), eine Formatierungsdatei (Suffix .fo), eine Quelldatei (Suffix .xml) und eine PDF-Ausgabedatei (Suffix .pdf). Falls wir zum Beispiel eine XML Datei namens `quelle.xml` mit Hilfe von `transform.xsl` in eine Datei `resultat.pdf` ausgeben wollen, so müssen wir wie folgt schreiben:

```
fop -xml quelle.xml -xsl transform.xsl -pdf resultat.pdf
```
(2.1)

Um es gleich vorweg zu sagen: man muss nicht sowohl eine `.fo`- *und* eine `.xsl`-Datei angeben.

Formatierung kann man anbringen, ohne dass man eine Transformation verwendet. In diesem Fall hat man eine XML Datei, die mit Hilfe von Formatierungsanweisungen strukturiert wird. Diese sieht dann wie folgt aus.

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
Text+Formatierung
</fo:root>
```
(2.2)

In diesem Fall hat die Datei das Suffix `.fo` und man benötigt auch keine Quelle, weil ja die Formatierung mit dem Text zusammen steht.

Diesen Fall werden wir vorrangig besprechen. Der andere ist davon abgeleitet: ein XSL-Stylesheet, in welchem auch Formatierungen vorkommen. Diese Datei sieht so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  version="2.0">
Transformation+Formatierung
</xsl:stylesheet>
```
(2.3)

Diese Datei enthält den Text in der Regel nicht sondern nur Templates, mit denen für eine XML-Datei festgelegt wird, wie die Elemente formatiert werden sollen. Es ist möglich, aus der XML Datei zusammen mit der Transformationsdatei eine fo-Datei zu erzeugen, die dann in einem zweiten Schritt zu der endgültigen Ausgabedatei (HTML, PDF) verarbeitet wird. Dies ist denn auch das tatsächliche Verfahren, allerdings werden diese Prozesse gerne kaskadiert und das Zwischenergebnis nicht aufgehoben.

Die Elemente, die der Layout Prozessor manipuliert, sind im Wesentlichen Rechtecke, die zuerst gegeneinander positioniert werden. Diese enthalten Inhalt, welcher aus Text, Graphik oder auch leerem Platz bestehen kann. Der Prozessor arbeitet allerdings in der Regel, ohne dass er den Inhalt der Rechtecke wirklich betrachtet. Für ihn sind lediglich die Abmessungen wichtig und die relative Positionierung.

Der Inhalt wird Seiten verteilt. Diese Seiten haben bestimmte Abmessungen. Das Layout dieser Seiten wird vorher bestimmt. Wie viele Seiten es werden und welche Elemente darin vorkommen, bestimmt der Prozessor. Die Definition der Formatierung besteht deswegen aus zwei Teilen. Der erste ist die Festlegung bestimmter Layout-Vorlagen, definiert durch

`fo:simple-page-master`

Es kann beliebig viele davon geben. Sie werden in dem Tag

`fo:master-layout-set`

gebündelt. Ferner braucht man eine Festlegung, wann welches Layout benutzt wird. Dies wird mit der folgenden Anweisung geleistet.

`fo:page-sequence-master`

Seiten bestehen aus sogenanntem statischen Inhalt (zum Beispiel Titelzeilen, deren Inhalt unabhängig vom Text ist) und dem eigentlichen Seiteninhalt. In diesen Inhalt "fließt" nun der Text. Das entsprechende Element heißt `fo:flow`. Die Gliederung von Fließobjekten wird durch verschiedene Elemente vorgenommen, Rechtecke (`fo:block`), Listen und Tabellen, sowie weitere Objekte, die wir einzelnen noch vorstellen werden.

Bei FO wird extensiver Gebrauch von Referenzierung gemacht. Das liegt daran, dass man die Seiten nicht konkret mit Inhalten füllt sondern nur ihre Form bestimmt. Die einzelnen Regionen der Seite sowie das Layout selbst werden jeweils mit Namen versehen, sodass man sie bei der Befüllung der Seiten aufrufen kann. Dadurch wird dem Prozessor mitgeteilt, welches Textelement in welche Region gesetzt werden muss.

Material. Es gibt eine Seiten im Internet, die jede für sich hilfreich sind. Dazu gehören

- W3-Spezifikation Erschöpfend, aber die Erklärungen sind oft nichts für Außenstehende. Ich benutze sie manchmal als Notnagel.

- Das Buch Ray 2003 bietet eine Einführung in FO.
- Data2Type. Sehr ausführliche Darstellung der Möglichkeiten von XSL-FO.
- Antennahouse Sehr ausführlich, daher am Anfang verwirrend. Benutzen auch fremde Stilemente. Ist aber als Quelle ganz gut, damit mit ausreichend Beispiele hat.
- W3-Schools Als Einführung recht gut, nur sind die Erklärung etwas zu knapp (und meist von dem W3-Konsortium abgekupfert).
- Erklärung der Seitenaufteilung Schöne Farbgrafiken, welche das Seitenmodell von FO illustrieren.
- Layout Sequenzierung Erklärt rudimentär, wie man die Seitenlayouts verschieden gestalten kann (zB eine verschiedene erste Seite).

Keine der Erklärungen ist aber wirklich für meinen Geschmack gut. Insofern hilft nur eines: Ausprobieren.

2.1.2 Das Seitenmodell

Eine Seite wird in 5 Regionen unterteilt. Achtung: die bereits erwähnte Erläuterung der Seitenaufteilung ist verschieden von der W3-Spezifikation, die ich hier wiedergebe.

- `region-body` (Zentrum) Das ist die zentrale Region, in die der laufende Text kommt.
- `region-start` Das ist der Rand links neben dem Zentrum. Sie schließt oben und unten zugleich mit der Seite ab.
- `region-end` Das ist der Rand rechts neben dem Zentrum. Schließt wie `region-start` oben und unten mit der Seite ab.
- `region-before` Das ist der Teil oberhalb von `region-start`, `region-body` und wird links und rechts von `region-start` bzw. `region-end` begrenzt.
- `region-before` Das ist der Teil unterhalb von `region-body` und wird von links und recht von `region-start` bzw. `region-end` begrenzt.

Die Abmessungen dieser Regionen können einzeln festgelegt werden. Betrachten wir vor diesem Hintergrund diesen Teil des Dokuments.

```
<fo:layout-master-set>
 <fo:simple-page-master master-name="A4"
   page-width="210mm" page-height="297mm"
   margin-top="1cm" margin-bottom="1cm"
   margin-left="1cm" margin-right="1cm">
  <fo:region-body margin="10mm"/>
  <fo:region-before region-name="titel" extent="20mm"/>
  <fo:region-after region-name="fuss" extent="20mm"/>
  <fo:region-start extent="10mm"/>
  <fo:region-end extent="10mm"/>
 </fo:simple-page-master>
</fo:layout-master-set>
```

(2.4)

Die Größe der Seite wird zuerst festgelegt. Dazu braucht man zwei Angaben: Breite und Höhe. Anschließend werden die Ränder angegeben (dazu braucht man nur eine einzige Zahl). Für die Regionen braucht man nur jeweils eine Zahl, die der Wert des Attributs `extent` ist. Bei `region-start` und `region-end` bestimmt diese die horizontale Ausdehnung, bei `region-start` und `region-end` die vertikale. Die Abmessungen von `region-body` stehen damit fest.

`fo:simple-page-master` ist eine Formatvorlage. Diese muss über `master-name` einen Namen bekommen. Das Tag `fo:layout-master-set` sammelt sämtliche Formatvorlagen. Die Vorlagen bekommen alle einen Namen, damit man später auf sie zugreifen kann. Diese hier heißt A4. Die Maße 297mm mal 210mm entsprechen genau den Abmessungen einer DIN A4 Seite. Die Seite bekommt in dem oben gegebenen Beispiel an allen vier Seiten einen Rand von 10mm. Der verfügbare Teil der Seite hat nunmehr die Maße 277mm mal 190mm. Der Kopf hat den Namen “titel” und die Höhe 20mm. Ebenso die Fußzeile. Der linke und rechte Rand haben jeweils die Breite von 10mm.

Der laufende Text wird durch <fo:flow> angegeben, der statische Text durch <fo:static-content>. Wie das geht, wird gleich noch besprochen. Beide haben ein Attribut namens `flow-name`, dessen Wert einer der folgenden sein kann aber nicht sein muss (Bedeutung selbsterklärend).

- `xsl-region-body`
- `xsl-region-before`

- `xsl-region-after`
- `xsl-region-start`
- `xsl-region-end`

Dies sind voreingestellte defaults und reichen für eine einfache Seite aus. Ansonsten ist der `flow-name` beliebig. Flows dürfen so lang oder so kurz sein, wie man möchte. Da allerdings Kopf- und Fußzeile jeweils nur so viel Platz enthalten, wie auf dem Block einer einzelnen Seite untergebracht werden kann, muss man entsprechend vorsichtig sein. Einzig `region-body` kann unbegrenzt viel Text aufnehmen, weil er über mehrere Seiten gebrochen wird.

Nachdem die Layoutvorlagen festgelegt wurden, muss man nun noch sagen, wenn welche Vorlage genommen werden soll. Dies geschieht mittels `fo:page-sequence-master`. Hintergrund ist, dass innerhalb eines Textes die Seitenformatierung wechseln kann. In einem Kapitel wird üblicherweise *keine* Titelzeile gesetzt und die Seitenzahl wird im Fuß der Seite mittig gedruckt, auch dann, wenn dies bei allen anderen Seiten nicht so ist. Manche Seiten bekommen auch *keine* Seitenzahl, und so weiter. Da wird nicht wissen, welcher Text am Ende auf welcher dieser Seiten landen wird, wird die Sequenzierung der Seiten festgelegt, bevor überhaupt der Text begonnen wird. Der folgende Abschnitt dürfte nach dieser Erläuterung. selbsterklärend sein.

```
<fo:page-sequence-master master-name="pages">
 <fo:repeatable-page-master-alternatives>
  <fo:conditional-page-master-reference
   page-position="first" master-reference="first-page"/>
  <fo:conditional-page-master-reference
   master-reference="other-page"/>
 </fo:repeatable-page-master-alternatives>
</fo:page-sequence-master>
```

(2.5)

Wie im Beispiel angegeben, kann man ein Layout auf die erste Seite beschränken. (Alternativ auch auf die letzte, indem man den Wert `last` benutzt.) Eine wichtige andere Bedingung, die man als Attribut einfügen kann, ist `odd-or-even`, mit Werten `odd`, `even` oder `any`. Man beachte, dass die Bedingungen in der angegebenen Reihenfolge abgearbeitet werden. Sonderfälle sollten also zuerst genannt werden.

Wir sind nun soweit, dass wir den Text eingegen können. Hier ist ein Beispiel.

```
<fo:page-sequence master-reference="A4">
 <fo:static-content flow-name="titel">
  <fo:block>Einführung</fo:block>
 </fo:static-content>
 <fo:static-content flow-name="fuss">
  <fo:block text-align="right">
   <fo:page-number/>
  </fo:block>                                          (2.6)
 </fo:static-content>
 <fo:flow flow-name="xsl-region-body">
  <fo:block>
text text text text
  </fo:block>
 </fo:flow>
</fo:page-sequence>
```

(Der Text in `fo:flow` muss in einen Block gesetzt werden.) Man beachte, wie die Referenzen jetzt benutzt werden. Da wird auf den `fo:flow-name` Bezug genommen, um zu bestimmen, wohin ein gewisser Inhalt gehen soll. Dieser wird dann vermittels der Sequenzierung auf eine Formatvorlage abgebildet, die nun ihrerseits den Inhalt formatiert. Man beachte, dass man sich mit Hilfe von `fo:page-number` die Seitennummer ausgeben lassen kann. Der Fließtext wird in `fo:flow` angegeben. Im Gegensatz zu `fo:static-content` kann dieses Element sehr umfangreich sein und sich über viele Seiten erstrecken.

2.1.3 Blöcke

Wie schon besprochen, wird der Platz in Rechtecke aufgeteilt. Das universelle Element ist hierbei `fo:block`. Sein natürliches Vorkommen ist innerhalb von `fo:flow`, aber es kann auch anderswo vorkommen, wo Elemente positioniert werden müssen (zum Beispiel `fo:static-content`).

Rechtecke besitzen verschiedene Hüllen um sich herum. Der Inhalt (content, also die eigentliche verfügbare Fläche des Rechtecks) wird eingeschlossen von der padding area. Diese von der Grenze (border), und diese von dem Rand (margin). Zusätzlich gibt es noch eine Fläche über dem Rechteck (space before) und darunter (space after), von der Breite des Rechtecks samt padding, Grenze und

Rand. In die Grenze (border) kann man zum Beispiel Farbe eintragen; man kann auch die Ränder an allen vier Seiten frei bestimmen. Näheres siehe in W3 Box Model.

Ich bespreche kurz die Optionen für diese Elemente. Zunächst die Dimensionen. Zunächst die Dimensionen des Rands.

- `margin`
- `margin-top`
- `margin-bottom`
- `margin-left`
- `margin-right`

Diese Attribute sind selbsterklärend. Die Werte dürfen metrische Maße sein (cm, mm), Punktangaben (pt) oder Zoll (in). Dimensionen kann man auch für die Grenze angeben. Das erste Attribut (ohne Modifikatoren) kann man dann verwenden, wenn alle Maße gleich sein sollen.

- `border-width`
- `border-before-width`
- `border-after-width`
- `border-start-width`
- `border-end-width`

Anstelle von `before` darf man auch `top` schreiben, anstelle von `after` `bottom`, anstelle von `start` `left` und anstelle von `end` auch `right`. Diese letzteren sind etwas suggestiver.

Zu guter Letzt noch die padding area.

- `padding`
- `padding-before`
- `padding-after`
- `padding-start`

- padding-end

Auch hier kann man left, right, top und bottom verwenden.

- border-color
- border-before-color
- border-after-color
- border-start-color
- border-end-color

Die Farben sind ein Fall für sich. Eine erschöpfende Auskunft gibt die folgende W3-Seite. Außer der Spezifikation mittels RGB gibt es noch die folgenden 17 Werte: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white und yellow.

Weiterhin gibt es noch die Stiloptionen für die Grenze.

- border-style
- border-before-style
- border-after-style
- border-start-style
- border-end-style

Werte sind unter anderem dotted, dashed, solid, double, groove und ridge. Wiederum findet man erschöpfende und klare Informationen im W3 Handbuch.

Den Hintergrund kann man ebenfalls bestimmen.

- background-color (wie oben angegeben)
- background-image (der Wert muss eine Resource sein oder none)
- background-repeat (repeat für horizontale und vertikale Wiederholung, repeat-x nur horizontale Wiederholung, repeat-y nur vertikale Wiederholung, no-repeat)
- background-attachment (scroll or fixed)

Der Text innerhalb eines Blocks wird wie folgt modifiziert.

- `font-family`
- `font-weight`
- `font-style`
- `font-size`
- `font-variant`

Auch hier kann man sich bei dem W3 Konsortium erschöpfend informieren. Ich weise darauf hin, dass FOP nicht alle Fonts unterstützt. Eine Liste kann man hier finden.

Die Positionierung des Textes wird unter anderem wie folgt bestimmt.

- `text-align`
- `text-align-last`
- `text-indent`
- `start-indent`
- `end-indent`

Die Einrückung am Anfang eines Paragraphen steuert man mit `text-indent`. Die Attribute `start-indent` und `end-indent` geben die Einrückung des *gesamten* Textes (links und rechts) an. Werte können entweder absolut (zB cm, mm, pt, px) oder in Prozent gegeben werden.

2.1.4 Listen und Tabellen

Um eine Liste zu erstellen, bedient man sich des Elements `fo:list-block`. Dieses besteht aus Elementen, die man wie folgt aufbaut.

- `fo:list-item` für die einzelnen Elemente.
- `fo:list-item-label` für das Label. Dies ist typischerweise ein Symbol, oder eine Zahl, welche man in `fo:block` einschließt.

- `fo:list-item-body` der Inhalt. Ebenfalls ein Element, welches typischerweise aus `fo:block` zusammengesetzt ist.

Tabellen sind etwas komplizierter, weil sie zweidimensional aufgebaut sind und auch noch intern aus Blockelementen bestehen.

- `fo:table-and-caption`. Dies ist das Hauptelement, welches minimal `fo:table` und optional `fo:table-caption` enthält.
- `fo:table` Die Tabelle.
- `fo:table-caption` Der Titelzeile der Tabelle.
- `fo:table-column` Enthält keinen Text sondern nur Spezifikationen für das Layout.
- `fo:table-header` Die Kopfzeile der Tabelle. Enthält die Titel der Spalten.
- `fo:table-footer` Die Fußzeile der Tabelle.
- `fo:table-body` Der Rumpf (eigentliche Tabelle).
- `fo:table-row` Eine Zeile.
- `fo:table-cell` Ein Blockelement der Tabelle.

Üblicherweise definiert man Tabellen Zeile für Zeile (wie in HTML). Die Eigenschaften von Tabellen sind ziemlich ähnlich, weswegen ich hier keine Aufzählung vornehme. Eine komplette Übersicht findet man wiederum in W3 Spezifikation.

2.1.5 FO zusammen mit XSLT

Falls man eine Liste erstellen will, ist es mühsam, die Formatierungen stets wiederholen zu müssen. Außerdem möchte man eine Nummerierung nicht selbst vornehmen müssen. Man vergleiche das zum Beispiel mit HTML, wo eine nummerierte Liste lediglich über das Tag `<ol>` eröffnet wird und die Listenelemente mit `<li>` eingeschlossen sind. Eine weitergehende Spezifierung kann hier völlig unterbleiben. Es wäre sinnvoll, wenn man die Formatierung für FO ebenso gestalten

könnte. Dazu bietet sich XSLT an. Will man nun die Vorteile von XSLT zusammen mit FO nutzen, so muss man die Präambel wie in Kapitel 1 erklärt gestalten. Anschließend kann man die Formatierung wie folgt vornehmen.

```
<xsl:template match="ol">
 <fo:list-block>
  <xsl:for-each select="li">
   <fo:list-item>
    <fo:list-item-label>
     <fo:block>
      <xsl:value-of select="position().">
     </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
     <fo:block>
      <xsl:value-of select=".">
     </fo:block>
    </fo:list-item-body>
   </fo:list-item>
  </xsl:for-each>
 </fo:list-block>
</xsl:template
```
(2.7)

Dieses Beispiel zeigt sehr schön, wie man XSLT und FO miteinander verweben kann. Die Aufgabe von XSLT ist die Transformation, also die Zusammenstellung des logischen Dokuments, wobei einzelne Elemente im Ablauf erzeugt werden, so zum Beispiel die Nummerierung. Dabei kann man sich ebenfalls auf XSLT verlassen, wenn es um die Ausgabe geht. XSLT kann Zahlen sowohl als arabische wie als lateinische Ziffern ausgeben, als Ordinalzahlen, oder auch das Buchstaben, und so weiter.

XSLT kann auch eingesetzt werden, um frei gestaltete XML Dokumente zu formatieren. Man kann dann eigene Elemente benutzen wie <kurzfassung> oder <zitat> und dann mittels XSLT in eine FO Datei übersetzen. Das erspart dann eine Menge Arbeit, weil das Quelldokument nunmehr lediglich die Gliederung enthält, während die konkrete Formatierung von der Stilvorlage übernommen wird und damit auch sehr viel kürzer ausfallen kann. In dem Abschnitt über Tabellen des W3-Konsortiums kann man sich ebenfalls ein Beispiel ansehen, wie eine Tabelle von HTML mittels XSL-FO formatiert werden kann. Man sieht dort sehr schön, wie die Formatierung elementweise durchgeführt wird. Anstelle von

<xsl:value-of> wird dort übrigens lediglich <xsl:apply-templates/> verwendet. Dies ruft rekursiv die Formatierungsvorlagen auf, bis man zum Text angekommen ist. Da für diesen keine Vorlage angegeben ist, bekommt man der Text ausgegeben, was ein willkommenes Ergebnis ist.

2.2 XQuery

2.2.1 Überblick

XQuery ist eine Sprache, die ähnlich wie XSLT zur Extraktion, Aufbereitung und Neukonfiguration von Daten gedacht ist. XQuery unterscheidet sich von XSLT in mehrerer Hinsicht. Der erste ist die Abkehr von dem syntaktischen Format. XQuery sieht nicht mehr aus wie XML, sondern schon eher wie XPath, von dem es genauso wie XSLT als unterliegende Maschine profitiert. Der zweite ist die Iterationsschleife als „das" Instrument zur Transformtion. Nach den darin vorkommenden Elementen wird es auch FLWOR genannt. Es besteht aus 5 Elementen:

For legt die Liste fest, über die iteriert wird.

Let erlaubt, Variable zu definieren.

Where erlaubt, zusätzliche Bedingungen an die Iteration zu stellen.

Order by legt die Ordnung der Abarbeitung fest.

Return gibt das Resultat an.

Hier ist ein erstes Beispiel, das ich noch erläutern werde.

```
for $s in doc('teilnehmerliste.xml')/liste/student
let $matrikel := $s/matrikel
where sum($s/klausur/aufgaben) >24
order by $matrikel
return <li>{data($matrikel)}
 Punkte: {sum($s/klausur/aufgabe)}</li>
```
(2.8)

Die dazugehörende Datei (`teilnehmerliste.xml`) sei wie folgt.

```
<?xml version="1.0"?>
<liste>
    <student>
        <matrikel>137</matrikel>
        <klausur>
            <aufgabe nummer="1">3</aufgabe>
            <aufgabe nummer="2">7</aufgabe>
            <aufgabe nummer="3">11</aufgabe>
            <aufgabe nummer="4">14</aufgabe>
            <aufgabe nummer="5">5</aufgabe>
        </klausur>
    </student>
</liste>
```
(2.9)

Die dritte Besonderheit ist, dass XQuery eigentlich eine imperative Sprache ist, auch wenn ihre Implementierung rein funktional ist. Es gibt in einem Dokument eine Präambel, in der auch Funktionen definiert werden, aber der zentrale Mechanismus ist der der Iteration (also imperativ). Die vierte Besonderheit ist die Abkehr von dem Baummodell. In XML müssen Daten stets unter einem einzigen zentralen Knoten gesammelt werden. In XQuery (und anderen neueren Sprachen) arbeitet man stattdessen mit Listen, also technisch gesehen Wäldern. Das bringt eine größere Flexibilität mit sich.

Zur Verwendung von XQuery gibt es zwei Möglichkeiten. Die eine ist über Oxygen. Die andere ist durch das Zeilenkommando. Die beste Maschine ist `basex` (unter unix siehe auch `man basex`, sofern es installiert ist). Für basex, welches recht komfortabel ist, gibt es sogar eine grafische Oberfläche, welche über das Kommando `basexgui` aufgerufen wird. Basex ist recht luxuriös. Ein wichtiges Element ist die Möglichkeit, in jeder Sprache zu ordnen (mit Hilfe sogenannter Kollationen). Erweiterungen existieren, in welchen man sogar Wortstämme aufsuchen kann (also unter Abziehen der Flexion). Das ist für die Verarbeitung von Texten sehr wichtig. Um die obige Datei, genannt `bestanden.xquery`, anzuwenden und in eine Datei namens `liste.txt` zu leiten, genügt der folgende Befehl.

```
basex -o liste.txt bestanden.xquery
```
(2.10)

Die Abkehr von der strikten Syntax bringt einige Besonderheiten mit sich, auf die man sich einstellen muss. Wie bei anderen Programmiersprachen auch, muss man

für eindeutige Lesbarkeit sorgen. Das Resultat eines einzigen Schritts in einer FLWOR-Iteration ist zum Beispiel ein einziger Block. Will man aber mehrere Elemente unterbringen, so kann man dies tun, indem man sie durch Kommata separiert. Zusätzliches Einschließen in runde Klammern ist ebenfalls nützlich.

Die Struktur einer XQuery-Datei ist wie folgt. Sie beginnt mit einer optionalen Zeile

```
xquery version "1.0";
```
(2.11)

(Die gegenwärtige Version ist 3.1. Eine Version 2.0 gibt es nicht.) Optional kann man noch `encoding "utf-8"` einfügen:

```
xquery version "1.0" encoding "utf-8";
```
(2.12)

Als nächstes folgt eine unbestimmte Anzahl von Deklarationen. Jede Deklaration muss mit einem Semikolon beendet werden. Diese können Namensraumdeklationen sein oder Funktionsdeklarationen (dazu noch später), sowie allerlei globale Abspachen wie die alphabetische Ordnung. Deklarationen beginnen mit `declare`. Ebenso kann man mit `import` auch ein Modul importieren.

```
declare default collation "?lang=de-DE";
```
(2.13)

XQuery kann wie XSLT auch ein Schema benutzen. Dazu muss man es mit `import` importieren. Dazu wird der folgende Befehl verwendet:

```
import schema "http://home/user"
  at "http://home/user/schema.xsd";
```
(2.14)

Man beachte die doppelte Buchführung. Das Schema wird mit `at` gegeben. Hinter `at` darf eine Liste von Dateien stehen. Direkt hinter `schema` muss eine valide absolute URL stehen (`targetNameSpace`) oder eine leere Zeichenkette.

XQuery benutzt statische Typenzuweisung ähnlich wie XSLT.

Ich komme noch kurz auf (2.8) zurück. Die erste Zeile ist klar: `for` bestimmt eine Variable, hier `$s`, die eine Liste durchläuft. Diese wird wie in XPath gegeben. Man beachte die Möglichkeit, direkt ein Dokument aufzurufen. Die zweite Klausel, eingeleitet durch `let`, definiert Setzungen. Darin können zusätzliche Variablen deklariert werden. Dabei muss man „`:=`” verwenden. Die `for` und `let` Klauseln dürfen beliebig miteinander vermischt werden. Es dürfen auch beliebig viele von ihnen vorkommen. Die `where` Klausel enthält eine Bedingung. Diese dient dazu, aus der Liste Elemente auszuschließen. `return` erlaubt, ein einziges Element auszugeben. Zum Beispiel gilt ein XML-Element als Wert. Der Wert kann

allerdings auch eine Liste sein. In diesem Fall muss man runde Klammern setzen und die Werte durch Kommata trennen. Diese Listen werden dann miteinander verkettet. Es gibt keinen Unterschied zwischen einem Element und einer Liste, die lediglich dieses Element enthält. Das Resultat kann natürlich auch wieder eine FLWOR-Klausel sein. Man kann im Übrigen mit der Funktion `distinct-values` die mehrfache Verwendung von Elementen unterbinden. (Bei Knoten führt dies allerdings nicht dazu, dass sie als verschieden angesehen werden, wenn ihre Werte verschieden sind.) Die Klausel für `return` darf natürlich komplex sein. Möglich sind bedingte Anweisungen nach dem Muster

`if (`<Ausdruck>`) then` <Ausdruck> `else` <Ausdruck> (2.15)

(Man beachte die Klammern sowie die Tatsache, dass `else` obligatorisch ist.) Ebenfalls kann der Ausdruck seinerseits ein FLWOR sein.

Man benötigt geschweifte Klammern, um Ausdrücke auszuwerten. Denn oftmals ist ja nicht klar, ob man jetzt Text ausgeben will, etwa `2+2`, oder den davon bezeichneten Wert. Ebenso geschieht es, dass Variable nicht einfach durch ihren Wert ersetzt werden. Damit dies aber geschieht, muss man geschweifte Klammern setzen.

Der Wert eines Knotens wird in XQuery stets zusammen mit seinen Tags ausgegeben. Wer diese nicht haben möchte, muss die Funktion `data` aufrufen.

Zu guter Letzt noch ein wichtiger Hinweis: Kommentar wird wie folgt erzeugt:

`(: `*hier den Kommentartext*` :)` (2.16)

Material. Ich benutze im Wesentlichen Walmsley 2007. Die zweite Auflage von 2015 behandelt auch XQuery 3.1. Die Standardquelle ist die Empfehlung des W3 Konsortiums. Eine gute Zusammenfassug bietet Wikipedia.

2.2.2 Listen

Listen sind Folgen von Objekten gleichen Typs. Sie sind stets flach. Es gibt nicht Listen von Listen. Die Elemente der Listen werden durch Kommata voneinander getrennt. So haben wir `(4, 5, -1)`, eine Liste von ganzen Zahlen. Objekte können mehrfach vorkommen, und die Objekte können verschieden angeordnet werden. Jedoch sind die Listen jedesmal verschieden. Deswegen sind `(4, 5, -1)`, `(4, 5, 4, -1)` und `(5, -1, 4)` als Listen verschieden. Nicht verschieden sind hingegen `((4, 5), -1)` und `(4, 5, -1)` oder gar `((4), 5, -1)`. Es gibt keinen Unterschied zwischen einer Liste mit einem einzigen Objekt und

dem Objekt selbst. Aus diesem Grund kann man zwei Listen L und L' verketten, indem man einfach schreibt `(`L`, `L'`)`. Ebenso kann man ein Element a zu L hinzufügen, indem man schreibt `(`a`, `L`)`. Von Listen kann man mittels der Funktion `distinct-values` Mehrfachnennungen eliminieren.

Auf Listen lassen sich Vergleichsoperationen anwenden. Dies sind im Einzelnen:

$$=,\ !=,\ <,\ <=,\ >,\ >= \tag{2.17}$$

Diese sind zunächst einmal auf Objekten gewissen Typs t definiert. Sie können darüberhinaus auch auf Listen von Objekten des Typs t verwendet werden. In diesem Fall ist $L\,R\,L'$ wahr (L und L' Listen, R eine der obenstehenden Vergleichsoperationen), falls es ein Element a in L gibt und ein Element a' in L', sodass $a\,R\,a'$. Hier ist ein Beispiel:

```
basex -Lq "(1,2)=(2,3)"
```
(2.18)

Das Programm antwortet darauf mit "`true`". Wir können damit effektiv testen, ob ein Element a in einer Liste L vorkommt oder nicht. Dazu schreiben wir einfach a=L. Dies bedeutet nach Definition, dass ein Element a' von L existiert mit $a = a'$. Achtung: der Test a!=L bedeutet nicht, dass a *nicht* in L vorkommt (dies leistet nur `not(`a `=` L`)`), sondern, dass a von einem Element in L verschieden ist, das heißt, L enthält ein Element ungleich a.

Listen werden oft anstelle von Mengen genommen. Obwohl sie eine Anordnung haben, ist diese unerheblich. Am nächsten kommt diesem eine Liste ohne Mehrfachnennungen, in der die Elemente eine unabhängige Ordnung haben, etwa eine Zahlenfolge, die aufsteigend angeordnet ist. Auf Knoten ist die Defaultordnung die Dokumentordnung. Diese ist wie folgt. Falls x y strikt dominiert, so ist x *vor* y. Falls x die Knoten y und z dominiert, so ist y *vor* y, falls y in dem Text linear gesehen vor z steht. Ist y vor z, so sind alle von y dominierten Knoten vor allen von z dominierten Knoten.

Auf Listen von Knoten gibt es folgende Operationen:

- `union` oder |. Repräsentiert die Vereinigung, wobei Mehrfachnennungen gestrichen werden. Die resultierende Ordnung ist die Dokumentordnung.
- `intersect`. Wählt nur diejenigen Knoten aus, die in beiden Listen vorkommen. Das Ergebnis wird in der Dokumentenordnung geordnet.
- `except`. Nimmt von der ersten Liste diejenigen Knoten heraus, die in der zweiten ebenfalls vorkommen. Anschließend wird nach Dokumentenordnung sortiert.

Achtung: diese Operationen können *nur* auf Listen von Knoten angewendet werden. In diesem Fall ist allerdings eine Dokumentenordnung definiert.

Ein wichtiges Element in XQuery ist die Ordnungsfunktion. Bevor das Ergebnis ausgegeben werden soll, können die Knoten angeordnet werden. Falls nichts gesagt ist, wird die Default Ordnung verwendet (Dokumentordnung). Ansonsten kann man eine neue Ordnung wie folgt vereinbaren.

```
(stable) order by <Pfadausdruck> <Modifikator>
  (,<Pfadausdruck> <Modifikator>)*
```
(2.19)

Im Klartext: zunächst kommt ein optionales `stable`. Dies regelt den Fall, wo die definierte Odnung keine eindeutige Entscheidung trifft (siehe oben). Dann das `order by`. Schließlich kommt eine durch Komma abgetrennte Liste von Ordnungskriterien. Seien diese K_1 bis K_n. Die effektive Ordnung bestimmt sich wie folgt. x ist vor y, falls entweder x in K_1 vor y, oder, falls weder x vor y noch y vor x in K_1, so ist x vor y in K_2; oder, falls weder x vor y noch y vor x in K_2 ist, so ist x vor y in K_3; und so weiter. Falls keine der Ordnungen x vor y setzt noch y vor x, dann ist der Prozessor frei, die Elemente irgendwie zu ordnen. Im Falle der stabilen Ordnung aber muss dann x in der Dokumentordnung vor y sein. Auf diese Weise ist das Ergebnis eindeutig bestimmt.

Ein Modifikator besteht aus drei Teilen:

- Zuerst `ascending` (aufsteigend) oder `descending` (absteigend), optional;
- dann `empty greatest` (das leere Element oder NaN ist das größte Element) oder `empty least` (das leere Element oder NaN ist das kleinste Element), optional;
- dann `collation`, dessen Argument eine Zeichenkette ist; ebenfalls optional.

Ein Modifikator kann insbesondere leer sein (er ist also optional, wenn man so will). Die `collation` ist wichtig, falls man Zeichenketten in der landeseigenen Ordnung wiedergeben möchte. Die Umlaute werden in der englischen (= default) Ordnung an das Ende gesetzt, wohingegen sie in Deutschland wie die Kombination des Vokals mit `e` behandelt werden. Genaueres kann man bei dem W3 Konsortium erfahren. Basex erlaubt als Zusatz "`?lang=de`" oder "`?lang=de-DE`". Die Dokumentordnung kann man durchgängig ändern, indem man folgende Zeile an den Anfang setzt:

```
declare default collation "?lang=de-DE";
```
(2.20)

2.2.3 Elemente und ihre Konstruktion

Wie schon angedeutet, ist die Syntax von XQuery nicht so konsequent XML basiert. Das hat zur Folge, dass man wie bei vielen anderen Programmiersprachen auch einiges dafür tun muss, dass die Ausdrücke auch so gelesen werden, wie sie gemeint sind. Ein Vorteil, den XQuery bietet, ist, dass es — analog wie XSLT — auf die Einhaltung der XML Syntax besteht.

Bei der Ausgabe von Elementen wie etwa

```
<li>137, Punkte: 40</li>
```
(2.21)

ist es darum klar, dass es sich hierbei um eine Einheit handelt. Klammern müssen also nicht gesetzt werden. Ebenso hatte ich schon bemerkt, dass ein Element für XQuery mitsamt seinem Tag ausgeworfen wird. Will man dies nicht, so muss man die Funktion `data` benutzen. Falls nun ein Ausdruck zu einem Attribut ausgewertet wird, etwa `@isbn`, so wird dieses von XQuery auch dem Element als Attribut zugegeben. Das heißt, XQuery behandelt den Wert immer noch wie ein Attribut (samt seinem zugehörigen Wert). Also etwa

```
for $b in doc('liste.xml')/katalog/buch
return <li>{$b/@isbn} {data($b/titel)}</li>
```
(2.22)

Dies liefert für

```
<buch isbn="123456789X">
 <titel>Madame Bovary</titel>
</buch>
```
(2.23)

die folgende Ausgabe:

```
<?xml version="1.0" encoding="utf-8"?>
<li isbn="123456789X">Madame Bovary</li>
```
(2.24)

Es ist also zunächst einmal möglich, Elemente direkt zu erzeugen, indem man ihr Tag hinschreibt sowie die Attribute und ihre Werte. Außerdem kann man relativ kurz Attribute hin- und herschieben. Man kann Elemente auch mit Hilfe der Mengenklammern (also { und }) erzeugen. Eingeschlossen in diese Klammern darf eine Liste von Objekten stehen, die der Reihe nach ausgewertet und dann in der gegebenen Reihenfolge ausgegeben werden. Alternativ hätten wir auch so schreiben können:

```
for $b in doc('liste.xml')/katalog/buch
return <li>{$b/@isbn, data($b/titel)}</li>
```
(2.25)

Ebenso wäre in `bestanden.xquery` auch Folgendes möglich (siehe Beispiel 2.8):

```
return <li>{data($matrikel), "Punkte:",
  sum($s/klausur/aufgabe)}</li>
```
(2.26)

Manchmal allerdings muss man die Werte der Attribute erst erzeugen. Dazu kann man an der Stelle des Werts auch einen Ausdruck schreiben, der diesen Wert erzeugt, etwa, falls einer Variable namens `$kennnummer` bereits eine Zeichenkette zugewiesen ist, dieser Ausdruck.

```
<buch isbn={$kennnummer}>
```
(2.27)

Manchmal aber ist selbst der Name des Elements beziehungsweise der Attribute nicht bekannt. In diesem Fall müssen diese ebenfalls erzeugt werden. Dazu stehen nun Funktionen bereit, die Zeichenketten in Elemente beziehungsweise Attribute umwandeln.

Die erste Funktion heißt `element` und besitzt zwei Argumente. (Siehe 3.9.3 Computed Constructors.) Das erste definiert das Tag und das zweite den Inhalt. So kann man mit dem Folgenden dasselbe erreichen wie mit (2.26).

```
result element li {data($matrikel), "Punkte: ",
  sum($s/klausur/aufgabe)}
```
(2.28)

Der erste Ausdruck kann auch komplizierter sein (zum Beispiel ein bedingter Ausdruck). Der zweite Ausdruck ist obligatorisch in geschweifte Klammern zu setzen. Natürlich kann man das Tag auch ausrechnen lassen. In diesem Fall muss auch der erste Ausdruck in geschweifte Klammern gesetzt werden.

Analog dazu gibt es eine zweistellige Funktion `attribute`, dessen erstes Argument der Name des Attributs und dessen zweites Argument der Wert dieses Attributs ist. Wiederum ist das zweite Argument obligatorisch in geschweifte Klammern zu setzen, das erste kann auch ein einziger String sein. Hier nun eine Kombination dieser Möglichkeiten. Wir erzeugen das Element und ein Attribut.

```
result element li {attribute matrikel {$matrikel},
  "Punkte: ",
  sum($s/klausur/aufgabe)}
```
(2.29)

Dies erzeugt ein Attribut `matrikel` im Tag `li`, dessen Wert die Matrikelnummer ist.

```
<li matrikel="137">Punkte: 40</li>
```
(2.30)

Man beachte, dass die Konstruktion des Elements wie folgt verlaufen muss: zunächst werden die Attribute und ihre Werte erzeugt, erst dann kommt der Wert des Elements.

Auf diesem Wege kann man also Attribute, deren Werte und jeden beliebigen Text zum Namen eines Elements machen und umgekehrt, sofern die üblichen Konventionen eingehalten werden.

Interessante Funktionen können auf diesem Weg erzeugt werden. So kann man gezielt Attribute aus einem Knoten wegnehmen. Ist `$s` einem Knoten zugewiesen, so kann man mittels des folgenden Ausdrucks den Knoten samt Teilstruktur bekommen mit Ausnahme sämtlicher Töchter namens `isbn`. (Auch wenn es den Anschein hat, als sei `isbn` ein einzelnes Element so steht es doch hier für Markup, insofern repräsentiert es eine Liste von Knoten.)

```
element {fn:node-name($s)} {$s/(@*, * except isbn)}
```
(2.31)

Oder man fügt auf dem folgenden Weg ein Attribut samt Wert hinzu.

```
element {node-name($s)} {attribute isbn {"1234"},
  $s/(@*, *)}
```
(2.32)

Außer `element` und `attribute` gibt es noch folgende Konstruktoren:

- `document`. Ein Argument. Erzeugt einen Dokumentknoten. Dies ist nur dann wichtig, wenn man ein Dokument abliefern muss anstelle einer einfachen Struktur.

- `text`. Ein Argument. Gebrauch:

  ```
  text {"Hallo!"}
  ```

- `comment`. Ein Argument. Gebrauch:

  ```
  comment {"Dies ist ein Kommentar."}
  ```

- `processing-instruction`. Zwei Argumente. Gebrauch:

  ```
  processing-instruction {"xml"}
  {"version='1.0'"}
  ```

2.2.4 Funktionen

XQuery erlaubt es, Funktionen zu definieren. Das generelle Schema ist wie folgt.

```
declare function <funktionsname>(<Argumentliste>)
  as <Listentyp>{<Prozedur>}
```
(2.33)

Die Argumentliste ist eine Folge von Variablen (`$i`, `$betrag`, und so weiter) zusammen mit einer Typdeklaration (`as xs:string`).

```
declare function local:quadrat ($i as xs:int)
  as xs:integer {$i * $i};
```
(2.34)

Wie schon bei XSLT dürfen Funktionen nicht ohne Namensraumpräfix auftreten. Hier kann man zum Beispiel `local` verwenden, welches voreingestellt zur Verfügung steht. (Man muss ihn also nicht mehr deklarieren.)

```
declare function local:quadrat ($i as xs:integer)
  as xs:integer {$i * $i};
for $i in (2, 3, -1)
return local:quadrat ($i)
```
(2.35)

Dies ergibt als Ausgabe

```
4 9 1
```
(2.36)

Man beachte, dass die Liste ohne Kommata ausgegeben wird, die Werte werden durch Leerzeichen getrennt hintereinandergeschrieben. Will man dies vermeiden, so muss man die Funktion `concat` verwenden.

```
declare function local:doppeln ($i as xs:string)
  as xs:string {concat($i,$i)};
```
(2.37)

Wie schon besprochen, ist es möglich, in der Präambel ein Modul zu laden. Dazu muss das Modul einen Namensraum definieren, der dem in der aufrufenden Datei nicht gleichen muss. Hier ist ein Beispiel. Dies ist das Modul.

```
module namespace q =
  "http://wwwhomes.uni-bielefeld.de/mkracht";
declare function q:quadrat ($i as xs:integer)
   as xs:integer {$i * $i};
```
(2.38)

(Achtung: Ein Modul darf nur Deklarationen enthalten.) Und hier ist nun die eigentliche Datei, die diese Funktion benutzt.

```
import module namespace p =
  "http://wwwhomes.uni-bielefeld.de/mkracht"
  at "quadrat.xquery";                                    (2.39)
for $i in (2, 3, -1)
return p:quadrat ($i)
```

Auch rekursiver Funktionsaufruf ist möglich.

```
declare function local:exp
  ($i as xs:integer, $j as xs:integer) as xs:integer
    {
      if ($j = 0)                                         (2.40)
      then 1
      else $i * local:exp ($i, $j - 1)
    };
```

Der Compiler macht allerdings keine Überprüfung, ob die Rekursion sinnvoll ist oder terminiert. Es ist also Vorsicht geboten. Dennoch ist eine Rekursion in XML Dateien sehr sinnvoll, wenn sie zum Beispiel über die Baumstruktur geht. Davon werden wir im nächsten Kapitel ausführlich Gebrauch machen.

Zum Schluss noch ein Wort zu den Typdeklarationen. Wie auch in XPath sind die Funktionen des öfteren flexibel. So kann man gestatten, dass ein Argument ausgelassen wird. In diesem Fall muss die Typdeklaration mit einem Fragezeichen enden. Oder man lässt eine Liste von Argumenten der gleichen Typs zu (etwa `xs:integer*`, eine Liste von ganzen Zahlen), beziehungsweise eine nichtleere Liste (`xs:integer+`). Die Anwesenheit eines Arguments kann man mit Hilfe eines Tests prüfen. Ist zum Beispiel `$i` vom Typ `xs:integer?`, so kann man die Anwesenheit mittels `if ($i)` überprüfen. Hierbei ist Vorsicht geboten: Der Test liefert `true` sofern der Variable `$i` eine Zahl verschieden von Null zugewiesen wird. Wenn man also lediglich die Anwesenheit haben will, so muss man `if ($i or ($i = 0))` schreiben. Siehe auch die Spezifikationen des W3 Konsortiums zum effektiven booleschen Wert.

2.2.5 Neuere Entwicklungen in XQuery 3.0

Das Zählen von Elementen in einer funktionalen Programmiersprache ist nicht so ohne weiteres möglich, da das dynamische Ersetzen in der Form $x := x + 1$

nicht erlaubt ist. In XQuery wird daher in dem FLWOR Element erlaubt, die `for`-Anweisung mit einer Zählvariable anzureichern:

```
for $i at $zahl in
    doc('bestellungen.xml')/bestelliste/bestellung
  return <tr>
    <td>{$zahl}</td>
    <td>{data($i/wert)}</td>
    </tr>
```
(2.41)

Das Zauberwort ist `at`, welches sofort auf die Deklaration der Laufvariable `$i` folgt und einen Zähler einführt mit Namen `$zahl`. Jedesmal, wenn ein neuer Wert für `$i` eingesetzt wird, wird der Wert von `$zahl` um Eins erhöht. In dem Beispiel erzeugt dies eine nummerierte Tabelle von den Bestellungen zugeordneten Werten.

Dieses Instrument hat einen Nachteil: wird die Liste anschließend umgeordnet, so wird mit ihr auch die Nummerierung durcheinandergebracht. Man kann sich damit behelfen, dass man zunächst die Liste sortiert und dann erst abarbeitet:

```
let $q := for $i in
    doc('bestellungen.xml')/bestelliste/bestellung
order by $i/wert descending
  return $i
for $i at $zahl in $q
  return <tr>
    <td>{$zahl}</td>
    <td>{data($i/wert}</td>
    </tr>
```
(2.42)

Dies ist aber nicht sehr vorteilhaft, weil man typischerweise erst sortieren möchte, bevor man nummeriert. In XQuery 3.0 ist deswegen ein neuer Befehl eingeführt worden:

```
for $i in
    doc('bestellungen.xml')/bestelliste/bestellung
  order by $i/wert descending
  count $zahl
  return <tr>
    <td>{$zahl}</td>
    <td>{data($i/wert}</td>
    </tr>
```
(2.43)

(Siehe dazu die Spezifikation von XQuery 3.0.) Hierbei wird die Variable zwar vor der Sortierung deklariert, ist jedoch bereits in `where`, `let` und `return` einsetzbar. Der Einsatz in `order by` macht selbstverständlinh keinen Sinn.

In `where` Bedingungen kann man Bedingungen auch an die Position stellen. Dazu gehört die Abfrage, ob man an 1., 2. Stelle steht wie auch, ob man an letzter Stelle steht. Dazu muss man sich allerdings zunächst die Liste besorgen, um dann abzufragen, ob `$zahl` gleich der Länge der Liste ist (dazu stellt XPath die Funktion `count` zur Verfügung, die die Anzahl der Elemente einer Liste angibt.)

Eine weitere Möglichkeit, die eingebaut wurde, ist die Spezifikation `group by`, die ähnlich wie in XSLT funktioniert. Sie ist ein neues Element in dem FLWOR-Konstrukt und darf nach `let` bzw. `for` eingesetzt werden (siehe die Spezifikation).

Zu guter Letzt bespreche ich noch eine Entwicklung in XPath, die von einiger technischer Bedeutung ist. Da XPath sowohl in XSLT wie auch in XQuery verwendet wird, ist diese Entwicklung sowohl für XQuery wie für XSLT relevant. Und zwar handelt es sich um Erweiterungen, die aus XPath letztlich eine volle funktionale Programmiersprache machen. (Siehe dazu auch die Ausführungen in den W3 Empfehlungen zu XPath 3.0.)

XPath erlaubt die Einführung beliebiger, auch höherstufiger, Funktionen. Zunächst einmal besitzt man die Elemente `fn:fold-left` und `fn:fold-right`. Sieher dazu den Link in XPath 3.1. Die Syntax ist wie folgt.

```
fn:fold-right($seq  as item()*,
    $zero           as item()*,
    $f              as function(item()*, item())
        as item()*)
      as item()*
```
(2.44)

(Dies ist die offizielle Version. In der Implementierung muss die Funktion allerdings zuerst stehen. Das ist leider nicht in Ordnung.) Hier ist ein Beispiel.

```
fn:fold-right(1 to 5, 0, function($a, $b) {$a +$b})
```
(2.45)

Dies summiert die Zahlen von 1 bis 5, ergibt also 15. Man beachte, dass die Funktion selber durch den Ausdruck `function` gebildet wird.

Die Wirkungsweise ist wie folgt. Gegeben eine Liste $l = (i_0, i_1, \cdots, i_{n-1})$ von Zahlen und ein Anfangswert z, so ergibt

```
fn:fold-right(l,z,function($a, $b)){$a + $b})
```
(2.46)

den folgenden Wert: $z + i_0 + i_1 + i_2 + \cdots + i_{n-1}$. Die Wirkungsweise ist wie folgt. Es wird eine Resultatvariable erzeugt, und dieser wird zuerst der Anfangswert

gegeben, hier z. Anschließend wird zu dem Anfangswert als `$a` das erste Listenelement als `$b` der Funktion `$f` übergeben (hier also die Funktion `$a + $b`). Das erste Listenelement wird aus der Liste gestrichen und das Ergebnis der Funktionsanwendung wird in die Resultatvariable geschrieben. Dies wird so oft wiederholt, bis die Liste leer ist. Der Wert der Resultatvariable ist dann das Ergebnis.

Gegeben eine beliebige Funktion f, eine Liste $L = (l_0, l_1, \cdots, l_{n-1})$ und einen Anfangswert z ist

$$\texttt{fn:fold-right}(l,\ z,\ f) = f(l_0, f(l_1, \cdots, f(l_{n-2}, f(l_{n-1}, z)) \cdots)) \tag{2.47}$$

Dies lässt sich mit Hilfe höherer Variablen (hier `$f` für einen Funktionsnamen) auch so implementieren.

```
declare function fn:fold-right(
    $seq as item()*,
    $zero as item()*,
    $f as function(item(), item()*) as item()*)
    as item()* {
  if (fn:empty($seq))
  then $zero
  else $f(fn:head($seq),
    fn:fold-right(fn:tail($seq), $zero, $f))
};
```
(2.48)

Man beachte, dass für diese Definition XQuery und XPath die Werte immer hin- und herschieben müssen. Mittels `fn:fold-right` lässt sich dies nun innerhalb von XPath selbst erledigen.

Analog dazu existiert `fn:fold-left`, bei der lediglich die Zuordnung der Variablen `$a` und `$b` andersherum läuft. Gegeben eine beliebige Funktion f, eine Liste $L = (l_0, l_1, \cdots, l_{n-1})$ und einen Anfangswert z ist

$$\texttt{fn:fold-left}(l,\ z,\ f) = f(f(\cdots f(f(z, l_0), l_1) \cdots, l_{n-2}), l_{n-1}) \tag{2.49}$$

Ferner existiert eine Funktion `fn:for-each-pair`, die zwei Listen $L = (l_0, l_1, \cdots, l_{n-1})$ und $M = (m_0, m_1, \cdots, m_{n-1})$ sowie eine Funktion f als Argumente erhält und die Liste $(f(l_0, m_0), f(l_1, m_1), \cdots, f(l_{n-1}, m_{n-1})$ ausgibt.

Es gibt in XPath 3.0 inzwischen auch neue Typen, nämlich Abbildungen (maps) und Vektoren (arrays). Diese stelle ich jetzt vor. Seien α und β Typen, dann kann

man eine (partielle) Funktion von Objekten des Typs α in Objekte des Typs β definieren. (Siehe dazu XPath 3.1 Maps and Arrays.) Hier ist ein Beispiel

```
map {
    "So" : "Sonntag",
    "Mo" : "Montag",
    "Di" : "Dienstag",
    "Mi" : "Mittwoch",
    "Do" : "Donnerstag",
    "Fr" : "Freitag",
    "Sa" : "Samstag"}
```
(2.50)

Die Abbildung wird also durch explizite Aufzählung beschrieben. Damit wird erreicht, dass zB die Zeichenkette `So` auf die Zeichenkette `Sonntag` abgebildet wird. Wir haben eine Abbildung von `xs:string` nach `xs:string`. Die Ordnung der Zeilen ist unerheblich. Ein Leerzeichen vor oder nach dem Doppelpunkt ist wesentlich. Der Wert kann auch wiederum eine Abbildung sein.

Es gibt eine Reihe von Funktionen, um mit Abbildungen umzugehen, siehe Maps and Arrays. Die wichtigste ist `map:get`. Sie hat zwei Argumente, das erste ist die Abbildungen und das zweite das Argument. Man bekommt den durch die Abbildung zugeordneten Wert. Wir im obigen Fall die Abbildung einer Variable `$wochentage` zugewiesen, so ergibt

```
map:get($wochentage,"Mo")
```
(2.51)

den Wert `"Montag"`. Da diese Abbildungen technisch auch Funktionen sind, werden sie nicht von Funktionen unterschieden, und wir dürfen auch schreiben

```
$wochentage("Mo")
```
(2.52)

Ebenso weise ich auf die Funktionen `map:entry` und `map:merge` hin. `map:entry` bildet eine Abbildung mit einem einzigen Paar. `map:merge` nimmt eine beliebige Liste von Abbildungen und fusioniert sie (sofern die beteiligten Typen einheitlich sind). Dies erlaubt, Abbildungen aus XML-Strukturen zu extrahieren:

```
map:merge(for $b in //book return
                                map:entry($b/isbn, $b))
```
(2.53)

Die Abbildungen machen Konstruktionen wie Schlüssel (siehe die Ausführungen in 1.6.5) übeflüssig. Da diese in XPath zur Verfügung gestellt werden, kann man in XSLT auf auf `xsl:key` im Prinzip verzichten.

Das zweite Instrument sind Vektoren. Vektoren sind Funktionen von Zahlen nach Objekten eines Typs β. Dabei ist festgelegt, dass der Vektor als Vorbereich die Zahlen $1, 2, \cdots, n$ benutzt, also ein Anfangsstück der natürlichen Zahlen. Anstelle von `array` könnte man zwar den Konstruktor `map` einsetzen. Allerdings sind Vektoren einfacher zu handhaben.

Beispiele.

- `[1, 3, 5, 7]` erzeugt einen Vektor mit den Zahlen 1, 2, 5 und 7. Dabei ist 1 der Wert von 1, 3 der Wert von 2, 5 der Wert von 3 und 7 der Wert von 4.
- `[ (), (2, 17,10)]` erzeugt einen Vektor mit den Werten () und (2, 17, 10).

Alternativ kann man den Konstruktor `array` einsetzen. Dieser hat als Argument eine Liste. Insofern liefert `array{ 1, 3, 5, 7}` dasselbe wie `[1, 3, 5, 7]`, aber

```
array{ (), (2, 17, 10)}                    (2.54)
```

nicht dasselbe wie `[ (), (2, 17, 10)]` sondern wie `[2, 17, 10]`, also die Abbildung $1 \mapsto 2, 2 \mapsto 17, 3 \mapsto 10$. Man kann `array` dafür einsetzen, um aus einer XML-Datei Vektoren zu extrahieren, etwa `array{ $b/autor }`, welches einen Vektor aus Knoten bildet. Das Auswerten geschieht wie folgt.

- `[ 1, 3, 5, 7](3)` ist 5.
- `[ (), (2, 17, 10)](2)` ist (2, 17, 10).
- `array{ (), (2, 17, 10)}(2)` ist 17.

2.2.6 XPath Full Text 1.0

Das Suchen von Worten im Internet ist nicht dasselbe wie das Suchen von Zeichenketten. In einer flexktierenden Sprache wie dem Deutschen haben Worte viele verschiedene Formen. Wer zum Beispiel nach dem Wort `Vater` sucht, möchte in der Regel auch die Formen `Vaters`, `Väter` oder `Vätern` gezeigt bekommen. Eine reine Zeichenkettenbasierte Suche würde daher nur einen kleinen Teil der gewünschten Vorkommen finden. Umgekehrt aber gibt es viele Vorkommen einer Zeichenkette, die gar keine Vorkommen des Worts sind; so enthält etwa das Wort `zerlegen` die Zeichenkette `legen`, aber das Wort `legen` enthält es nicht.

(Ich merke hier nur am Rande an, dass die Entscheidung, welche Worte nun zusammengesetzt sind und welche nicht, in Einzelfällen durchaus schwierig sein kann. In diesem speziellen Fall ist die Lage allerdings klar: semantisch gesehen hat `zerlegen` fast nichts mit `legen` gemein.)

Natürlich ist die Suche letztendlich immer eine Suche nach Zeichenketten, aber die Generierung der Suchterme möchte man nicht selbst erledigen. Dies ist ein mühsames Geschäft, zumal es zu einem Wort je nach Sprache bis zu Dutzenden von Formen geben kann und zudem viele verschiedene Wortklassen existieren, bei denen einem Wortstamm ganz verschiedene Wortformen gegenüberstehen. Das Wort `Demokratie` besitzt zum Beispiel nur noch eine einzige Form, `Demokratien`.

XPath Full Text 1.0 wurde genau zu diesem Zweck geschaffen: mit Hilfe eines Wortschatzes ermöglicht es die Suche nach einem Wort unabhängig von seinen verschiedenen Formen. Wie der Name sagt, ist es eine Erweiterung von XPath, genauer von XPath 2.0. Diese kann jedoch nur von XQuery genutzt werden und nicht von XSLT. Der zugehörige Namensraum ist `ft`.

Die Hauptarbeit von Full Text ist in der Definition von "Enthaltensein". Sehen wir uns ein Beispiel an.

```
for $b in /books/book
where $b/title contains text ("Hund" using stemming)
    ftand "Katze"
return $b/author
```
(2.55)

Die Kombination `contains text` erlaubt eine Volltextsuche. Dabei gibt es sehr viele verschiedene Parameter, die man einzeln einstellen kann. Der wichtigste ist die Frage, ob man alle Wortformen zu einem gegebenen Wortstamm finden möchte oder nicht. Falls ja, so muss man noch `using stemming` hinzufügen. Ein explizites Beispiel findet sich hier.

```
/books/book/title contains text "Vater"
    using language "de"
    using no wildcards
    using no thesaurus
    using no stemming
    using case insensitive
    using diacritics insensitive
    using no stop words
```
(2.56)

2.3 Funktionales Programmieren mit XML: CDuce und OCamlDuce

Wie wir im vorigen Abschnitt gesehen haben, geht die Reise dahin, dass das Arbeitspferd, XPath, zunehmend zu einer funktionalen Programmiersprache ausgebaut wird. Zusätzlich wirden XSLT ebenfalls aufgerüstet. Das erscheint allerdings überflüssig. Es erscheint ratsam, das Interface sparsamer zu gestalten, wie dies etwa mit XQuery versucht wird. Das erscheint im Moment die vernünftigste Lösung. Alternativ dazu kann man sich auch fragen, ob man nicht das Typensystem konsequent ausbaut und die Berechnungen einer höheren Programmiersprache überlässt. Idealerweise nimmt man dazu eine funktionale Sprache. Dieser Zugang wurde in CDuce mit einiger Konsequenz beschritten.

Da funktionale Sprachen getypt sind, wird in CDuce von vornherein das volle Programm gefahren: Typen jeglicher Art existieren, also neben den Basistypen auch Funktionen über Typen, sowie Vereinigung. Der Aufbau ist sehr an XML Schema angelehnt.

CDuce ist die XML Schwester von OCaml. (Nicht zufällig werden beide in Frankreich entwickelt.) In beiden Fällen ist die Software frei verfügbar. Zudem existiert eine interessante Hybridisierung mit Nahmen OCamlDuce. Indem man OCamlDuce verwendet, besitzt man zwei Universen: eines bestehend aus getypten XML Objekten, genannt X-Werten, und ein anderes aus getypten Caml-Objekten. Zwischen den Typen besteht eine Übersetzung. Die Universen sind nicht identisch, OCaml hat wesentlich mehr Funktionalität. Jedoch ist es jetzt möglich, aus einer XML Datei ein getyptes CDuce Objekt zu machen (analog wie in XML Schema), und dieses in ein OCaml Objekt zu übersetzen. Anschließend hat man die volle Bibliothek von OCaml zur Verfügung und kann, je nach Belieben, Ergebnisse wieder zurück übersetzen.

Beginnen wir mit einem Beispiel (siehe Tutorial.) Dies ist eine XML Struktur in CDuce Syntax. Man beachte den obligatorischen Gebrauch von Klammern

sowie das Fehlen der Prozessinstruktion.

```
let parents : ParentBook =
<parentbook>[
  <person gender="F">[
    <name>"Clara"
    <children>[
      <person gender="M">[
        <name>['Pål' 'André']
        <children>[]
      ]
    ]
    <email>['clara@lri.fr']
    <tel>"314-1592654"
  ]
  <person gender="M">[
    <name="Bob"
    <children>[
      <person gender="F">[
        <name>Alice
        <children>[]
      ]
      <person gender="M">[
        <name>"Anne"
        <children>[
          <person gender="M">[
            <name>"Charlie"
            <children>[]
          ]
        ]
      ]
    ]
    <tel kind="work">"271828"
    <tel kind="home">"66260"
  ]
```

(2.57)

Diese Klausel definiert ein Objekt namens `parents` und subsumiert es unter den Typ `ParentBook`. Diese Deklaration ist optional; allerdings wird CDuce versuchen, dem Objekt einen Typ zuzuweisen, auch wenn dieser keinen Namen hat.

Fehler im Typenaufbau können durch explizite Benennung der Typen besser diagnostiziert werden. Ein solches Objekt existiert allerdings erst dann, wenn der deklarierte Typ `ParentBook` definiert ist. Dies geschieht wie folgt.

```
type ParentBook = <parentbook>[Person*]
type Person = FPerson | MPerson
type FPerson = <person gender="F">
    [ Name Children (Tel | Email)*]
type MPerson = <person gender="M">
    [ Name Children (Tel | Email)*]
type Name = <name>[ PCDATA ]
type Children = <children>[Person*]
type Tel = <tel kind=?"home"|"work">
    ['0'-'9'+ '-'? '0'-'9'+]
type Echar = 'a'-'z' | 'A'-'Z' | '_'| '0'-'9'
type Email= <email>
    [ Echar+ ('.' Echar+)* '@' Echar+ ('.' Echar+)+ ]
```
(2.58)

Dies legt Typen fest, entweder, indem ein regulärer Ausdruck über Zeichenketten angegeben wird oder indem die Typen über einfachere Typen mittels einer partiellen Struktur definiert werden (dies erinnert an `ComplexType` in XML Schema). Man beachte auch die Disjunktion in der Definition von `Person`.

Zum Vergleich zeige ich einige Vereinbarungen in XML Schema. Die erste Zeile sieht ungefähr so aus:

```
<xs:complexType name="ParentBook">
  <xs:sequence>
    <xs:element ref="Person"
      minOccurs="0" maxOccurs="unbounded"
  </xs:sequence>
</xs:complexType>
```
(2.59)

Man beachte, dass in CDuce der Typ direkt als Iteration von Elementen eines gewissen Typs definiert wird. Dies ist in XML Schema nicht möglich. Dort wird der Typ über eine Folge von gewissen Elementen definiert, deren Typ nicht angegeben werden muss. Ähnlich sieht es bei den Zeichenketten aus. In CDuce kann

man relativ leicht Zwischentypen vereinbaren, in XML Schema geht das nicht.

```
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-zA-Z0-9_]+(\.[a-zA-Z0-9_]+)*
      @[a-zA-Z0-9_]+(\.[a-zA-Z0-9_]]+)+"/>
  </xs:restriction>
</xs:simpleType>
```
(2.60)

Index

Technische Namen

Literatur

Friedl, Jeffrey E. F. *Reguläre Ausdrücke*. Sebastopol, CA: O'Reilly, 2003.

Kay, Michael. *XSLT 2.0 and XPath 2.0. A Programmer's Reference*. 4. Aufl. Indianapolis: Wrox, 2008.

Korpela, Jukka. *Unicode Explained*. Sebastopol, CA: O'Reilly, 2006.

Ray, Eric T. *Learning XML*. 2. Aufl. Sebastopol, CA: O'Reilly, 2003.

Vlist, Eric van der. *XML Schema: The W3C's Object-Oriented Descriptions for XML*. Sebastopol, CA: O'Reilly, 2002.

Walmsley, Priscilla. *XQuery*. Sebastopol, CA: O'Reilly, 2007.