Folien zum Lehrmodul

XSLT, Teil 3

Lernziele:

- weitere Steuer- und Ausgabekommandos von XSLT kennen
- Variablenkonzept verstehen
- Verbund berechnen können, hier mit Variablen

Inhaltsverzeichnis

1 Ausgabe- und Steuerkommandos 5

1.1 Versuch einer Systematik und Übersicht 6
1.2 Das Kommando `xsl:copy-of` 11
1.3 Das Kommando `xsl:if` 12
1.4 Das Kommando `xsl:choose` 14
1.5 Attributwertschablonen 15
1.6 Das Kommando `xsl:attribute` 21
1.7 Das Kommando `xsl:element` 24

2 Variablen 26

2.1 Wertangabe in `select` 28
2.2 Wertangabe in innerer Schablone 30
2.3 Verbundbildung mit Variablen 32
2.4 Weiterverarbeitung des Verbundergebnisses 40

3 Mehrere Ein- und Ausgabedateien 41

3.1 Das Kommando `xsl:document` 42
3.2 Die XSLT-Funktion `document(...)` 45

1 Ausgabe- und Steuerkommandos

hier nur punktuell und informell dargestellt, Details (insb. weitere Parameter) und genaue Definition s. XSLT-Standard

Gemeinsamkeiten:

- sind entweder Steuerkommandos oder erzeugen direkt irgendwelche Teile des Ausgabebaums
- treten in Schablonen auf
- haben oft einen Inhalt, der wiederum eine Schablone darstellt, m.a.W. können Ausgabekommandos geschachtelt werden
- werden als Element mit Typ `xsl:.....` notiert

1.1 Versuch einer Systematik und Übersicht

1. Steuerkommandos

   `xsl:for-each` iteriert über eine Knotenmenge und ruft für jeden Knoten die innere Schablone auf

   `xsl:apply-templates` iteriert über eine Knotenmenge und wendet die jeweils zutreffende Transformationsregel an

   `xsl:if` bedingter Aufruf einer inneren Schablone

   `xsl:choose` verallgemeinertes `if`; mehrere nacheinander zu testende Bedingungen, ggf. eine `otherwise`-Alternative

2. Ausgabekommandos, die *Knoten verschiedener Typen* durch Kopieren vom Eingabebaum in den Ausgabebaum erzeugen:

   `xsl:copy` kopiert einen Knoten des Eingabebaums (den Kontextknoten, kein select-Parameter); Typ des Knotens ist beliebig

   `xsl:copy-of select='` *`XPath-Ausdruck`* `'` kopiert beliebig viele Teilbäume des Eingabebaums, Wurzeln der Teilbäume gemäß XPath-Ausdruck

3. Ausgabekommandos, die *Elementknoten* erzeugen:

`<...>`	direkte Angabe der öffnenden und schließenden *tags*
`xsl:element`	erzeugt einen Elementknoten; der Typ wird als Parameter angegeben und kann dynamisch berechnet werden

   ggf. auch `xsl:copy` und `xsl:copy-of`

4. Ausgabekommandos, die *Textknoten* erzeugen

`...`	direkte Angabe von Text
`xsl:text`	erzeugt den angegebenen Text; spezielle Möglichkeiten zur Behandlung von Leerraum
`xsl:value-of`	konvertiert einzelne Eingabeknoten oder ganze Teilbäume in textuelle Darstellung

ggf. auch `xsl:copy` und `xsl:copy-of`

5. Ausgabekommandos, die *Attributknoten* erzeugen

`xxx='...'`	direkte Angabe von Attributname und Wert im öffnenden *tags*; nur möglich, wenn auch das Element direkt angegeben wird ggf. variable Inhalte mit Attributwertschablonen
`xsl:attribute`	erzeugt einen Attributknoten; Attributname und Wert werden in Parametern angegeben und können dynamisch berechnet werden

ggf. auch `xsl:copy` und `xsl:copy-of`

1.2 Das Kommando `xsl:copy-of`

Syntax:

```
<xsl:copy-of select=' ... ' />
```

- keine innere Schablone
- Wenn der `select`-Parameter einen XPath-Ausdruck enthält, werden alle Treffer ausgegeben. Ausgegeben wird zu jedem Treffer eine *komplette Kopie des Teilbaums*, dessen Wurzel dieser Treffer ist.

1.3 Das Kommando `xsl:if`

Schema:

```
<xsl:if test=' boolean-expression ' >
  <!-- innere Schablone -->
</xsl:if>
```

Merkmale:

- hat einen Parameter `test` , der einen Booleschen Ausdruck enthält (Vorsicht mit < -Zeichen! müssen umcodiert / vermieden werden)
- Inhalt: innere Schablone
- kein "else"-Zweig möglich

Wirkung:

1. der im Parameter `test` enthaltene Booleschen Ausdruck wird ausgewertet
2. bei positivem Testergebnis wird die innere Schablone ausgeführt

   Kontextknoten für die innere Schablone: der gleiche (!) wie der der aufrufenden Schablone

1.4 Das Kommando `xsl:choose`

entspricht einer case- / switch-Verzweigung
Schema:

```
<xsl:choose>
   <xsl:when test=' boolean-expression ' >
     <!-- Content: template -->
   </xsl:when>
   ......
   ......
   <xsl:otherwise>
     <!-- Content: template -->
   </xsl:otherwise>
</xsl:choose>
```

1.5 Attributwertschablonen

Problem, falls Attributwerte keinen festen Wert erhalten, sondern *berechnet* werden sollen:

- keine inneren Elemente erlaubt
- daher Ausgabekommandos als `<xsl:... select=' ... ' />` Element nicht direkt im Inhalt eines Attributs erlaubt

direkte Angabe des Werts nur brauchbar, wenn der Wert immer gleich ist

Beispiel: die Telefonliste aus LM XSLT2 soll in folgende Form umgewandelt werden:

```
<Telefonliste>
  <Eintrag name='Meier' land='0049' vorwahl='0271'
    nummer='891234' />
  <Eintrag name='Schmitz' land='0049' vorwahl='0228'
    nummer='870887' />
</Telefonliste>
```

Attribut `land` kann direkt mit festem Wert angegeben werden:

```
<xsl:template match=' Eintrag '>
  <Eintrag land='0049' vorwahl='??????' ....  />
</xsl:template>
```

Die Werte der anderen Attribute hängen von anderen Knoten des Eingabebaums ab;

z.B. Wert von `nummer` ist Kopie des Inhalts des Elements `Telnr`

`xsl:value-of` kann man nicht benutzen:

```
<xsl:template match=' Eintrag '> <!-- FALSCH !!!!! -->
  <Eintrag
    vorwahl='<xsl:value-of select=" Telnr/@vorwahl " />'
    nummer ='<xsl:value-of select=" Telnr " />'
  />
</xsl:template>
```

ist falsch, weil ....

syntaktisch betrachtet: `<` ist in Attributwerten nicht erlaubt

umcodieren des `<` mit `<` nützt nichts: dann wird das `<` "wörtlich" genommen und kein Kommando interpretiert

von der Struktur des Transformationsdokuments her betrachtet: das Kommando müßte in der Schablone ein Kindelement des Attributs sein - generell nicht erlaubt

Attributwertschablonen:

Attributwertschablone = Ausdruck, der Knotenliste liefert, in geschweiften Klammern

Syntax: `{` *Ausdruck* `}`

Ausgabe: textuelle Darstellung des *ersten* (!) Knotens der Liste

Beispiel:

```
<xsl:template match=' Eintrag ' >
  <Eintrag name    ='{ name }'
           land    ='0049'
           vorwahl ='{ Telnr/@vorwahl }'
           nummer  ='{ Telnr }' />
</xsl:template>
```

Beispiel 2: nur 1 Attribut mit kompletter Telefonnummer gemäß Muster "[0049] 0271-7402611"

```
<xsl:template match=' Eintrag ' >
  <Eintrag name='{ name }'
        telefonnr='[0049] { Telnr/@vorwahl }-{ Telnr }' />
</xsl:template>
```

d.h. Attributwert wird durch *mehrere* Attributwertschablonen und feste Texte erzeugt

1.6 Das Kommando `xsl:attribute`

Anzuwenden, wenn auch *der Name des auszugebenden Attributs berechnet* werden soll oder wenn der Attributwert ziemlich komplex ist

Merkmale und Wirkung:

- `xsl:attribute` -Anweisung muß *vor* Anweisungen ausgeführt werden, die den Inhalt des Elements erzeugen
- hat einen Parameter `name` , der den Namen des zu erzeugenden Attributs angibt
- innere Schablone: berechnet Wert des Attributs
  Wert: Konkatenation aller erzeugten Text-Knoten

Beispiel 1: In einer Lehrveranstaltungsbeschreibung alle Durchführungen in einem einzigen Attribut zusammenfassen;

Bsp: `<DURCHFUEHRUNG semester='Meier:2007s; Koch:2008w;' />`

Lösung:

```
<xsl:template match=' DURCHFUEHRUNG ' />
<xsl:template match=' DURCHFUEHRUNG[1] ' >
  <DURCHFUEHRUNGEN>
    <xsl:attribute name='semester'>
      <xsl:for-each select=' ../DURCHFUEHRUNG ' >
        <xsl:value-of select=' @dozentId ' />:
        <xsl:value-of select=' @semester ' />;
      </xsl:for-each>
    </xsl:attribute>
  </DURCHFUEHRUNGEN>
</xsl:template>
```

Beispiel 2: `xsl:attribute`-Schablone, die Elemente und einen Kommentar enthält

```
<xsl:template match=" / " >
  <alles>
    <xsl:attribute name="x">
      <!-- test -->
      123<b>abc</b>456
    </xsl:attribute>
  </alles>
</xsl:template>
```

ergibt:

```
<alles x="&#10;      123abc456&#10;    "/>
```

1.7 Das Kommando `xsl:element`

Anzuwenden, wenn auch *der Name des auszugebenden Elements berechnet* werden soll

Merkmale und Wirkung:

- `xsl:element` -Anweisung statt öffnendem und schließenden Tag
- Parameter `name` gibt Namen des zu erzeugenden Elements an
- Parameter `namespace` deklariert Namensraumbezeichner für dieses Element
- innere Schablone: erzeugt Kinder des Elements

Beispiel (um des Beispiels willen): DURCHFUEHRUNG-Elemente bilden, die die dozentId im Elementnamen enthalten

```
<xsl:template match=' DURCHFUEHRUNG ' >
  <xsl:element
       name='DURCHFUEHRUNGvon{ @dozentId }'>
    <xsl:attribute name='semester'>
      <xsl:value-of select=' @semester ' />
    </xsl:attribute>
  </xsl:element>
</xsl:template>
```

2 Variablen

- Variable = im Prinzip Paar (Name, Wert)
- werden u.a. benötigt, um Gruppierungen / Aggregationen und Verbunde zu berechnen
- können in verschiedenen Kontexten benutzt werden (Suchbedingungen in Pfaden, Ausgabeanweisungen usw.)
- sehr viele Sonderfälle und Verhaltensvarianten → eher schlecht verständlich
- in XSLT Version 2 deutlich besser als in XSLT Version 1
- sind als top-level-Element zulässig, aber auch als Anweisung innerhalb von Schablonen
- finale Wertzuweisung bei der Deklaration, *keine erneute Wertzuweisung erlaubt!*

Deklaration:

```
<xsl:variable name='...'
              select=' ... ' >
  <!-- Content: template -->
</xsl:variable>
```

- Attribut `name` muß syntaktisch korrekten Namen enthalten
- Angabe des Werts *entweder* im Attribut `select` *oder* durch eine innere Schablone

Benutzung in der Form $*variablenname*

2.1 Wertangabe in `select`

`select` muß Ausdruck enthalten, der

1. eine Zeichenkette,
2. eine Zahl,
3. einen Booleschen Wert oder
4. eine Knotenmenge liefert

falls Knotenmenge: Knoten können als Ausgangspunkt von Navigationen dienen. Beispiele:

```
<xsl:value-of select=' $variablenname / lokalerPfad ' />
<element attribut='{ $variablenname / lokalerPfad }' />
```

Vorsicht: viele automatische Konversionen!

Beispiele:

```
<xsl:variable name='n1' select='2+3' />
<xsl:variable name='n2' select='"2+3"' />
<xsl:variable name='n3' select='$n1+3*4' />
<xsl:variable name='n4' select="'xxyyzz'" />
<xsl:variable name='n5' select="''" />
<xsl:variable name='n6' select='1>2' />
<xsl:variable name='n7' select='//@semester' />

<xsl:template match=' / '>    <out n1='{$n1}' n2='{$n2}'
  n3='{$n3}' n4='{$n4}' n5='{$n5}' n6='{$n6}' />
</xsl:template>
```

liefert:

```
<out n1="5" n2="2+3" n3="17" n4="xxyyzz" n5="" n6="false" />
```

2.2 Wertangabe in innerer Schablone

Beispiele:

```
<xsl:variable name='n8' >
  Dies ist ein <b>fetter</b> Text.
</xsl:variable>

<xsl:variable name='n9' >
  <xsl:for-each select='//@semester' >
    <xsl:value-of select='.' />
    <xsl:text>..</xsl:text>
  </xsl:for-each>
</xsl:variable>
```

Ergebnistyp: *result tree fragment*

Nutzungsmöglichkeiten eines *result tree fragments*:

- Ausgabe mit `xsl:value-of`: konvertiert zu einem Textknoten
- Ausgabe mit `<xsl:copy-of select='$xxx' />`: als komplette Kopie mit allen inneren Knoten
- können *nicht* als weitere Eingabe, Startpunkt von Navigationen o.ä. benutzt werden (in XSLT 1.0)

2.3 Verbundbildung mit Variablen

- Nachimplementierung eines Verbunds von Hand unter Benutzung von Variablen
- Bestimmung der Verbundpartner mit einem XPath-Ausdruck

Beispiel:

- wie bisher Lehrveranstaltungsdaten, die eine `dozentId` als "Fremdschlüssel" auf die Personendaten enthalten
- zusätzliche Personendaten

```
<FBINFO>
  <PERSONEN>
    <PERSON persId='Kelter' nachname='Kelter'
            vornameInit='U.' fachgr='PI'      />
    ....
  </PERSONEN>

  <LEHRVERANSTALTUNG>
    ....
    <VERANTWORTLICHER dozentId='Kelter' />
    ....
  </LEHRVERANSTALTUNG>
</FBINFO>
```

Aufgabe: Im Element `VERANTWORTLICHER` sollen innen der Name, Initialen und Fachgruppenzugehörigkeit eingetragen werden, Beispiel:

```
<FBINFO>
  ....
  <LEHRVERANSTALTUNG>
    ....
    <VERANTWORTLICHER dozentId='Kelter' >
       Kelter, Udo (PI)
    </VERANTWORTLICHER>
    ....
  </LEHRVERANSTALTUNG>
</FBINFO>
```

Lösungsstrategie: Verbund manuell wie folgt in drei Schritten implementieren

1. eine Variable mit dem Fremdschlüsselwert anlegen (`dzId`)
2. die Variable nutzen, um in der "Zielrelation des Fremdschlüssels" den zugehörigen Eintrag zu lokalisieren –
   in zweiter Variable (`dzElem`) Referenz auf diesen Eintrag speichern
3. von der Referenz in der zweiten Variablen aus zu den auszugebenden Daten navigieren

Lösungsausschnitt:

```
<xsl:template match=' VERANTWORTLICHER ' >
                                        <!-- Schritt 1 -->
  <xsl:variable name='dzId' select=' @dozentId ' />
                                        <!-- Schritt 2 -->
  <xsl:variable name='dzElem' select=
        ' // PERSON [ @persId = $dzId ] ' />
                                        <!-- Schritt 3 -->
  <VERANTWORTLICHER dozentId='{ $dzId }' >
    <xsl:value-of select=' $dzElem / @nachname ' />,
    <xsl:value-of select=' $dzElem / @vornameInit ' />
    (<xsl:value-of select=' $dzElem / @fachgr ' />)
  </VERANTWORTLICHER>
</xsl:template>
```

Anmerkungen:

- in Schritt 2 ist intuitiv naheliegend, aber *falsch*:

  `select=' // PERSON [ @persId = @dozentId ] '`

  Kontextknoten der Pfade `@persId` und `@dozentId` ist ein `PERSON`-Element, dort gibt es kein Attribut `@dozentId`!

  im absoluten Pfad `'//PERSON [..]'` ist die Position des aktuellen Vergleichs-Attributs `@dozentId` ohne die Variable `@dzId` mit den bisher eingeführten Konzepten nicht rekonstruierbar

- Variable `dzElem` enthält i.a. eine Menge von Referenzen auf Knoten im Eingabebaum, weil mit `select=' `*`pfad`*` '` gesetzt; wenn Daten korrekt, max. 1 Element.
- von dort aus weiternavigieren,
  Beispiel: `$dzElem / @nachname`

- Variable `dzElem` ist verzichtbar (verbessert aber die Lesbarkeit): jedes Auftreten von `$dzElem` in Schritt 3 kann ersetzt werden durch
  `select=' // PERSON [ @persId = $dzId ] '`
- in Schritt 3 wäre `dozentId='{ @dozentId }'` ebenfalls richtig,
- `dozentId='$dzId'` wäre falsch, das würde `$dzId` wörtlich ausgeben

Effizienzproblem: Implementierung des Pfadausdrucks `// PERSON [ @persId = $dzId ]`

- kann am einfachsten durch lineare Suche implementiert werden – sehr ineffizient
- könnte durch einen automatisch angelegten Sekundärindex beschleunigt werden

  Optimierung der Ausführung von Transformationen sehr komplex - entfällt / nicht voraussetzbar

  → manueller Einsatz von Sekundärindexen

2.4 Weiterverarbeitung des Verbundergebnisses

Beispiele:

- weiterer Verbund, z.B. `@fachgr` ist Referenz auf Daten der Fachgruppe, Ergänzung von Merkmalen der Fachgruppe
- Suche nach allen Lehrveranstaltungen einer bestimmten Fachgruppe

möglich, führt aber zu sehr komplizierten (fehleranfälligen) Lösungen

besser: temporäre XML-Datei

3 Mehrere Ein- und Ausgabedateien

bisher: genau 1 Ein- und Ausgabedatei - oft zu restriktiv

Beispiele:

- aus der `FBINFO`-Datei sollen für jede Lehrveranstaltung eine separate HTML-Datei erzeugt werden
- Eingabedaten sollen auf mehrere XML-Dateien verteilt werden, z.B. pro Fachgruppe eine separate XML-Datei

3.1 Das Kommando `xsl:document`

Schema:

```
<xsl:document
  method=".."
  href=".."
  encoding=".."
  .....
>
  <!-- innere Schablone -->
</xsl:document>
```

- erst ab XSLT 1.1 verfügbar
- erzeugt eine XML-/HTML-/Text-Datei gemäß Angabe in `@method` (Angaben wie in `xsl:output`-Elementen)
- Name der Ausgabedatei in `@href`

- bei `method="xml"`: alle Merkmale in der XML-Deklaration können durch weitere Attribute angegeben werden, Beispiel: `encoding="ISO-8859-1"`

Beispiel (Lösungsausschnitt):

```
<xsl:template match=' LEHRVERANSTALTUNG ' >
  <xsl:document
    method="html"
    href="{LEHRVERANSTALTUNGSKUERZEL}.html"
  >
    <html>
      <head> .......... </head>
      <body> .......... </body>
    </html>
  </xsl:document>
</xsl:template>
```

dazu passendes “Hauptprogramm”:

```
<xsl:template match=’ FBINFO ’ >
  <xsl:apply-templates select=’ LEHRVERANSTALTUNG ’ />
</xsl:template>
```

3.2 Die XSLT-Funktion `document(...)`

Schnittstelle (stark vereinfacht; vollständige Spezifikation s. `https://www.w3.org/TR/xslt-10/#document`):

```
node-set document(uri:string)
```

- im Argument `uri` wird eine XML-Datei angegeben
- die Datei wird eingelesen zu einem eigenen Syntaxbaum
- zurückgegeben wird eine Referenz auf die Dokumentwurzel dieses Syntaxbaums
- vor dort kann (wie bei einer Variablen) mit einem relativen Pfadausdruck weiternavigiert werden

Beispiel:

```
<xsl:output method="text" />
<xsl:template match=" / ">
  <xsl:document method="xml" href="/tmp/test.xml" >
    <a>
      <b x="1">eins</b>
      <b x="2">zwei</b>
      <b x="3">drei</b>
    </a>
  </xsl:document>

  <xsl:variable name="doc"
    select=" document('/tmp/test.xml') " />

  <xsl:value-of select=' $doc / a / b [ @x=2] ' />
  <xsl:value-of select=' $doc / a / b [ @x=3] ' />
</xsl:template>
```

- diese Transformation erzeugt (bei beliebiger Eingabe) die Text-Ausgabe “zweidrei”
- das Kommando `xsl:document` erzeugt die Datei `/tmp/test.xml`
- diese Datei kann sofort danach wieder eingelesen werden, die Dokumentwurzel wird hier einer Variablen zugewiesen

Nutzung von `document(...)` :

- aufsammeln von Daten aus verschiedenen Quellen
- Vorverarbeitung der Eingabedaten mit `xsl:document` , z.B. Verbundbildung,
  danach Wiedereinlesen dieser Daten mit `document(...)`