

Kommentierte Referenz der Sprachelemente von XSLT

Überarbeitung des Referenzteils des Titels
Web-Design mit XML, dpunkt.Verlag 2001
Version 1.2

Autor: M. Knobloch © 2001

1	SPRACH-ELEMENTE VON XSLT	3
1.1	BEZEICHNUNGEN UND KONVENTIONEN	3
1.2	ELEMENTE ZUR DEFINITION VON SCHABLONEN UND DER STEUERUNG IHRES AUFRUFS	4
1.2.1	<xsl:template>	4
1.2.2	<xsl:apply-templates>.....	5
1.2.3	<xsl:call-template>	6
1.3	ELEMENTE ZUR STRUKTURIERUNG DES STYLESHEETS.....	7
1.3.1	<xsl:stylesheet>.....	7
1.3.2	<xsl:import>	8
1.3.3	<xsl:include>	9
1.4	AUSGABEORIENTIERTE ELEMENTE	9
1.4.1	<xsl:value-of>.....	9
1.4.2	<xsl:element>.....	10
1.4.3	<xsl:attribute>	11
1.4.4	<xsl:attribute-set>.....	11
1.4.5	<xsl:comment>.....	12
1.4.6	<xsl:processing-instruction>	12
1.4.7	<xsl:text>	13
1.4.8	<xsl:copy>	14
1.4.9	<xsl:output>.....	15
1.4.10	<xsl:preserve-space> und <xsl:strip-space>	17
1.5	ELEMENTE ZUR ERZEUGUNG VON VARIABLEN UND PARAMETERN.....	17
1.5.1	<xsl:variable>.....	17
1.5.2	<xsl:param>.....	19
1.5.3	<xsl:with-param>	19
1.6	ELEMENTE ZUR STEUERUNG BEDINGTER VERARBEITUNG	19
1.6.1	<xsl:if>.....	19
1.6.2	<xsl:choose>, <xsl:when>, <xsl:otherwise>	20
1.6.3	<xsl:for-each>	20
1.7	ELEMENTE FÜR SORTIERUNG UND NUMERIERUNG.....	21
1.7.1	<xsl:sort>	21
1.7.2	<xsl:number>.....	22
1.8	WEITERE ELEMENTE.....	24
1.8.1	<xsl:decimal-format>	24
1.8.2	<xsl:key>.....	24
1.8.3	<xsl:message>	25
1.8.4	<xsl:fallback>	25
1.8.5	<xsl:namespace-alias>	25
1.9	XSLT-FUNKTIONEN	27
1.9.1	current().....	27
1.9.2	document().....	28
1.9.3	element-available().....	29
1.9.4	format-number()	29
1.9.5	function-available().....	29
1.9.6	generate-id().....	30
1.9.7	key().....	30
1.9.8	system-property().....	31
1.9.9	unparsed-entity-uri().....	31
2	XPATH-FUNKTIONEN	31
2.1	FUNKTIONEN, DIE SICH AUF KNOTENMENGEN (NODE SETS) BEZIEHEN	31
2.2	FUNKTIONEN, DIE SICH AUF ZEICHENKETTEN BEZIEHEN.....	31
2.3	FUNKTIONEN, DIE LOGISCHE WERTE ERZEUGEN (XPATH 4.3)	32
2.4	NUMERISCHE FUNKTIONEN.....	32

1 Sprach-Elemente von XSLT

1.1 Bezeichnungen und Konventionen

Attributwert-Schablone (`attribute value template`)

bezeichnet das Attribut einer XSLT-Anweisung, dessen Wert nicht als feststehende Zeichenkette notiert wird, sondern einen Ausdruck enthalten darf. Dieser Ausdruck muß in geschweiften Klammern notiert werden.

Folgende Zeile enthält einen feststehenden Attributwert:

```
<a href="test.xml">test</a>
```

Die entsprechende Attributwert-Schablone, die ihren Wert aus dem `src`-Attribut des aktuellen Knotens des Eingabedokuments empfängt, sieht dagegen so aus:

```
<a href="{@src}">test</a>
```

Attributwert-Schablonen dürfen nur verwendet werden, wenn die Attribute des Elements als Attributwert-Schablonen interpretierbar sind. Nicht auf alle Attribute die in xslt-Elementen auftreten können, trifft dies zu (vgl. XSLT §7.6.2).

Dokumentordnung (`document order`) beschreibt die Reihenfolge der Elemente, wie sie im XML-Dokument notiert sind. Die Ausgabe einer XSLT-Transformation kann von der Dokumentordnung des Eingabedokuments abweichen.

Eingabe- / Ausgabebaum (`input tree / source tree, output tree`)

Jede XSLT-Transformation baut zu Beginn der Verarbeitung eine Repräsentation des Eingabedokuments auf, die einer Baumstruktur aus Knoten entspricht (XPath Dokumentmodell). Die Transformation arbeitet auf dieser Repräsentation, nicht auf dem Originaldokument. Die Ausgabe ist ebenfalls als Baum strukturiert.

Entsprechend werden beide Repräsentationen als `input tree` oder `output tree` bezeichnet.

Ergebnisbaumfragment (`result tree fragment`)

bezeichnet einen Teil der Ausgabestruktur. Ein solches Fragment wird anders behandelt als eine Menge von Knoten (`node set`), die dem Eingabebaum entstammen.

Knotenauswahl / Menge von Knoten (`node set`)

bezeichnet eine Auswahl von Knoten aus dem Eingabebaum. Diese Auswahl ist Ergebnis der Auswertung eines Ausdrucks.

Literales Ergebnis-Element (`literal result element`)

ist ein Element, das direkt in das Ausgabeziel geschrieben wird z.B. `<h1>`.

Qualifizierter Name (`QName, qualified name`)

ist ein XML-konformer Name (z.B. `titel` im Element `<titel>`) mit optionaler Angabe eines Namensraum-Präfixes (z.B. `<xsl:template>`).

Schablonenrumpf / Anweisungsrumpf (`template body`)

bezeichnet den Inhalt einer Anweisung, also alles, was zwischen Start- und Ende-Tag von `<xsl:template>` `</xsl:template>`, `<xsl:if>` `</xsl:if>` etc. notiert ist.

1.1.1.1 Top-Level-Elemente

sind die XSLT-Elemente, die als direkte Kind-Elemente von `<xsl:stylesheet>` in einem XSLT-Skript auftreten können.

Weißraum / Leerraum (`whitespace`)

bezeichnet Leerzeichen, Zeilenumbruch, Absatz- und Tabulatorzeichen. Diese Zeichen werden bei der Verarbeitung einer XML-Datei ignoriert, sofern sie nicht Teil einer Zeichenkette sind.

1.2 Elemente zur Definition von Schablonen und der Steuerung ihres Aufrufs

1.2.1 `<xsl:template>`

Mit dem Top-Level-Element `<xsl:template>` wird die Ausgabe von Daten gesteuert. Ein `<xsl:template>`-Element wird entweder durch Vergleich von Knoten mit (Such-) Mustern (`pattern`) aktiviert oder durch expliziten Aufruf mit Hilfe eines Namens.

```
<xsl:template
  name=QName
  match=Muster
  mode=QName
  priority=Number >
  <xsl:param>*</xsl:param>
  template-body
</xsl:template>
```

Alle Attribute von `<xsl:template>` sind optional. Mit dem Attribut `name` kann dem Template ein Name zugewiesen werden. Ist ein Name vorhanden, kann die Schablone über die Anweisung `<xsl:call-template>` aktiviert werden.

Wird kein Name angegeben, muß jedoch ein `match`-Attribut formuliert werden. Dieses definiert ein Muster (`pattern`), mit dem die Knoten des Eingabebaums (`input tree`) verglichen werden. Bei Übereinstimmung von Muster und Knoten wird der Knoten (oder die Menge der übereinstimmenden Knoten) durch diese Schablone (`template`) verarbeitet. Ist ein `match`-Attribut vorhanden, wird die Schablone über eine `<xsl:apply-templates>`-Anweisung aktiviert. Das `match`-Attribut von `<xsl:template>` darf keine direkte Referenz auf eine Variable enthalten, d.h. dieses Attribut ist nicht als `attribute-value template` interpretierbar.

Wird das `match`-Attribut nicht angegeben, muß ein `name`-Attribut vergeben werden. Sind sowohl `name`- als auch `match`-Attribut vorhanden, kann das Template auf beide Arten aktiviert werden.

Für den Fall, daß verschiedene Template-Muster auf denselben Knoten zutreffen, kann ein numerisches `priority`-Attribut vergeben werden, mit dem der Vorrang einer Schablone (`template`) vor anderen festgelegt werden kann.

Eine weitere Möglichkeit, die Anwendung einer Schablone unter konkurrierenden Schablonen zu erzwingen, ist das `mode`-Attribut. Dieses Attribut kann bei Aktivierung von Schablonen mit `<xsl:apply-templates mode="xyz">` angegeben werden.

Im Schablonenrumpf (`template body`) werden die Verarbeitungsregeln für den passenden Knoten formuliert. Ein mögliches Kind-Element kann `<xsl:param>` sein, womit ein Parameterwert an die Schablone übergeben werden kann. Dieses Element muß vor weiteren Kind-Elementen der Schablone notiert werden.

1.2.2 `<xsl:apply-templates>`

Die Anweisung `<xsl:apply-templates>` wählt eine Menge von Knoten zur Bearbeitung aus. Sie veranlaßt den XSLT-Prozessor nach einer passenden Schablone (`<xsl:template>`) für die Verarbeitung der angegebenen Knoten zu suchen.

```
<xsl:apply-templates select=Ausdruck mode=QName>
  <xsl:with-param>Anweisungsrumpf</xsl:with-param>
  <xsl:sort />
</xsl:apply-templates>
```

Die Anweisung wird immer im Anweisungsteil einer Schablone (`template body`) verwendet. Die Beschreibung der Knoten, die zur Verarbeitung ausgewählt werden sollen, geschieht mit Hilfe des `select`-Attributs. Innerhalb des `select`-Attributs steht ein XPath-Ausdruck, der als Rückgabewert eine Menge von Knoten (`node set`) besitzt. Ein Ausdruck, welcher einen numerischen Wert erzeugt, ist an dieser Stelle also unzulässig.

Wird das `select`-Attribut weggelassen, werden alle Kind-Knoten des aktuellen Knotens ausgewählt. Entsprechend den Festlegungen der Knotentypen und Eigenschaften von XPath bedeutet dies, daß sowohl die Attribut-Knoten als auch Namensraum-Knoten nicht von der automatischen Verarbeitung erfaßt werden. Sollen beispielsweise Fragmente des Eingabebaums durchkopiert werden, genügt `<xsl:apply-templates />` nicht. Damit auch die Attribute von der ausführenden Schablone erfaßt werden, müßte die Anweisung `<xsl:apply-templates select="@*|*" />` lauten.

Mit `<xsl:apply-templates>` wird der Anweisungsteil der passenden Schablone aktiviert und die Eigenschaft "aktueller Knoten" geht an den in Arbeit befindlichen Knoten der ausgewählten Knotenmenge über. Innerhalb dieser Menge läßt sich die Position des aktuellen Knotens mit der `position()`-Funktion bestimmen, die Anzahl der Knoten in der ausgewählten Menge hingegen mit der `count()`-Funktion. Mit jedem Eintritt in den Anweisungsteil einer Schablone, die durch `<xsl:apply-templates>` aktiviert wurde, wird ein Kontext mit `current node list`, `current node` etc. aufgebaut.

Wurde innerhalb `<xsl:apply-templates>` eine `<xsl:sort>`-Anweisung definiert, wird die Auswahl der Knoten sortiert, bevor sie verarbeitet werden. Wie geht nun die Verarbeitung der weiteren Knoten vor sich?

Für jeden Knoten der Auswahl wird eine Schablone ausgewählt, welche die Verarbeitung beschreibt. Diese Auswahl wird durch das `match`-Attribut der

Schablone gesteuert. Wird keine passende Schablone gefunden, wendet der XSLT-Prozessor eine Standardschablone (`built in template rule`) an. Eine Schablone wird jedoch immer nur dann angewendet, wenn ihr `mode`-Attribut identisch ist mit dem `mode`-Attribut des Aufrufs durch `<xsl:apply-templates>`. Identität besteht auch dann, wenn das `mode`-Attribut bei beiden fehlt.

Mit Hilfe der Anweisung `<xsl:with-param>` kann der aufgerufenen Schablone ein Parameter übergeben werden. Der Wert des übergebenen Parameters wird an eine Parameter-Variable gleichen Namens der Schablone übergeben. Existieren innerhalb der Schablone nur Parameter mit abweichenden Namen, wird `<xsl:with-param>` ignoriert.

Die `<xsl:apply-templates>`-Anweisung wird am sinnvollsten dann angewendet, wenn die Struktur des zu verarbeitenden Elements komplex oder variantenreich ist oder häufige Änderungen an der Struktur zu erwarten sind, denn die Anweisung überläßt die Steuerung des Verarbeitungsablaufs den Schablonen (`template rules`) bzw. dem XSLT-Prozessor. Soll eine definierte, bekannte oder selten geänderte Elementstruktur verarbeitet werden, kann `<xsl:for-each>` verwendet werden.

1.2.3 `<xsl:call-template>`

Die Anweisung `<xsl:call-template>` wird benutzt, um eine benannte Schablone aufzurufen.

```
<xsl:call-template name=QName>
  <xsl:with-param>*</xsl:with-param>
</xsl:call-template>
```

Das bei dieser Art des Aufrufs zwingend erforderliche `name`-Attribut muß identisch sein mit dem Namen des `<xsl:template>`-Elements, welches aufgerufen wird.

Die Behandlung der optionalen Parameter ist analog zur `<xsl:apply-templates>`-Anweisung. Im Unterschied zur `<xsl:apply-templates>`-Anweisung wird durch `<xsl:call-template>` kein neuer Kontext aufgebaut, d.h., der `current node` bleibt bestehen. Diese Anweisung ähnelt damit am ehesten den Funktionen prozeduraler Programmiersprachen. Anwendungslogik, die an verschiedenen Stellen der Verarbeitung gebraucht wird, kann damit sinnvoll durch benannte Schablonen realisiert werden. Um Effekte ähnlich dem Rückgabewert einer Funktion zu erzielen, kann die `<xsl:call-template>`-Anweisung innerhalb der Erzeugung einer XSLT-Variablen aufgerufen werden.

```
<xsl:variable name="temp_wert">
  <xsl:call-template name="nummer_in_liste" />
</xsl:variable>
```

Die Variable `$temp_wert` enthält fortan den Wert, der vom Template "nummer_in_liste" erzeugt wurde. Mit diesem Wert können beispielsweise bedingte Verarbeitungen (`<xsl:if test="$temp_wert=3">`) beeinflusst werden.

1.3 Elemente zur Strukturierung des Stylesheets

1.3.1 <xsl:stylesheet>

Mit <xsl:stylesheet> wird das Wurzel-Element eines XSL-Dokuments notiert. Erlaubt ist auch die Auszeichnung <xsl:transform>.

```
<xsl:stylesheet
id="identifikation"
version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
extension-element-prefixes="präfixliste"
exclude-result-prefixes="präfixliste"
>
<!-- Top-Level-Elemente -->

</xsl:stylesheet>
```

Die Anweisung muß mindestens eine Namensraumdeklaration enthalten, möglich sind aber auch mehrere. Sollen beispielsweise Formatting Objects ausgegeben werden, ist noch folgende Deklaration einzubauen:

```
xmlns:fo="http://www.w3.org/1999/XSL/Format"
```

Das version-Attribut muß angegeben werden und darf gegenwärtig lediglich die Nummer 1.0 tragen, während die anderen Attribute optional sind.

Mit dem Attribut `extension-element-prefixes` wird eine Methode definiert, um Funktionserweiterungen eines XSLT-Prozessors zu kennzeichnen und anzusprechen, auch wenn diese zum Zeitpunkt der XSLT-Spezifikation noch nicht bekannt sind. Das Attribut besagt welches Namensraumkürzel Funktionserweiterungen kennzeichnet. Beispielsweise hat der im Buch verwendete XSLT-Prozessor Xalan erweiterte Funktionen, die es erlauben, durch die Verarbeitung eines Stylesheets mehrere Ausgabedateien zu erzeugen. Um diese zu nutzen, sind die folgenden Angaben im Stylesheet notwendig:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xalan="org.apache.xalan.xslt.extensions.Redirect"
extension-element-prefixes="xalan">
.....
</xsl:stylesheet>
```

Entsprechend werden die Erweiterungen innerhalb des Stylesheets angesprochen:

```
<xalan:write select="$Datei-Namensvariable">
  <xsl:value-of select="."/ >
</xalan:write>
```

Dieses Attribut sollte allerdings nur verwendet werden, wenn ein Stylesheet die Erweiterungen auch wirklich nutzt. Das alleinige Vorhandensein zusätzlicher Funktionen in einem XSLT-Prozessor erzeugt nicht die Notwendigkeit, `extension-element-prefixes` zu deklarieren. Erweiterungsfunktionen können vom Hersteller des

XSLT-Prozessoren oder von Dritten, also auch den Nutzern selbst zur Verfügung gestellt werden. Details dazu hängen von den APIs des Herstellers ab. Auch dann jedoch müssen die Präfixe für die Erweiterungen deklariert werden.

Hinweis: Die Fähigkeit Ergebnisse in mehrere Dateien auszugeben findet sich bei den meisten der „wichtigen“ XSLT-Prozessoren. Im Entwurf zu XSLT 1.1 wird diese Funktion deshalb in den Standard-Funktionsumfang eines XSLT-Prozessors aufgenommen.

Mit dem Attribut `exclude-result-prefixes` kann die Ausgabe der in diesem Attribut gelisteten Namensraumdeklarationen unterdrückt werden. Dies verhindert, daß sie durch eine Transformation in das Ausgabedokument gelangen. Beispielsweise würde eine Schablone wie folgende:

```
<xsl:template match="web-design:kapitel">
  <testdruck>
    <xsl:value-of select="*">
  </testdruck>
</xsl:template>
```

automatisch das Namensraum-Präfix in das Ausgabedokument schreiben. Mit dem Attribut `exclude-namespace-prefixes="web-design"` innerhalb des `<xsl:stylesheet>`-Elements wäre dies im gesamten Stylesheet unterdrückt.

1.3.2 `<xsl:import>`

Das Top-Level-Element wird für die Modularisierung von Stylesheets benutzt. Es importiert die Definitionen eines zweiten Stylesheets in das Stylesheet, das diese Anweisungen enthält.

```
<xsl:import href="URI" />
```

Das Stylesheet, welches die `<xsl:import>`-Anweisung enthält, soll in diesem Abschnitt als Eltern-Stylesheet (E), die eingebundenen als Kind-Stylesheet (K) bezeichnet werden.

Aufgrund seiner Wirkung auf die Vorrangregelung bei importierten Anweisungen muß `<xsl:import>` als erstes Kind-Element von `<xsl:stylesheet>` notiert werden. Es ist damit das einzige Top-Level-Element, dessen Position im Stylesheet eine Rolle spielt. Alle Top-Level-Elemente von (K) werden an der Position der `<xsl:import>`-Anweisung im Eltern Stylesheet eingefügt. Das Kind-Stylesheet darf weitere `<xsl:import>`- und `<xsl:include>`-Anweisungen enthalten. Allerdings werden in jedem Fall relative Pfadangaben im href-Attribut auch relativ zur Position des Stylesheets ausgewertet, in dem sich die `<xsl:import>`-Anweisung befindet. Es kann sehr unübersichtlich werden, welches Stylesheet welches weitere einbindet. Hier lauert die Gefahr, Ketten zirkulärer `<xsl:import>`-Anweisungen zu erzeugen, die explizit verboten sind.

Bei gleichlautenden Elementen im Eltern- und im Kind-Stylesheet grieft eine Vorrangregelung. Die Vorrangregelung für importierte Top-Level Anweisungen besagt, daß sie eine niedrige Vorrangstellung haben als solche, die im Eltern-Stylesheet codiert sind. Werden mehrere Stylesheets importiert, hat jeweils das zuerst importierte Stylesheet eine niedrigere Vorrangstufe als das nächste.

1.3.3 <xsl:include>

Das Top-Level-Element wird für die Modularisierung von Stylesheets benutzt. Es bindet die Definitionen eines zweiten Stylesheets in das Stylesheet, welches diese Anweisungen enthält.

```
<xsl:include href="URI" />
```

Im Unterschied zu den durch <xsl:import> zugänglichen Elementen aus anderen Stylesheets besitzen mit <xsl:include> eingebundene Elemente die gleiche Vorrangstellung wie die innerhalb des Eltern-Stylesheet definierte. Sie wirken also, als ob sie irgendwo im Eltern-Stylesheet notiert wären. An Position der <xsl:include>-Anweisung werden die externen Schablonen damit als Text einkopiert. Der Übersichtlichkeit halber sollten <xsl:include>-Anweisungen am Beginn des Stylesheets notiert werden, wenn dies auch keine Auswirkungen auf die Verarbeitungslogik hat.

1.4 Ausgabeorientierte Elemente

1.4.1 <xsl:value-of>

Die <xsl:value-of>-Anweisung schreibt eine Zeichenkette in den Ausgabedatenstrom.

```
<xsl:value-of select="Ausdruck"
             disable-output-escaping="yes" | "no" />
```

Die Zeichenkette, die von der Anweisung zurückgegeben wird, ist die Auswertung des select-Ausdrucks. Die Angabe eines select-Ausdrucks ist zwingend. Wird das optionale Attribut disable-output-escaping mit "yes" angegeben, werden die in XML reservierten Zeichen wie & oder < ausgegeben, ohne daß eine Codierung (& oder <) verwendet werden muß. Voreingestellter Wert dieses Attributs ist "no". Das Attribut wirkt bei Angabe von "yes" wie <xsl:text>.

Vor Ausgabe der Zeichenkette finden unter Umständen Konvertierungen statt, je nachdem, welchen Wert der select-Ausdruck produziert. Der Umgang der <xsl:value-of>-Anweisung mit den Ergebnissen des select-Ausdrucks soll anhand eines kleinen Beispiel-Stylesheets dargestellt werden, welches das Folien-Beispiel als Quelldatei verwendet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" />
<xsl:template match="folien">
(1)  logischer Ausdruck:    <xsl:value-of select="1=2"/>
(2)  node set:              <xsl:value-of select="//titel" />
(3)  einzelner Knoten:     <xsl:value-of select="folie[2]" />
(4)  numerisch:            <xsl:value-of
                           select="count(descendant::node())" />
</xsl:template>
</xsl:stylesheet>
```

1. Ein logischer Ausdruck liefert die booleschen Werte true oder false. Sie werden als Zeichenkette "true" oder "false" von der Anweisung zurückgegeben. Ausgabe von (1): logischer Ausdruck: false
2. Der zweite select-Ausdruck liefert eine Menge von Knoten (node set), nämlich alle <titel>-Elemente. Erzeugt der Ausdruck eine Knotenmenge, wird lediglich der Wert des ersten Knotens (bezogen auf die Dokumentanordnung) als Zeichenkette zurückgegeben. Alle folgenden <titel>-Elemente werden ignoriert! Ausgabe von (2): node set: Folienübersicht
3. Der select-Ausdruck liefert einen einzelnen Knoten, nämlich das zweite <folie>-Element innerhalb des Dokuments. In diesem Fall werden die Werte aller Kind-Elemente des Knotens, d.h., alle Text-Knoten aller Kind-Elemente, ausgegeben. Ausgabe von (3): „Motive für den Einsatz eines CMS Eine große Menge von Inhalten muß verwaltet werden Verschiedene Personengruppen arbeiten gemeinsam an den Dokumenten“
4. Ist der Wert des Ausdrucks numerisch, findet lediglich eine Konversion in eine Zeichenkette statt. Ausgabe (4) zeigt die Anzahl aller Kind-Knoten von Root: numerisch 82

1.4.2 <xsl:element>

Um ein XML-Element in der Ausgabe zu erzeugen, kann dies direkt im Stylesheet codiert werden, sofern Name und Aufbau des Elements festgelegt, d.h., bekannt sind. Hängt die Ausformung des Elements jedoch von Daten des Eingabedokuments ab, ohne mit den Eingabe-Elementen identisch zu sein, kann die Flexibilität der <xsl:element>-Anweisung genutzt werden.

```
<xsl:element name="QName" namespace="URI"
  use-attribute-set="QNames">
  *
</xsl:element>
```

Die Angabe des name-Attributs ist zwingend, die von Namensraum und Attribute-Sets optional. Sowohl name als auch namespace können als Attributschablone (attribute value template) notiert werden. Der Inhalt des Elements besteht aus Kind-Elementen, die aus weiteren Anweisungen und literalen Ergebnis-Elementen erzeugt werden. Dem Element können Attribute entweder durch das use-attribute-set-Attribut oder durch <xsl:attribute>, <xsl:copy> oder <xsl:copy-of> zugewiesen werden. Beispielhaft läßt sich die Anweisung einsetzen, wenn die Darstellung von XML direkt mit Cascading Stylesheets realisiert werden soll. Cascading Stylesheets können verwendet werden, um die Formatierung von XML-Elementen zu definieren, nicht jedoch die Attribute. Eine Transformation, welche CSS in eine Ausgabedatei generiert, kann bei dieser Gelegenheit auch alle Attribute in Elemente umwandeln. Das CSS-Stylesheet ist damit in der Lage, alle Inhalte einer XML-Datei zu erfassen. Ein Template kann exemplarisch so aussehen:

```
<xsl:template match="@*" mode="a2e" >
  <xsl:element name="{name()}">
    <xsl:value-of select="." />
  </xsl:element>
```

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

Die Schablone überschreibt die Standardschablone für Attribute und legt ein Element auf Basis des Attributnamens an. Um dieses Verhalten nicht als global gültig zu definieren, wird das `mode`-Attribut verwendet. Der Wert des neuen Elements wird mit `<xsl:value-of>` dem Attributwert entnommen. Innerhalb einer Schablone, die Elemente verarbeitet, kann obiges Template dann mit

```
<xsl:apply-templates select="@*" mode="a2e" />
```

angestoßen werden.

1.4.3 `<xsl:attribute>`

Die Anweisung erzeugt einen Attributnamen und einen Wert im aktuell auszugebenden Element.

```
<xsl:attribute name="QName" namespace="URI">  
  Attributwert  
</xsl:attribute>
```

Attributname und Wert müssen erzeugt werden, bevor dem aktuellen Element weitere Attribute oder Kind-Elemente hinzugefügt werden. Die Angabe des `name`-Attributs ist verpflichtend. Der Wert des neu erzeugten Attributs besteht aus der Zeichenkette, die innerhalb des Elements angegeben wird.

Die Anweisung wird dann sinnvoll eingesetzt, wenn ein Attribut nicht hart codiert werden kann, sondern vom Inhalt des Eingabedokuments abhängt. Soll beispielsweise ein HTML-Link aus dieser Eingabe

```
<page title="Web-Design" src="wd.xml" />
```

erzeugt werden, kann die produzierende Schablone so notiert werden:

```
<xsl:template match="page">  
  <a>  
    <xsl:attribute name="href">  
      <xsl:value-of select="@src" />  
    </xsl:attribute>  
    <xsl:value-of select="@title" />  
  </a>  
</xsl:template>
```

Es wäre ein Fehler, wenn die `<xsl:value-of>`-Anweisung vor der Ausgabe des `href`-Attributs notiert wäre.

1.4.4 `<xsl:attribute-set>`

Das Top-Level-Element erlaubt es, eine Menge von Attributnamen und Werten mit einem Namen zu versehen. Unter Verwendung dieses Namens können die Attribute als Paket einem beliebigen Ausgabeelement zugeordnet werden.

```
<xsl:attribute-set name="QName"  
  use-attribute-sets="QNames">  
  <xsl:attribute>*</xsl:attribute>
```

```
</xsl:attribute-set>
```

Die Angabe des name-Attributs ist zwingend, mit Hilfe des optionalen use-attribute-sets-Attributs können Attribut-Set Definitionen modular aufgebaut werden. Mit keiner oder beliebig vielen <xsl:attribute>-Anweisungen werden die einzelnen Attribute des Sets definiert. Nachfolgendes Beispiel definiert eine Attributmenge für Standardschrift.

```
<xsl:attribute-set name="standardschrift">
  <xsl:attribute name="font-name">Verdana</xsl:attribute>
  <xsl:attribute name="font-size">12pt</xsl:attribute>
</xsl:attribute-set>
```

Eine Anwendung der Attributmenge gibt folgendes Beispiel, in dem ein HTML-Tabellen-Element erzeugt wird:

```
<xsl:template match="thema">
  <xsl:element name="td"
    use-attribute-sets="standardschrift">
    <xsl:value-of select="." />
  </xsl:element>
</xsl:template>
```

Wird das Ausgabeelement nicht mit <xsl:element> erzeugt, wird die Angabe eines Namensraums notwendig:

```
<td xsl:use-attribute-sets="standardschrift">
  <xsl:value-of select="." />
</td>
```

1.4.5 <xsl:comment>

Die Anweisung erlaubt es, einen Kommentar in die Ausgabe zu schreiben.

```
<xsl:comment> Kommentartext </xsl:comment>
```

Die Anweisung erzeugt lediglich einen Text-Knoten für den Inhalt des Kommentars. Dieser Text darf nicht die Zeichenfolge <<!-- oder --> also die Kommentarzeichen für XML oder HTML enthalten.

1.4.6 <xsl:processing-instruction>

Mit dieser Anweisung wird eine Processing Instruction in den Ausgabebaum geschrieben.

```
<xsl:processing-instruction name="QName">
  *
</xsl:processing-instruction>
```

Das name-Attribut ist zwingend anzugeben und darf nicht irgendeine Ausformung der Zeichenkette "xml" enthalten. Die Anweisung kann also nicht benutzt werden, um eine XML-Deklaration am Beginn der Ausgabedatei zu erzeugen. Die Deklaration gilt auch nicht als Processing Instruction. Sie wird dagegen vom XSLT-Prozessor bei entsprechender Parametrisierung des <xsl:output>-Elements erzeugt. Die Anweisung kann benutzt werden, um nachfolgenden Prozessen oder

Transformationen Informationen zu übermitteln. Beispielsweise bei der Verwendung von Stylesheets.

```
<xsl:processing-instruction name="xml-stylesheet">
  <xsl:text>href="web-design.css"
    type="text/css" </xsl:text>
</xsl:processing-instruction>
```

In Umgebungen, wie sie im vorliegenden Buch verwendet werden, ist die Generierung von PIs nur von untergeordneter Bedeutung. In serverbasierten Produktionsumgebungen, die oftmals mehrere Transformationsschritte nacheinander ausführen, sind sie jedoch ein wichtiges Steuerungsinstrument. So werden beispielsweise die verschiedenen serverseitigen Verarbeitungsschritte innerhalb des Publishing Frameworks Apache-Cocoon durch Processing Instructions gesteuert. Bei mehrstufigen Transformationen kann so je nach Anwendungslogik oder Benutzer-Request eine entsprechende Processing Instruction generiert werden, die dem Folgeprozeß mitteilt, was zu tun ist. Ein Beispiel ist der Zugriff auf die identische URL einmal aus einem herkömmliche Web-Browser, ein anderes mal mit einem WAP-Gerät. Eine erste Transformation kann die richtige Datenmenge für das jeweilige Endgerät selektieren. Sie erzeugt auch die PI für den zweiten Verarbeitungsschritt. In dieser Processing Instruction wird dann ein passendes Stylesheet angezogen, welches HTML oder eben WML produziert.

1.4.7 <xsl:text>

Die Anweisung wird benutzt, um die Ausgabe von Weißraum (`whitespace`) und von speziellen Zeichen, wie `&` oder `<`, zu steuern.

```
<xsl:text disable-output-escaping="yes" | "no" >
  text
</xsl:text>
```

Die Anweisung wirkt vor allem auf die Ausgabe von Leerraum. Ein Knoten, der nur Leerraum enthält, wird nur dann in den Ausgabebaum geschrieben, wenn er innerhalb einer `<xsl:text>`-Anweisung erzeugt wird. Am häufigsten wird die Anweisung deshalb verwendet, um Leerzeichen zwischen zwei Ausgabe-Elementen zu schaffen. Beispielsweise wird bei Ausgabe von

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

der Leerraum zwischen den beiden Elementen unterdrückt. Um den Abstand sicherzustellen, kann `<xsl:text>` verwendet werden:

```
<xsl:value-of select="nachname" />
<xsl:text> </xsl:text>
<xsl:value-of select="geburtsdatum" />
```

Eine weitere Anwendung ist die Steuerung der Ausgabe von speziellen, in XML reservierten Zeichen, die vom XSLT-Prozessor lediglich durch Codierung ausgegeben werden könnten. Dies wird durch das `disable-output-escaping`-Attribut gesteuert. Ist das Ausgabeziel nicht reines XML oder XHTML, sondern beispielsweise JSP, werden oftmals genau diese Zeichen benötigt.

```
<%@ page errorPage="errp.jsp" session="true"%>
```

Um zu verhindern, daß in der Ausgabe < statt < erscheint, wird die <xsl:text>-Anweisung folgendermaßen aufgebaut:

```
<xsl:text disable-output-escaping="yes">
    &lt;%@ page errorPage="errp.jsp" session="true"%>
</xsl:text>
```

1.4.8 <xsl:copy>

Mit dieser Anweisung wird der aktuelle Knoten des Eingabebaums in die Ausgabe kopiert. Die Anweisung erzeugt eine flache (shallow) Kopie des aktuellen Knotens, das heißt, Kind-Elemente werden nicht mit kopiert. Diese Anweisung entspricht damit der Wirkung von <xsl:element>.

```
<xsl:copy use-attribute-sets="QNames" >
    *
</xsl:copy>
```

Beim Kopieren kann mit dem optionalen Attribute use-attribute-sets eine Menge von vordefinierten Attributen dem neuen Element zugeordnet werden. Die Anweisung wirkt verschieden, je nachdem, um welche Art es sich beim aktuellen Knoten handelt. Element-Knoten: Der Knoten wird kopiert, sein Name wird übertragen, eventuell vorhandene Namensraumdefinitionen werden mit übertragen und die zugeordneten Attribut-Sets werden ausgegeben. Kind-Knoten werden nicht mit kopiert. Wird im Anweisungsrumpf beispielsweise ein <xsl:apply-templates> angegeben, wird dies ausgewertet. D.h., wenn keine weiteren Schablonen Näheres bestimmen, wird lediglich der Text aller Kind-Elemente mittels der eingebauten Standardschablonen ausgegeben.

Text-Knoten: Ein neuer Text-Knoten wird im Ausgabebaum angelegt. Übertragen wird lediglich der Wert, eventuelle Attribut-Set Angaben werden ignoriert, da Text-Knoten per Definition weder Namen noch Attribute haben.

Attribut-Knoten: Name und Wert des Knotens werden in die Ausgabe übertragen. Existiert im Ausgabebaum kein aktueller Knoten, der das Attribut aufnehmen kann, wird ein Fehler ausgegeben. Attribut-Set-Angaben werden ignoriert.

Processing Instruction und Kommentare: Ein entsprechender Knoten wird in der Ausgabe erzeugt, Attribut-Set-Angaben ignoriert, ein eventuell notierter Anweisungsrumpf innerhalb von <xsl:copy> wird ebenfalls ignoriert.

Zum Durchkopieren von Teilbäumen des Eingabedokuments kann eine Schablone mit rekursivem Aufruf erzeugt werden. Nachfolgendes Beispiel ist in der Wirkung mit der <xsl:copy-of>-Anweisung identisch, die rekursives Kopieren durchführt.

```
<xsl:template match="@*node()" mode="copy">
    <xsl:copy>
        <xsl:apply-templates select="@*" mode="copy" />
        <xsl:apply-templates mode="copy" />
    </xsl:copy>
</xsl:template>
```

Durch die erste `<xsl:apply-templates>`-Anweisung ruft sich die Schablone rekursiv auf unter Auswahl der eigenen Attribut-Knoten, die ohne diese Zeile verlorengehen würden. Im zweiten `<xsl:apply-templates>` geschieht dieser Aufruf für die Kind-Elemente. Wie bereits erwähnt entspricht dieses Beispiel dem deep-copy von `<xsl:copy-of>`. Mit einer kleinen Manipulation lassen sich jedoch in obigem Beispiel gezielt Attribute übernehmen. Verändert man die Zeile für das Kopieren der Attribute wie folgt, werden lediglich die Attribute, deren Namen ident lautet, kopiert.

```
<xsl:apply-templates select="@*[name()='ident']"
                    mode="copy" />
<xsl:copy-of>
```

Die Anweisung kopiert einen Knoten und rekursiv all seine Nachkommen in den Ausgabebaum.

```
<xsl:copy-of select="Ausdruck" />
```

Die Anweisung wirkt wie `<xsl:value-of>`, wenn der Ausdruck keine Menge von Knoten ergibt, im anderen Falle werden alle Attribut- und Kind-Knoten durchkopiert. Diese Eigenschaft wird verwendet, wenn Teile des Eingabedokuments unverändert durchkopiert werden sollen.

Ein weiterer Anwendungsfall ist die Verwendung von `<xsl:copy-of>` für den Zugriff auf den Ausgabebaum. Dies kann mit Hilfe einer Variablen und einer Variablenreferenz im Ausdruck erzeugt werden. Beispielsweise soll eine Spaltenbeschriftung von Tabellen mehrfach in der Ausgabe genutzt werden. Dazu kann zunächst eine Variable angelegt werden.

```
<xsl:variable name="beschriftung" >
  <tr>
    <td>Name</td>
    <td>Geburtstag</td>
  </tr>
</xsl:variable>
```

In der Schablone, welche die Tabellen aufbaut, kann nun mit Hilfe von `<xsl:copy-of>` diese Struktur reproduziert werden:

```
<xsl:template match="..." >
.....
  <table>
    <xsl:copy-of select="$beschriftung" />
    <tr>
      <td><xsl:value-of .../></td>
      <td><xsl:value-of .../></td>
    </tr>
  </table>
</xsl:template>
```

1.4.9 `<xsl:output>`

Das Top-Level-Element ermöglicht es, das Ausgabeformat eines Stylesheets zu steuern. Diese Anweisung wirkt direkt auf die Serialisierung des Ergebnisbaums d.h.,

in der Regel auf das Schreiben der Ergebnisse in Dateien. In Umgebungen, die dynamisches Publizieren z.B. von Web-Seiten erlauben, kann es sein, daß keine Serialisierung stattfindet, sondern eventuell der Ergebnisbaum oder eine DOM-konforme Struktur an Folgeprozesse übergeben wird. In solchen Umgebungen darf `<xsl:output>` ignoriert werden.

```
<xsl:output
  method="xml" | "html" | "text" | "QName"
  version="Zeichenfolge"
  encoding="Zeichenfolge"
  indent="yes" | "no"
  omit-xml-declaration="yes" | "no"
  cdata-section-elements="QNames"
  doctype-public="Zeichenfolge"
  doctype-system="Zeichenfolge"
  standalone="yes" | "no"
  media-type="Zeichenfolge"/>
```

Alle Attribute sind optional, ihre Angabe ist jedoch im einzelnen abhängig von der Wahl des `method`-Attributs. In der Regel ist es nicht notwendig, die `<xsl:output>`-Anweisung in einem XSLT-Skript zu verwenden. Standardannahme ist immer, daß XML ausgegeben werden soll. Trifft ein XSLT-Prozessor als erste Ausgabeanweisung auf `<html>`, wird als `method`-Attribut "html" angenommen. Aufgrund der Vielzahl von Kombinationsmöglichkeiten der Attribute wird hier nur auf einige eingegangen.

Methode `xml`: Wird im `method`-Attribut "xml" angegeben, ist das Versionsattribut verpflichtend und derzeit auf "1.0" festgelegt. Das `encoding`-Attribut ist optional, da alle Prozessoren mindestens UTF-8 unterstützen müssen und die Verarbeitung aller Unicode-Zeichen damit gewährleistet ist. Wird das `omit-xml-declaration`-Attribut auf "yes" gesetzt, wird keine `<?xml version="1.0"?>`-Deklaration in das Ausgabedokument geschrieben.

Die Attribute `standalone`, `doctype-public` und `doctype-system` erzeugen eine entsprechende Document Type Declaration (z.B. erzeugt `doctype-system="test.dtd"` diese `<!DOCTYPE test SYSTEM "test.dtd">`), wenn der Name der DTD als Wert des Attributs angegeben wird oder eben bei einer `standalone`-Angabe. Mit dem Attribut `cdata-section-elements` können Elemente namentlich benannt werden, die als CDATA-Sections ausgegeben werden sollen. Beispielsweise kann mit

```
cdata-section-elements="code"
```

festgelegt werden, daß das Element

```
<code> a &lt; b </code>
```

als

```
<![CDATA[ a < b ]]>
```

ausgegeben wird.

Methode html: Diese Angabe erzeugt standardmäßig HTML-4.0-konforme Ausgaben. Hierbei wird vor allem die Eigenart einiger nicht wohlgeformter HTML-Elemente unterstützt, wie z.B. `
`, `<hr>`, `` und einige andere. Die Elemente `<script>` und `<style>` werden bei Ausgabe so behandelt, daß die öffnende spitze Klammer erhalten bleibt und nicht in `<` umcodiert wird. Die href- oder src-Attribute werden erkannt und darin enthaltene Leerzeichen entsprechend als "%20" ausgegeben. Speziell zur Unterstützung älterer Browser werden einige HTML-Elemente in abgekürzter Form (`<OPTION SELECTED>` statt `<OPTION SELECTED="selected">`) ausgegeben. Die Attribute `cdata-section-elements`, `omit-xml-declaration` und `standalone` sind bei HTML-Output nicht anwendbar. Die Attribute `doctype-system` und `doctype-public` erzeugen eine Deklaration mit Namen "html".

Methode text: Diese Methode bewirkt, daß lediglich die Text-Knoten entsprechend dem Zeichensatz, der im encoding-Attribut angegeben wurde, in die Ausgabedatei geschrieben werden.

1.4.10 `<xsl:preserve-space>` und `<xsl:strip-space>`

Die Top-Level-Elemente dienen dazu, die Behandlung von Weißraum im Eingabedokument zu steuern.

```
<xsl:preserve-space elements="Liste von Elementnamen" />
<xsl:strip-space elements="Liste von Elementnamen" />
```

Ohne diese Anweisungen bleiben leere Text-Knoten im Eingabedokument bei Verarbeitung und Ausgabe erhalten. Die Anweisung `<xsl:preserve-space>` muß deshalb nur verwendet werden, wenn mit `<xsl:strip-space>`-Elemente benannt wurden, die ausgesondert werden sollen, wenn sie nur aus Leerraum bestehen. Als Weißraum (`whitespace`) werden folgende Zeichen behandelt: Leerzeichen (`space`), Tabulatorzeichen (`tab`), Zeilenschaltung (`carriage return` – Wagenrücklauf für die Menschen, die noch Schreibmaschinen kennen) und Zeilenvorschub (`linefeed`). Ein Text-Knoten, der nur Leerraum enthält, besteht ausschließlich aus solchen Zeichen. Nur auf solche Knoten wirken die Elemente.

Wird ein Element `<thema>`, welches nur Leerraum enthält, mit `<xsl:strip-space elements="thema" />` aus der Eingabe entfernt, kann es nicht von `<xsl:template>`-Anweisungen verarbeitet werden. Dies bedeutet, daß es nicht in die Ausgabe kopiert und damit auch nicht von eventuell vorhandenen `<xsl:number>`-Anweisungen erfaßt wird. Sprünge innerhalb der Numerierung mit `<xsl:number>` rühren oftmals von übernommenen Leerraum-Elementen her.

1.5 *Elemente zur Erzeugung von Variablen und Parametern*

1.5.1 `<xsl:variable>`

Die Anweisung wird verwendet, um eine benannte Variable zu erzeugen und ihr einen Wert zuzuweisen. Zur Definition einer globalen Variablen wird `<xsl:variable>` als Top-Level-Element verwendet. Für Variablen mit lokalem Gültigkeitsbereich kann die Anweisung jedoch auch im Rumpf anderer Anweisungen notiert werden.

```
<xsl:variable name="QName" select="Ausdruck">
  Anweisungsrumpf
</xsl:variable>
```

Die Angabe eines Namens ist zwingend, die Formulierung eines select-Attributes zur Bestimmung des Werts der Variablen jedoch optional. Wird das select-Attribut nicht verwendet, erzeugt der XSLT-Prozessor den Wert der Variablen aus dem Anweisungsrumpf. Wird das select-Attribut jedoch verwendet, kann der Anweisungsrumpf leer bleiben. Sind beide Teile leer, enthält die Variable eine leere Zeichenkette als Wert.

Im Unterschied zu Variablen in anderen Programmiersprachen ist es bei XSLT-Variablen nicht möglich, den einmal zugewiesenen Wert im Lauf des XSLT-Skripts zu ändern. XSLT-Variablen entsprechen damit eher den Konstanten anderer Programmiersprachen. Da dies sehr gewöhnungsbedürftig ist, sind einige Hersteller von XSLT-Prozessoren dazu übergegangen, Erweiterungen in ihre Produkte (z.B. Saxon) einzubauen, die genau dies erlauben, was die Spezifikation nicht vorsieht.

Um eine Zeichenkette einer Variablen zuzuweisen, ist darauf zu achten, daß je nach Art der Zuweisung auf die korrekte Verwendung der Hochkommata geachtet werden muß. Äquivalent und richtig sind folgende Anweisungen:

```
<xsl:variable name="buchtitel" select="'Web-Design'" />
<xsl:variable name="buchtitel" select='"Web-Design"' />
<xsl:variable name="buchtitel">Web-Design</xsl:variable>
```

Wird der Wert im Anweisungsrumpf codiert, wird die Angabe in eine Zeichenkette konvertiert. Notiert man den Wert innerhalb des select-Attributs muß gekennzeichnet werden, daß es sich um eine Zeichenkette handelt, weil der Wert ansonsten als XPath-Ausdruck interpretiert wird.

```
<xsl:variable name="buchtitel" select="Web-Design" />
```

Diese Variable hat als Wert einen XPath-Ausdruck, der eine Auswahl von Knoten mit dem Namen Web-Design selektiert. Enthält eine Variable einen Selektionsausdruck oder eine Berechnung, kann sie ähnlich einer SQL-View benutzt werden. Wobei jedoch die Restriktionen des Geltungsbereichs berücksichtigt werden müssen.

Ungewöhnlich ist auch der Geltungsbereich (*scope*) von Variablen innerhalb des Stylesheets. Die `<xsl:variable>`-Anweisungen, die als Top-Level-Elemente formuliert sind, haben globale Gültigkeit, die sich auch auf Stylesheets erstreckt, die mit `<xsl:include>` oder `<xsl:import>` eingebunden wurden. Globale Variable können deshalb bereits vor ihrer Deklaration verwendet werden.

Variablen, die innerhalb von Schablonen definiert werden, sind lediglich lokal gültig, d.h., nur innerhalb des Elements, in dem sie definiert werden. Ihre Gültigkeit erstreckt sich auf die Nachfolger-Achse, nicht jedoch auf die der eigenen Nachfahren (*descendants*)! Die häufigsten Probleme mit Variablen in Stylesheets rühren von Mißverständnissen her, die den Gültigkeitsbereich betreffen. Ein Problem entsteht auch bei der Definition von Variablen gleichen Namens, sofern sich ihr Gültigkeitsbereich überschneidet.

1.5.2 <xsl:param>

Mit <xsl:param> kann ein benannter Parameter erzeugt und diesem ein Wert zugewiesen werden. Die Anweisung kann als Top-Level- oder als erstes Kind-Element innerhalb einer <xsl:template>-Anweisung verwendet werden.

```
<xsl:param name="QName" select="Ausdruck">
    Anweisungsrumpf
</xsl:param>
```

Die Angabe eines Namens ist zwingend, die Formulierung eines select-Attributes zur Bestimmung des Werts optional.

Parameter sind den Variablen sehr ähnlich, mit dem Unterschied, daß sie ihren Wert beim Eintritt in ihren Gültigkeitsbereich von einer anderen Instanz oder aus einem anderen Element erhalten können, dies also nicht nur innerhalb der Deklaration möglich ist. Dennoch sind sie ebenfalls keine Variablen im Sinn üblicher Programmiersprachen, deren Wert beliebig oft gesetzt werden kann.

Globale Parameter enthalten ihren Wert eventuell als Übergabeparameter beim Aufruf des Stylesheets. Da die XSLT-Spezifikation die Art der Realisierung den Anbietern von XSLT-Prozessoren überläßt, muß die Art, wie globale Parameter ihren Wert erhalten, im jeweiligen Dokument zum Produkt nachgesehen werden. Lokale Parameter empfangen ihren Wert durch ein <xsl:with-param>-Kind-Element des Eltern-Elements.

1.5.3 <xsl:with-param>

Die Anweisung kann benutzt werden, um Parameterwerte an Schablonen weiterzugeben, die durch <xsl:call-template> oder <xsl:apply-templates> aktiviert wurden.

```
<xsl:with-param name="QName" select="Ausdruck">
    Anweisungsrumpf
</xsl:with-param>
```

Die Angabe des name-Attributs ist zwingend, das select-Attribut optional. Die Anweisung kann nur als Kind-Element von <xsl:call-template> oder <xsl:apply-templates> benutzt werden. Innerhalb des aufgerufenen Templates muß ein Parameter existieren, dessen Namen mit dem in <xsl:with-param> angegebenen übereinstimmt. Ist kein solcher Parameter vorhanden, wird die Anweisung ignoriert, eine Referenz auf diesen Parameter innerhalb der Schablone führt dann zu einer Fehlermeldung.

1.6 Elemente zur Steuerung bedingter Verarbeitung

1.6.1 <xsl:if>

Die Anweisung prüft eine Bedingung. Ergibt die Auswertung "true" wird der Anweisungsrumpf (template body) ausgeführt. Evaluiert die Anweisung zu "false", wird keine Aktion angestoßen, da ein else-Teil, wie er in anderen Programmiersprachen bekannt ist, fehlt.

```
<xsl:if test="Ausdruck" >
```

```
        Anweisungsrumpf
    </xsl:if>
```

Die Angabe eines Ausdrucks im test-Attribut ist zwingend. Die Auswertung des XPath-Ausdrucks wird zu einem booleschen Wert konvertiert. Ergibt die Auswertung eine Knotenauswahl (*node set*), eine Zeichenkette oder ein Fragment des Ergebnisbaums (*result tree fragment*), die nicht leer sind, evaluiert der Ausdruck zu "true". Ist das Ergebnis numerisch, werden alle Werte außer 0 zu "true". In folgendem Beispiel wird ein Komma ausgegeben, solange der aktuell verarbeitete Knoten nicht der letzte ist:

```
<xsl:for-each select="thema" >
    <xsl:if test="not(position()=last)" >, </xsl:if>
</xsl:for-each>
```

1.6.2 <xsl:choose>,<xsl:when>,<xsl:otherwise>

Um unter einer Anzahl von Alternativen eine Auswahl zu treffen, kann <xsl:choose> verwendet werden.

```
<xsl:choose>
    <xsl:when test="Ausdruck">+
    <xsl:otherwise>
</xsl:choose>
```

Die Anweisung <xsl:choose> besitzt keine Attribute und kann eine oder mehrere <xsl:when>- und höchstens eine einzige <xsl:otherwise>-Anweisung besitzen, die jedoch lediglich als letzte in der Aufzählung erscheinen darf.

Der erste test-Ausdruck eines <xsl:when>-Elementes, der zu true evaluiert, wird ausgewertet. Alle nachfolgenden <xsl:when>-Anweisungen werden ignoriert, unabhängig davon, ob ihr test-Ausdruck ebenfalls zu true ausgewertet würde. Besitzt kein <xsl:when>-Element einen Ausdruck, der zu true evaluiert, wird <xsl:otherwise> ausgewählt. Ist dieses nicht vorhanden, bewirkt die gesamte <xsl:choose>-Anweisung nichts. Eine <xsl:choose>-Anweisung, die lediglich ein einzelnes <xsl:when>-Kind-Element enthält und keine <xsl:otherwise>-Anweisung, entspricht einer <xsl:if>-Anweisung.

1.6.3 <xsl:for-each>

Mit <xsl:for-each> werden die Verarbeitungangaben des Anweisungsrumpfes auf eine Menge von ausgewählten Knoten angewendet.

```
<xsl:for-each select="Ausdruck">
    <xsl:sort>
    Anweisungsrumpf
</xsl:for-each>
```

Die Angabe des select-Attributs ist zwingend, der select-Ausdruck muß eine Auswahl von Knoten (*node set*) als Auswertungsergebnis erzeugen. Diese Auswahl wird mit Hilfe von <xsl:for-each> durchlaufen, wobei die Anweisungen im Anweisungsrumpf einmal für jeden Knoten ausgeführt werden.

Nachfolgendes Beispiel selektiert alle <thema>-Knoten des Dokuments, kopiert das jeweilige Element, erzeugt ein nummer-Attribut und weist diesem die aktuelle Position innerhalb der Selektion zu:

```
<xsl:for-each select="//thema">
  <xsl:copy>
    <xsl:attribute name="nummer">
      <xsl:value-of select="position()" />
    </xsl:attribute>
  </xsl:copy>
</xsl:for-each>
```

Wird keine Sortieranweisung eingegeben, werden die Knoten in Dokumentanordnung verarbeitet. Ist ein <xsl:sort>-Element notiert, werden die ausgewählten Knoten sortiert, bevor sie verarbeitet werden.

1.7 Elemente für Sortierung und Numerierung

1.7.1 <xsl:sort>

Mit <xsl:sort> wird die Reihenfolge bestimmt, in der die Knoten verarbeitet werden, die durch eine <xsl:for-each>- oder eine <xsl:apply-templates>-Anweisung ausgewählt wurden.

```
<xsl:sort
  select="Ausdruck"
  order={"ascending" | "descending"}
  case-order={"upper-first" | "lower-first"}
  lang="Ländercode"
  data-type={"text" | "number" }
/>
```

Die Anweisung <xsl:sort> muß am Beginn des Anweisungsrumfies von <xsl:for-each> notiert werden und kann bei <xsl:apply-templates> vor den <xsl:with-param>-Angaben stehen. Es können auch mehrere <xsl:sort>-Anweisungen ausgegeben werden, die in der Reihenfolge ihrer Notation den 1., 2., n-ten Sortierschlüssel definieren.

Alle Attribute sind optional. Das select-Attribut beschreibt den Sortierschlüssel. Der ausgewertete Ausdruck wird in eine Zeichenkette konvertiert. Die Behandlung dieser Zeichenkette kann durch die weiteren Attribute festgelegt werden. Mit dem order-Attribut wird aufsteigende (ascending) oder absteigende (descending) Sortierreihenfolge festgelegt, mit case-order die Behandlung von identischen Zeichen, die jedoch in unterschiedlicher Groß- und Kleinschreibung auftreten. Die Sortierung der Grossbuchstaben an vorderer Stelle beschreibt der Attributwert "upper-first". Entsprechend werden Kleinbuchstaben nach vorne sortiert, wenn "lower-first" gesetzt ist. Das Attribut lang soll den Eigenheiten einer jeden Sprache bei der Sortierung Rechnung tragen. Leider ist die Sortierung innerhalb eines Sprachraums nicht einheitlich, sondern wechselt von Katalog zu Katalog und ist historischen Schwankungen unterworfen.

Wird bei data-type "number" angegeben, erfolgt eine weitere Konvertierung des ausgewerteten Ausdrucks in eine Zahl.

Nachfolgendes Beispiel führt für das Folien-Beispiel zwei Sortierdurchgänge aus. Im ersten Durchgang wird anhand der Zeichenfolge des select-Attributs sortiert, im zweiten Durchgang anhand der numerischen Werte des ausgewerteten select="position()" -Ausdrucks.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

<xsl:template match="/">
  <themenliste>
    <xsl:comment>
      Nach Themen alphabetisch aufsteigend
    </xsl:comment>
    <xsl:for-each select="//thema">
      <xsl:sort order="ascending"/>
      <xsl:call-template name="thema-out" />
    </xsl:for-each>

    <xsl:comment>
      Jetzt in umgekehrter Dokumentordnung
    </xsl:comment>
    <xsl:for-each select="//thema">
      <xsl:sort select="position()"
        data-type="number" order="descending"/>
      <xsl:call-template name="thema-out" />
    </xsl:for-each>
  </themenliste>
</xsl:template>

<xsl:template name="thema-out">
  <thema>
    <xsl:attribute name="position">
      <xsl:value-of select="position()" />
    </xsl:attribute>
    <xsl:value-of select="." />
  </thema>
</xsl:template>
</xsl:stylesheet>
```

1.7.2 <xsl:number>

Die Anweisung dient zur Erzeugung und Formatierung einer Zählnummer. Sie gibt als Ergebnis ihrer Berechnung eine formatierte Zeichenkette als Text-Knoten aus.

```
<xsl:number level="any" | "multiple" | "single"
  count="Muster"
  from="Muster"
  value="Ausdruck"
  format="Format Zeichenfolge"
  lang="Ländercode"
  letter-value="alphabetic" | "traditional"
  grouping-separator="Zeichen"
```

```
grouping-size="Zahl" />
```

Die Angabe der Attribute ist optional, Implementierungsdetails sind in vielen Fällen den Herstellern der XSLT-Prozessoren überlassen. Das Zusammenspiel der Attribute ist komplex.

Die Anweisung ermittelt die Nummer der Knoten des Quellbaums. Als nicht weiter spezifizierte Anweisung `<xsl:number />` wird jedem Knoten die Positionsnummer innerhalb seiner Geschwister-Knoten zugeordnet. Die Art der Numerierung kann durch die Attribute `level`, `count`, `from` und `format` gesteuert werden. Standardwert für `level` ist "single", d.h., es wird eine einfache Zählnummer ausgegeben. Wird der Wert des Attributs dagegen auf "multiple" gesetzt, erfolgt die Numerierung hierarchisch entsprechend der Knotenhierarchie (1., 1.1, 1.2,...). Handelt es sich bei den Knoten, deren Nummer verwendet werden soll, um hierarchisch nicht lückenlos geschachtelte Elemente, treten Probleme auf. Im Beispieldokument des Buches enthält jedes `<folie>`-Element ein `<titel>`-Element und ein `<themenliste>`-Element, die Themenpunkte `<thema>` sind wiederum als Kind-Elemente zu `<themenliste>` gestaltet, d.h., hierarchisch ist eine Stufe zwischen `<titel>` und `<thema>`. Eine hierarchische Numerierung, die eine durchlaufende Nummer für den Titel vergibt und auf der nächsten Stufe eine Zählnummer für alle zugehörigen Themen, ist mit diesen Mitteln nicht möglich. Folgendes Template

```
<xsl:template match="titel | thema">
  <eintrag>
    <xsl:number level="multiple" format="1.1"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </eintrag>
</xsl:template>
```

gibt Einträge dieser Form aus:

```
<eintrag>1.1.1 Folienübersicht</eintrag>
<eintrag>1.1.2.1 Was muss ein CMS können?</eintrag>
<eintrag>1.1.2.2 Wer sollte ein CMS einsetzen?</eintrag>
```

Es ist deutlich, daß die übergeordnete oder zwischen den Elementen liegende Hierarchiestufen, die nicht von Interesse sind auch in die Numerierung einbezogen werden. Das Setzen des Attributs `from="titel"` führt lediglich dazu, daß die Zählung mit diesem Element erneut beginnt und die Titel-Elemente keine Nummer tragen, alle anderen jedoch in obiger Weise behandelt werden.

Etwas vorhersagbarer werden die Resultate, wenn es nicht dem XSLT-Prozessor überlassen wird, eine Nummer zu finden, sondern das `value`-Attribut verwendet wird, um der Anweisung eine Zahl zuzuweisen (beispielsweise mit der `position()`-Funktion), die dann lediglich noch formatiert werden muß. Das `format`-Attribut erlaubt die Angabe einer Zeichenkette, welche die Zahlendarstellung beschreibt. Mit `format="I"` kann römische, mit `format="a"` alphabetische Numerierung erzeugt werden.

1.8 Weitere Elemente

1.8.1 <xsl:decimal-format>

Die Anweisung steuert die Darstellung dezimaler Zahlen, die durch die Funktion `format-number()` ausgegeben werden. Die Anweisung wirkt nicht auf `<xsl:number>`!

```
<xsl:decimal-format
  name="QName"
  decimal-separator="Zeichen"
  grouping-separator="Zeichen"
  infinity="Infinity"
  minus-sign="Zeichen"
  NaN="Zeichenfolge"
  percent="Zeichen"
  per-mille="Zeichen"
  zero-digit="Zeichen"
  digit="Zeichen"
  pattern-separator="Zeichen" />
```

Die durchweg optionalen Attribute beschreiben Zeichen und Zeichenfolgen, die als Dezimaltrennzeichen (`decimal-separator`), Nullzeichen, Hinweis für nicht numerische Angabe (NaN – Not a Number) etc. festlegbar sind. Die Benutzung des `name`-Attributs ermöglicht es, innerhalb eines `format-number()`-Aufrufs eine Dezimalformat-Spezifikation anzusprechen.

1.8.2 <xsl:key>

Mit dem Top-Level-Element wird ein Suchschlüssel definiert, der mit der `key()`-Funktion verwendet werden kann.

```
<xsl:key name="QName" match="Muster" use="Ausdruck" />
```

Die Angabe aller Attribute ist verpflichtend. Mit dem `name`-Attribut wird der Name festgelegt, unter dem der Suchschlüssel innerhalb der `key()`-Funktion angesprochen werden kann. Mit `match` wird ein Suchmuster angegeben, welches die auszuwählende Knotenmenge beschreibt, und das `use`-Attribut spezifiziert den Wert der gesuchten Knoten. Folgendes Beispiel definiert einen Suchschlüssel für Folien. Die Selektion aller `<folien>`-Elemente lässt sich über die Angabe des `use`-Attributs eingrenzen auf solche, deren `ident`-Attribut dem Wert entspricht.

```
<xsl:key name="getfolie" match="folie" use="@ident" />
```

Die `key()`-Funktion kann diese Definition wie folgt verwenden:

```
<xsl:value-of select="key('getfolie', 'zugriff')/titel" />
```

Mit Hilfe der `<xsl:value-of>`-Anweisung wird der Titel der Folie mit dem `ident`-Attribut "zugriff" selektiert. Natürlich könnte der Zugriff auf diesen Knoten auch mit Hilfe eines XPath-Ausdrucks realisiert werden:

```
<xsl:value-of select="//folie[@ident='zugriff']/titel" />
```

Hersteller von XSLT-Prozessoren sind jedoch gehalten, einen Index für die Datei aufzubauen, wenn mit `<xsl:key>` ein Suchschlüssel definiert wurde. Der Zugriff mit der `key()`-Funktion sollte somit schneller sein, als mit einem normalen XPath-Ausdruck.

Die beste Zugriffsgeschwindigkeit bietet sicher die `id()`-Funktion. Sie setzt jedoch das Vorhandensein einer DTD voraus, welche auch wirklich id-Attribute mit dem XML-Datentyp ID verbindet. Die `key()`-Funktion ist also auch als Ersatz für die `id()`-Funktion gedacht, da XSLT-Verarbeitung auch ohne DTDs auskommt und in der Praxis auch häufig so durchgeführt wird. Ist eine entsprechende DTD vorhanden, wird die `id()`-Funktion sicher der Bequemlichkeit und der Schnelligkeit halber vorzuziehen sein. Kann aber nicht davon ausgegangen werden, daß eine DTD vorhanden ist, sollte die `key()`-Funktion verwendet werden.

1.8.3 `<xsl:message>`

Die Anweisung gibt einen Hinweis aus und beendet eventuell die Verarbeitung des Stylesheets.

```
<xsl:message terminate="yes" | "no">  
    Anweisungsrumf  
</xsl:message>
```

Die optionale Angabe des `terminate`-Attributs mit "yes" führt zur Beendigung der Stylesheet-Verarbeitung, Standardwert ist "no". Die Anweisung gibt den im Anweisungsrumf erzeugten Text als Nachricht aus. Ob dies in einer Dialogbox am Bildschirm oder in einer Log-Datei geschieht, ist nicht festgelegt. Das Verhalten dieser Anweisung muß also den Angaben des Herstellers des XSLT-Prozessors entnommen werden.

1.8.4 `<xsl:fallback>`

Die Anweisung beschreibt, was geschehen soll, wenn der Schablonenrumf, in dem sie notiert ist, nicht ausgeführt werden kann.

```
<xsl:fallback>  
    Anweisungsrumf  
</xsl:fallback>
```

Die Anweisung berücksichtigt, daß es in zukünftigen Versionen von XSLT Neuerungen und Erweiterungen geben kann, die noch nicht von den vorhandenen XSLT-Prozessoren unterstützt werden. In diesem Anwendungsfall kann mit `<xsl:fallback>` beschrieben werden, was an Stelle der noch nicht unterstützten Anweisungen getan werden soll. Ebenso kann ein Stylesheet, das herstellerabhängige Erweiterungen nutzt, mit dieser Anweisung Hinweise geben, wenn es in anderen Umgebungen nicht lauffähig sein sollte.

1.8.5 `<xsl:namespace-alias>`

Mit dem Top-Level-Element `<xsl:namespace-alias>` kann festgelegt werden, daß ein Namensraumkürzel, das innerhalb des Stylesheets notiert ist, bei der Ausgabe auf das Zieldokument in eine andere Namensraumdefinition umgesetzt wird. Das ist beispielsweise dann notwendig, wenn eine Transformation ein XSL-

Stylesheet erzeugen soll, der xsl: Namensraum also nicht direkt als literales Ergebnis-Element genutzt werden kann.

```
<xsl:namespace-alias stylesheet-prefix="Präfix"
    result-prefix="Präfix" />
```

Die Angabe beider Attribute ist zwingend. Es ist nicht definiert, was bei der Änderung mit dem Namensraumkürzel geschieht. Die Anweisung greift in die Serialisierung des Ausgabebaums auf die Ausgabedatei ein. Einige Prozessoren behalten das stylesheet-prefix bei und ordnen bei der Ausgabe lediglich den Namensraum, der dem result-prefix entspricht, dem stylesheet-prefix zu (so z.B. Xalan und XT).

Das nachfolgende Stylesheet erzeugt zu einer beliebigen XML-Datei ein Stylesheet-Skelett und verwendet dazu das Namensraumkürzel "out", um den XSLT-Prozessor nicht zu verwirren. Es enthält für alle Elementnamen eine <out:template>-Anweisung, innerhalb dieser wiederum für jedes Kind-Element einen <out:apply-templates>-Eintrag. Hat das Element keine weiteren Kind-Elemente, wird ein <out:value-of select="."> erzeugt. Das Namespace-Präfix in der ausgegebenen Stylesheet-Datei ist dann eben 'out:'. Für die Funktionalität des erzeugten Stylesheets spielt es keine Rolle, ob das Namensraumkürzel "out" beibehalten wird oder nicht, solange diesem der richtige XSLT-Namensraum zugeordnet ist.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:out="test.xsl">
    <xsl:namespace-alias stylesheet-prefix="out"
        result-prefix="xsl"/>

    <xsl:template match="/">
        <out:stylesheet version="1.0">
            <xsl:for-each select="//*">
                <xsl:variable name="nm" select="name()" />
                <xsl:if
                    test="not(preceding::node()[name()=$nm])">
                    <out:template>
                        <xsl:attribute name="match">
                            <xsl:value-of select="name()" />
                        </xsl:attribute>
                        <xsl:call-template name="get-subelements" />
                    </out:template>
                </xsl:if>
            </xsl:for-each>
        </out:stylesheet>
    </xsl:template>
    <xsl:template name="get-subelements">
    <xsl:choose>
        <xsl:when test="not(*)">
            <out:value-of select="." />
        </xsl:when>
        <xsl:otherwise>
```

```

        <xsl:for-each select="*">
            <xsl:variable name="nm" select="name()" />
            <xsl:if
test="not(preceding::node()[name()=$nm])">
                <out:apply-templates>
                    <xsl:attribute name="select">
                        <xsl:value-of select="name()" />
                    </xsl:attribute>
                </out:apply-templates>
            </xsl:if>
        </xsl:for-each>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>

```

Um zu vermeiden, daß häufig vorkommende Elemente des Eingabedokuments für jedes Vorkommen eine `<out:template>`-Schablone erhalten, wird in obigem Stylesheet überprüft, ob der Knotenname bereits einmal vorgekommen ist. Initial wird mit

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

eine Menge aller Knoten erstellt und ein aktueller Knoten festgelegt. Sein Name wird in der Variablen `$nm` gemerkt und nachfolgend vor jeder Schreiboperation zur Prüfung verwendet, ob dieser Name in der Liste aller Vorgänger nicht bereits einmal vorkommt.

```
<xsl:if test="not(preceding::node()[name()=$nm])">
```

In diesem Fall wurde ja bereits eine Schablone für dieses Element angelegt. Ist ein Knoten dieses Namens bereits vorhanden, wird für ihn keine weitere Schablone erzeugt.

1.9 XSLT-Funktionen

1.9.1 current()

Die Funktion gibt ein Knotenmenge zurück, die nur den aktuellen Knoten (current node) enthält. In der Regel ist der Kontext-Knoten (context node) mit dem aktuellen Knoten identisch. Der Kontext-Knoten kann mit dem XPath-Ausdruck "." erfragt werden. Innerhalb eines normalen XPath Ausdrucks sind `current()` und "." identisch, innerhalb eines Prädikats jedoch nicht unbedingt. Denn bei Auswertung eines Prädikats kann sich der Kontext-Knoten verändern, der aktuelle Knoten jedoch unverändert bleiben.

Eine sehr gute Beschreibung dieses komplexen Sachverhalts findet sich bei M. KAY, XSLT Programmrs Reference, die nachfolgend in freier Übersetzung wiedergegeben wird:

„Der aktuelle Knoten (current node) wird wie folgt erzeugt:

Bei Auswertung einer globalen Variablen ist der aktuelle Knoten der Wurzel-Knoten (root node) des Quelldokuments.

Wird eine <xsl:apply-templates>-Anweisung ausgeführt, um eine Menge von Knoten zu verarbeiten, wird jeder einzelne der ausgewählten Knoten der Reihe

nach zum aktuellen Knoten. Wenn also eine Schablone aktiviert wird, ist der aktuelle Knoten immer derjenige, der dazu geführt hat, daß die Schablone ausgewählt wurde. Bei der Rückkehr aus der `<xsl:apply-templates>`-Verarbeitung wird der aktuelle Knoten auf seinen vorigen Wert zurückgesetzt. In ähnlicher Weise wird der Wurzel-Knoten des Quelldokuments zum aktuellen Knoten, wenn das System implizit eine Schablone aufruft, die den Wurzel-Knoten des Dokuments verarbeitet.

Wird `<xsl:for-each>` benutzt, um eine ausgewählte Knotenmenge zu verarbeiten, wird jeder einzelne der ausgewählten Knoten der Reihe nach zum aktuellen Knoten. Wenn die Verarbeitung der `<xsl:for-each>`-Schleife endet, wird der aktuelle Knoten auf seinen vorigen Wert zurückgesetzt.

Die Anweisungen `<xsl:call-template>` und `<xsl:apply-imports>` lassen den aktuellen Knoten unverändert.

Der aktuelle Knoten verändert sich – im Unterschied zum Kontext-Knoten (*context node*) nicht – wenn ein Prädikat innerhalb eines XPath-Ausdrucks ausgewertet wird.

Der Kontext-Knoten wird durch den XPath-Ausdruck "." zurückgegeben. Innerhalb eines einfachen XPath-Ausdrucks ergeben die Auswertungen von "." und `current()` denselben Ergebniswert. Werden sie jedoch innerhalb eines Prädikats verwendet, sind die Ergebniswerte in der Regel unterschiedlich.“ (S.437).

1.9.2 document()

Die `document()`-Funktion öffnet ein externes Dokument, baut einen Eingabebaum auf und gibt den Wurzel-Knoten zurück. Hauptanwendungsgebiet dieser Funktion ist es, weitere XML-Dokumente während einer Transformation zusätzlich zum Standard-Eingabedokument zu öffnen. Beispielsweise kann ein Verweis auf eine Datei folgendermaßen aussehen:

```
<page id="Einstieg" href="xml_einstieg.xml" />
```

Bei Verarbeitung der Quelldatei, welche diesen Eintrag enthält, kann mit

```
<xsl:apply-templates select="document(@href)" />
```

das Dokument geöffnet werden, und die Verarbeitung der Elemente dieser Datei wird durch die Schablonen des aktuellen Stylesheets angestoßen. Sind in der geöffneten Datei Tags enthalten, die den selben Namen tragen, wie Elemente der Standard-Quelldatei, werden sie dementsprechend von den definierten Schablonen des Stylesheets verarbeitet. Sollen jedoch je nach Herkunft andere Schablonen verwendet werden, kann dies z.B. durch Einsatz des `mode`-Attributs beim `<xsl:apply-templates>` Aufruf geschehen.

Es wird davon ausgegangen, daß sich die Datei im selben Verzeichnis befindet wie die Stylesheet-Datei, in der die `document()`-Funktion notiert ist. Andere Verzeichnisse können jedoch angegeben werden. Möglich ist auch die Angabe von Zeichenketten wie z.B.

```
document('xml_einstieg.xml')  
document('../texte/xml_einstieg.xml')
```

Der Aufruf `document('')` gibt den Wurzel-Knoten des aktuellen Stylesheets zurück. Ein Stylesheet könnte also auch sich selbst analysieren.

1.9.3 element-available()

Diese Funktion wird benutzt, um festzustellen, ob ein XSLT-Element verfügbar ist oder nicht. Derzeit betrifft dies lediglich Erweiterungs-Elemente, die Hersteller in XSLT-Prozessoren einbauen; `element-available()` wird aber im Falle zukünftiger Versionen verwendet werden können, um zu überprüfen, ob der aktuelle Prozessor ein Element unterstützt oder nicht. Der Ausdruck

```
element-available('xsl:apply-templates')
```

sollte `true` ergeben, ebenso wie die Prüfung der anderen Elemente, die in XSLT-Version 1.0 definiert wurden.

1.9.4 format-number()

Die Funktion `format-number()` dient zur Konvertierung von numerischen Werten in Zeichenketten. Die Formatierung der erzeugten Zeichenkette wird durch `<xsl:decimal-format>` kontrolliert. Es gibt keine Verbindung und keine Wirkung von Seiten des Elements `<xsl:number>`. Als Aufrufparameter werden Wert und Formatmuster angegeben. Ein Beispiel illustriert das am einfachsten:

```
<xsl:value-of  
  select="format-number(76542.127, '#,##0.00 DM')"/> />
```

Diese Anweisung produziert die Ausgabe 76.542,13 DM. Ist der Wertparameter kein numerischer Wert, sondern eine Zeichenkette, wird diese zunächst in einen numerischen Wert konvertiert. Das Formatmuster muß mit den amerikanischen Separatoren für Tausender bzw. Dezimaltrennzeichen notiert werden, es sei denn innerhalb einer benannten `<xsl:decimal-format>`-Anweisung wird etwas anderes vorgegeben. Die Verknüpfung zu einer solchen Anweisung geschieht über einen dritten möglichen Parameter beim Aufruf von `format-number()`. Hier kann der Name einer `<xsl:decimal-format>`-Anweisung angegeben werden, die andere Separatoren etc., festlegt.

Beschreibt das Formatmuster weniger Dezimalstellen, als der Wertparameter besitzt, wird auf die Dezimalstellen des Formatmusters gerundet. Der Aufbau des Formatmusters wird durch folgende spezielle Zeichen aufgebaut:

- # eine Ziffer, Nullen werden nicht ausgegeben
- 0 eine Ziffer, Nullen erscheinen als 0
- , Gruppierungsseparator
- Negativkennzeichen
- % Zeigt einen Wert als Prozentwert an

Das Formatmuster besteht aus zwei Teilen, die durch den Strichpunkt ";" voneinander getrennt sind. Der linke Teil des Formatmusters beschreibt dabei den positiven Zahlenbereich, der rechte Teil den negativen Bereich.

1.9.5 function-available()

Ähnlich wie bei `element-available()` wird mit `function-available()` geprüft, ob die Funktion, deren Namen als Parameter angegeben wird, innerhalb des XSLT-Prozessors zur Verfügung steht.

1.9.6 generate-id()

Diese Funktion erzeugt eine Zeichenkette, die einen Knoten eindeutig identifiziert. Wird die Funktion für einen Knoten, z.B. im Laufe der Stylesheet-Verarbeitung, mehrfach aufgerufen, so wird für den identischen Knoten auch eine identische Zeichenkette erzeugt. Die Art der Erzeugung ist jedoch von der Implementierung der Funktion im jeweiligen Prozessor abhängig. Das bedeutet, daß es keinen Sinn macht, eine mit `generate-id()` erzeugte Zeichenkette in einen XML-Ausgabedatenstrom zu schreiben, der anschließend mit einem anderen XSLT-Prozessor weiterverarbeitet wird.

Wichtigstes Einsatzgebiet der Funktion ist die Erzeugung von Namen, die als HTML-Linkziele dienen sollen, und den zugehörigen Links. Beispielsweise kann in einem ersten Durchgang durch ein Eingabedokument eine Liste von Links erzeugt werden.

```
<xsl:for-each select="//thema" >
  <a href="#{generate-id(.)}">
    <xsl:value-of select="@titel"/>
  </a>
</thema>
```

Bei diesem Durchgang existiert das jeweilige `<a name>`-Element in der Ausgabedatei noch nicht. Es ist jedoch sicher gestellt, daß bei jedem späteren Durchgang durch den identischen Knoten mit `generate-id()` eine identische `id` erzeugt wird. Im Durchgang, der die Quelldaten dann darstellt, wird deshalb mit

```
<a name="{generate-id(.)}">
```

die entsprechenden Namensanker erzeugt werden.

Die `generate-id()`-Funktion hat keine Beziehung zum XML-Datentyp `ID`, der in einer DTD definiert werden kann. Weder die `id()`- noch die `key()`-Funktion kann verwendet werden, um einen Knoten wiederzufinden, dessen Kennung mit `generate-id()` erzeugt wurde. Es ist lediglich möglich, eine Variable mit dem Wert der `generate-id()`-Funktion zu belegen und diese Variable in einem XPath-Prädikat (z.B. `[generate-id()=$generated]`) als Vergleichswert gegen die `generate-id()`-Funktion zu prüfen.

1.9.7 key()

Diese Funktion ist eine Suchfunktion, die in der Lage ist, Knoten zu finden, die dem Wert eines benannten `<xsl:key>`-Elements entsprechen. Wurde mit der Top-Level-Anweisung

```
<xsl:key name="fsuche" match="folie" use="@ident" />
```

ein benannter Schlüssel mit dem Namen `fsuche` erzeugt, kann mit dem Funktionsaufruf

```
<xsl:value-of select="key('fsuche', 'hintergrund')"/>
```

die Folie, deren `ident`-Attribut der Zeichenkette `'hintergrund'` entspricht, gefunden werden. Im Unterschied zur `id()`-Funktion setzt die `key()`-Funktion nicht die Existenz einer DTD voraus. Die verwendeten `match`-Attribute müssen demnach auch nicht vom Datentyp `ID` sein.

1.9.8 system-property()

Die Funktion `system-property()` gibt lediglich Informationen über die Umgebung des XSLT-Prozessors aus. Mögliche Parameter sind derzeit nur `xsl:version`, `xsl:vendor` und `xsl:vendor-url`. Die XSLT-Version wird wie folgt erfragt:

```
<xsl:value-of select="system-property('xsl:version')"/>
```

1.9.9 unparsed-entity-uri()

Diese Funktion gibt den Dateinamen eines Entity zurück, das innerhalb einer DTD mit dem Typ NDATA definiert wurde. Als Parameter wird die Entity-Referenz des gesuchten Entity angegeben.

Neben der `id()`-Funktion ist dies eine der wenigen und beschränkten Möglichkeiten, Informationen aus DTDs innerhalb von XSLT-Stylesheets zu verwenden.

2 XPath-Funktionen

2.1 Funktionen, die sich auf Knotenmengen (*node sets*) beziehen

last() liefert die Anzahl der Knoten der aktuellen Menge.

count(node set) liefert die Anzahl von Knoten, die in der Menge der Knoten des Funktionsarguments enthalten sind.

position() liefert die Position des aktuellen Knotens innerhalb der aktuellen Knotenmenge.

id(ID) liefert einen Knoten in beliebiger Schachtelungstiefe aufgrund seiner ID. Die Funktion kann nur angewendet werden, wenn eine DTD existiert, in der das Attribut ID mit dem Datentyp ID definiert wurde.

local-name(node set) gibt den lokalen Teil des Namens eines Knotens zurück.

namespace-uri(node set) gibt die Namensraumbezeichnung des Knotens zurück.

name(node set) gibt den Namen des aktuellen Knotens zurück.

2.2 Funktionen, die sich auf Zeichenketten beziehen

string (object?) konvertiert ein beliebiges Objekt in eine Zeichenkette.

concat(zeichenkette, zeichenkette, zeichenkette) verbindet die angegebenen Zeichenketten zu einer einzigen und gibt diese zurück.

starts-with(zeichenkette1, zeichenkette2) prüft, ob `zeichenkette1` mit der Zeichenfolge `zeichenkette2` beginnt. Trifft dies zu, wird `true` zurückgegeben.

contains(zeichenkette1, zeichenkette2) gibt `true` zurück, wenn `zeichenkette2` in `zeichenkette1` enthalten ist.

substring-before(zeichenkette1, zeichenkette2) gibt den Teil von `zeichenkette1` zurück, der vor dem ersten Auftreten von `zeichenkette2` innerhalb von `zeichenkette1` steht. Beispielsweise gibt `substring-before("Hund und Katz", " und")` "Hund" zurück.

substring-after(zeichenkette1, zeichenkette2) gibt den Teil von zeichenkette1 zurück, der nach dem ersten Auftreten von zeichenkette2 innerhalb von zeichenkette1 steht. Beispielsweise gibt `substring-after("Hund und Katz", " und")` "Katz" zurück.

substring(zeichenkette, start, n) gibt den Teil von zeichenkette zurück, der an der Position start beginnt und n Zeichen lang ist. Wird n als Längenparameter nicht angegeben wird ab Startposition der gesamte Reste der Zeichenkette zurückgegeben.

string-length(zeichenkette) gibt die Anzahl der Zeichen in dieser Zeichenkette aus.

normalize-space(zeichenkette) entfernt führenden und abschließenden Weißraum (whitespace characters) und reduziert innerhalb der Zeichenkette mehrfach vorkommende Leerzeichen auf ein einzelnes.

translate(zeichenkette1, zeichenkette2, zeichenkette3) ersetzt alle Vorkommen von zeichenkette2 in zeichenkette1 durch zeichenkette3. So ersetzt `translate("1.1.2000", ".", "-")` alle Datumstrennpunkte durch einen Bindestrich.

2.3 Funktionen, die logische Werte erzeugen (XPath 4.3)

boolean(object) konvertiert das Argument in einen Wahrheitswert. Je nach Datentyp des Objekts geschieht folgendes:

- Zahlen gleich 0 werden zu false, alle anderen Werte zu true.
- Zeichenketten der Länge 0 werden zu false, sonst true.
- Ein leere Menge von Knoten wird zu false, sonst zu true.
- Ein Ergebnisbaumfragment wird zu false, wenn es keinerlei Text-Knoten enthält, sonst zu true.

not(logischer Ausdruck) gibt die Negation des logischen Ausdrucks zurück.

true() gibt true zurück.

false() gibt false zurück.

lang(zeichenkette) überprüft, ob das lang-Attribut des Kontext-Knotens mit der angegebenen Zeichenkette übereinstimmt.

2.4 Numerische Funktionen

number(object) konvertiert das Argument in einen numerischen Wert. Wird kein Argument angegeben, wird die Zeichenkettenrepräsentation (`string value`) des Kontext-Knotens konvertiert. Abhängig vom Typ des Arguments wird false zu 0, true zu 1. Läßt sich eine angegebene Zeichenkette nicht in eine Zahl umwandeln, wird NaN (`not a number`) zurückgegeben. Eine übergebene Menge von Knoten wird zunächst in eine Zeichenkette gewandelt und dann entsprechend behandelt. Dasselbe gilt auch für Ergebnisbaum-Fragmente.

sum(node set) berechnet die Summe aller numerischen Werte der angegebenen Knotenmenge. Bei der Ausführung der Funktion werden nichtnumerische Knoten in numerische konvertiert, sofern dies möglich ist.

floor(numerisch) gibt die größte Ganzzahl zurück, die kleiner oder gleich dem numerischen Wert des Arguments ist.

ceiling(numerisch) gibt die kleinste Ganzzahl zurück, die größer oder gleich dem numerischen Wert des Arguments ist.

$\text{ceiling}(1.2) = 2$

$\text{ceiling}(-1.2) = 1$

round(numerisch) rundet den Wert des Arguments auf die nächste Ganzzahl.