UNIVERSITÄT SIEGEN

Vorlesung Datenbanksysteme II

XSLT

Inhalt

- Transformation von XML-Dokumenten
- Grundlagen von XSLT
- XSLT-Prozessoren
- XSLT-Transformationsdokumente
- XSLT-Transformationsregeln
- Verwaltung von Transformationsregeln
- Ausgabe- und Steuerkommandos
- Indizes

UNIVERSITÄT SIEGEN

Transformation von XML-Dokumenten

Motivation

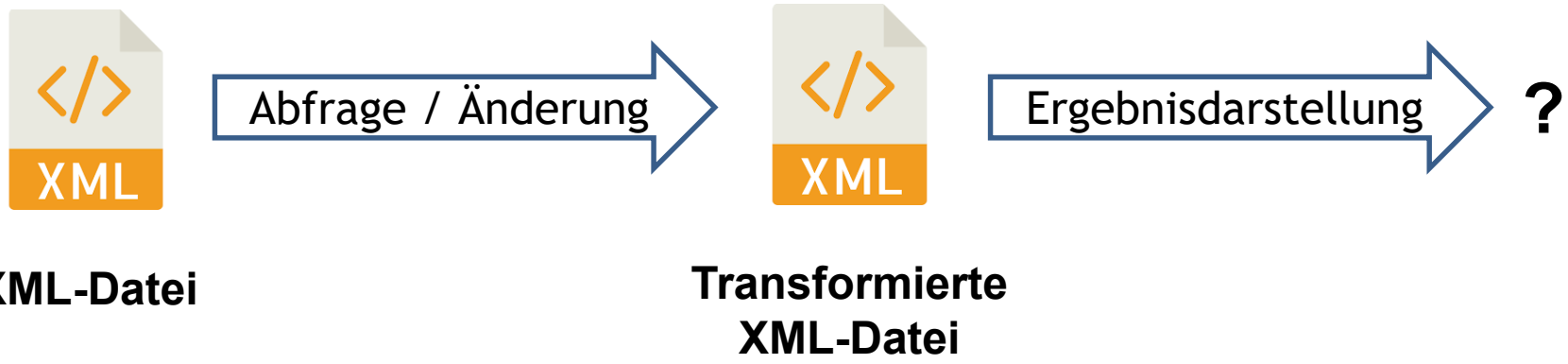
```
SELECT Name
FROM Adressliste
WHERE PLZ > 40000;
```

Name
Karl Müller

Name	Straße	Hausnr.	PLZ	Stadt
Hans Meier	Hauptstr.	21	34567	Neustadt
Ursula Meier	Hauptstr.	21	34567	Neustadt
Karl Müller	Nebenstr.	13	76543	Altstadt
Otto Schulze	Hauptstr.	22	34567	Neustadt
…	…	…	…	…

- Abfragen, Änderungen etc. auf relationalen Datenbanken werden „in-place“ auf einzelnen (Verbund-)Tabellen ausgeführt.
- Ergebnis zumeist wieder eine Tabelle („Destillat“).

Motivation



- Abfragen/Änderungen auf/von XML-Dokumenten werden als **Dokumenttrans-formationen** aufgefasst.
- Ausführung zumeist „out-of-place“ durch technische Umgebung (z.B. WWW-Server).

Abfragen als Dokumenttransformation

- Abfragen auf XML-Dokumenten können als (spezielle) Transformationen des Eingabe-Baumes in einen Ausgabebaum angesehen werden.
- Der Ausgabebaum kann dabei völlig anders strukturiert sein als der Eingabebaum und sogar neue Knotentypen enthalten (z.B. zur Darstellung von Aggregationsergebnissen).

Anforderungen an XML-Abfragesprachen

Erstes Fazit: Funktionen zur Beschreibung des Aufbaus des Ausgabebaums gehen weit über reine Abfrageoperationen hinaus.

Anforderungen an XML-Abfragesprachen

- Funktionen zum Erzeugen von Elementknoten:
  Von einfachen Kopien der Knoten des Eingabebaums bis zu völlig neue Knotentypen.
- Funktionen zum Berechnen des Inhalts von Elementen:
  Von der einfachen Extraktion und Ausgabe von Eingabe-Texten oder Attributwerten als Text bis zu beliebigen Textverarbeitungsproblemen.

Anforderungen an XML-Abfragesprachen

- Funktionen zum Erzeugen von Attributen.
- Funktionen zum Berechnen des Inhalts von Attributen (andere Restriktionen als bei Elementinhalten).

Anforderungen an XML-Abfragesprachen

- Funktionen für “low level”-Probleme (z.B. Umgang mit verschiedenen Zeichensätzen usw.).
- Funktionen zum „gleichzeitigen“ Erzeugen verschiedener Knotentypen, um den kompletten Ausgabebaum in einem koordinierten Verarbeitungsschritt aufzubauen.

UNIVERSITÄT SIEGEN

Grundlagen von XSLT

Was ist XSLT?

Transformationssprache XSLT (e**X**tensible **S**tylesheet **L**anguage for **T**ransformation) zur Definition von Transformationsregeln und Ausgabeanweisungen.

Woraus besteht XSLT?

- XPath: Definition von Pfadausdrücken zur Extraktion von Eingabedaten.
- XSL: Formatierungssprache (Extensible Stylesheet Language) zur Definition von Ausgabeanweisungen.
  http://www.w3.org/TR/xsl11/

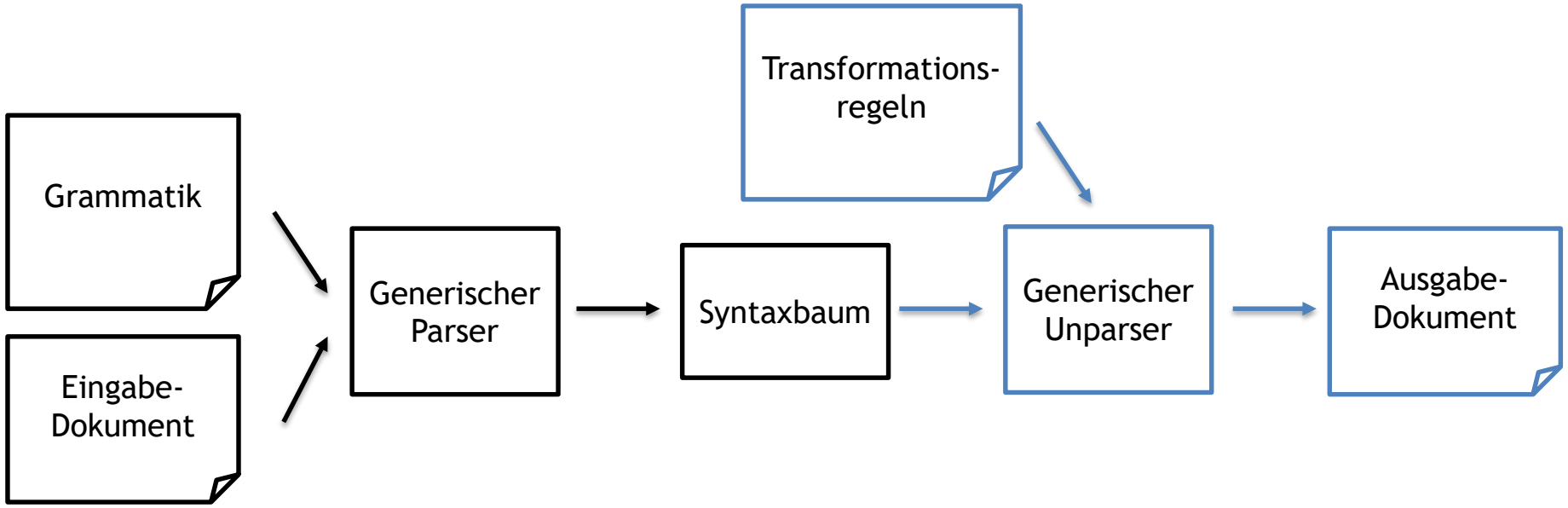
Versionen von XSLT

- XSLT 1.0, von 1999, basiert auf XPath 1.0
- XSLT 2.0, von 2007, basiert auf XPath 2.0

UNIVERSITÄT SIEGEN

XSLT-Prozessoren

Grundlagen



Grundlegende Funktionsweise von XSLT-Prozessoren.

Ablauf einer Transformation

1. Eingabe-XML-Datei(en) in einen Syntaxbaum übersetzen.
2. Den Baum durchlaufen (meistens ähnlich preorder) und für jeden besuchten Knoten passende Transformationsregeln anwenden.

Besonderheit: In Transformationsregeln können beliebig Abfrageschritte und Layout-Aufbereitung gemischt werden.

Ablauf einer Transformation

Mögliche Eingabeverarbeitung:

Validierend (ggf. optional) oder nicht validierend.

Mögliche Ausgaben:

- Ausgabe-Syntaxbaum, in XML dargestellt.
- WWW-Seiteninhalt, z.B. in HTML dargestellt.
- Beliebiger Text, z.B. in LATEX-Syntax.

Technische Umgebung

- Nicht-interaktive Batch-Programme.
  Beispiele: xsltproc , Xalan , Saxon , ...
- Nicht-interaktive, dynamisch gebundene Bibliotheken.
  Beispiel: libxslt, als Teil von:
  - WWW-/Applikationsservern (ggf. inkl. Weiterverarbeitung zu HTML)
  - WWW-Browsern (inklusive sofortiger Weiterverarbeitung in HTML)

Eingabe von XSLT-Prozessoren

- Nur wohlgeformtes (und ggf. gültiges) XML / XHTML mit den zuvor besprochenen Knotenarten.
- Die zugehörige Dokumenttypdefinition ist **kein** Bestandteil der Eingabe (allenfalls indirekt wegen Vorgabewerten)

Eingabe der Dokumentwurzel

- Der Wurzelknoten entsteht implizit beim Einlesen, ist nicht explizit textuell in der Eingabedatei repräsentiert.
- Der Wurzelknoten hat keine Attribute und (bei einer wohlgeformten Datei) genau einen Element-Kindknoten des Typs, der in der DOCTYPE-Klausel als Typ des Wurzelelements angegeben ist (dazu evtl. Kommentarknoten und Leerraum).

Ausgabe von XSLT-Prozessoren

- Wohlgeformte XML-Datei, die z.B. erneut transformiert werden könnte.
- Aber: ohne DTD, da Transformationen möglicherweise zu Abweichungen von der DTD der Eingabe-Datei führen können und eine passende DTD im Allgemeinen nicht automatisch ableitbar ist.

Ausgabemethoden von XSLT-Prozessoren

Ausgabemethoden von XSLT-Prozessoren über Steueranweisung „xsl:output“ und Parameter „method“:

- <xsl:output method=’xml’ /> ist default
- <xsl:output method=’html’ /> für html
- <xsl:output method=’text’ /> für text

(Angabe direkt hinter öffnendem „xsl:transform“-tag im Transformationsdokument)

Weitere Ausgabe-Optionen

- „version“: zur Zeit nur 1.0 erlaubt.
- „omit-xml-declaration=yes“: falls „yes“ keine XML-Deklaration in der Ausgabe, „no“ sonst.
- „standalone “: „yes/no“-Wert wird in die ausgegebene XML-Deklaration übernommen.
- „encoding“: vom Prozessor unterstützte Zeichensatzcodierungen (Vorgabewert: UTF-8).
- „indent“: „yes/no“ erlaubt dem Prozessor Leerraum für bessere Lesbarkeit einzufügen.

UNIVERSITÄT SIEGEN

XSLT-Transformationsdokumente

XSLT-Transformationsdokumente

- Dokumente enthalten im Wesentlichen eine Menge von Transformationsregeln.
- Synonymer Sprachgebrauch: „XSLT-Stylesheet-Dokumente“ oder „XSLT-Stylesheet“.

Aufbau eines Transformationsdokumentes

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<xsl:transform version='1.0'
            xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
   <xsl:output .... />
   ...
   <xsl:template ...> .... </xsl:template>
   <xsl:template ...> .... </xsl:template>
   ...
</xsl:transform>
```

Transformationsdokumente sind selbst auch XML-Dokumente.

Aufbau eines Transformationsdokumentes

- Fester Namensraum: http://www.w3.org/1999/XSL/Transform
- Wurzelelementtyp: „stylesheet“ oder „transform“ (äquivalent).
- Zulässige Kindelementtypen des „transform“ - Elements:
  - „output“-Elemente: Ausgabesteuerung; müssen zuerst kommen.
  - „template“-Elemente: Transformationsregeln.

Syntax von XSLT-Kommandos

- Notation als XML-Element (Kommandoname = Elementtypname).
  Beispiel: <xsl:template ...> .... </xsl:template>
- Oft ist der Element-Inhalt leer und Parameter werden als Element-Attribute angegeben.
  Beispiel: <xsl:output method=’text’ />

UNIVERSITÄT SIEGEN

XSLT-Transformationsregeln

Vereinfachter Einstieg in Transformationsregeln

- Das genaue Verständnis der Funktionsweise von Transformationsregeln kann schnell kompliziert werden, insbesondere wenn mehrere Regeln vorhanden sind.
- Deshalb werden wir zunächst (nicht stilgerecht) nur eine einzelne Transformationsregel für die Dokumentwurzel mit vereinfachter Struktur betrachten.

Konzeptueller Inhalt einfacher Transformationsregeln

1. Knotentyp, auf den die Regel anzuwenden ist.
2. Template (Schablobe, Textmuster), das für einen Knoten dieses Typs auszugeben ist.

Die Schablone enthält u.a. Verarbeitungs-anweisungen (Teilbäume im Transformations-dokument), bestehend aus Elementknoten, Textknoten, Kommentaren usw.

Notation einer Transformationsregel

```
<xsl:template match='Knotentyp' >
.... Schablone ....
</xsl:template>
```

- Knotentyp wird im Parameter „match“ angegeben.
- Schablone wird im Inhalt des „template“-Elements angegeben.

Transformationsregel für den Wurzelknoten

```
<xsl:template match= ’ / ’ >
.... Schablone ....
</xsl:template>
```

- Knotentyp „/“.
- Die Regel für die Dokumentwurzel wird implizit zu Beginn der Verarbeitung automatisch ausgeführt.

Aufruf einer Schablone

- Aufrufe sind zumeist implizit (kein sichtbarer Quelltext).
- Bei jedem Aufruf wird ein Kontext erzeugt, in dem die Schablone ausgeführt wird.
- Auch explizite Aufrufe von Schablonen möglich (siehe später).

Kontext eines Schablonenaufrufs

- Kontextknoten: aktuell zu verarbeitender Knoten im Eingabe-Syntaxbaum vom Typ gemäß „match“-Parameter.
- Nummerierungen (siehe später).

Grundregeln für die Ausführung einer Schablone

Elementknoten, die eine Ausgabe- oder Steueranweisung darstellen (inkl. zugehörige Attributknoten):

- Diese Anweisung ausführen.

Andere Element- und Attributknoten:

- Identisch (“wörtlich”) ausgeben (identisch von Eingabebaum in den Ausgabebaum kopieren).

Grundregeln für die Ausführung einer Schablone

Kommentarknoten:

- Ignorieren.

Grundregeln für die Ausführung einer Schablone

Textknoten:

1. Textknoten enthält nicht nur Leerraum:
identisch ausgeben.

2. Textknoten enthält nur Leerraum und ist Kind eines „xsl:text“-Elements (s.u.):
identisch ausgeben.

3. Ansonsten:
ignorieren (keine Ausgabe).

Beispiel: „Hello World!“

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

   <!-- dies ist ein Test -->
   <xsl:template match=’ / ’>
      <b x=’3’>Hello World!</b>
   </xsl:template>

</xsl:transform>
```

Leerraum-Knoten

- Achtung: Leerraum-Textknoten werden in den Eingabedaten nicht automatisch entfernt, sind ganz normale Knoten im Eingabe-Syntaxbaum.
- Zählen in Pfadschritten wie „child::node()“ o.ä. mit, z.B.„position()“, „last()“, „count(...)“.
- Leerräume innerhalb von Tags sind keine Textknoten, sind nicht im Eingabe-Syntaxbaum repräsentiert.

Whitespace Stripping

Problem: Textknoten, die nur Leerraum enthalten.

- Treten in den Eingabedaten bei eingerückter Formatierung massenhaft auf.
- Würden ohne Sonderregelung für Leerraum-Textknoten in Templates massenhaft in der Ausgabe erzeugt werden.
- Stören manchmal bei der Weiterverarbeitung.

Umfangreiche Techniken in XLST zur Behandlung vorhanden (hier nicht weiter betrachtet).

Ausgabe- und Steueranweisungen

- Alle Elemente, deren Typ im reservierten Namensraum für Anweisungen („xsl:“) liegt. Beispiel: <xsl:value-of .... />
- Formaler Parameter „select“ gibt Knoten an, auf die die Regel grundsätzlich anwendbar ist. Beispiel: <xsl:value-of select=’pfad’ />
- Kontextknoten der Schablonenausführung ist weiterer (versteckter) Parameter: Ausgangs-knoten für relative Pfade in der Schablone.

Umwandlung von Eingabeknoten

Kommando „xsl:value-of“ wandelt einen beliebigen Eingabeknoten in einen ausgegebenen Textknoten um:

- Stets leerer Inhalt.
- Hat formalen Parameter „select“, der i.d.R. einen XPath-Ausdruck enthält, um Knoten auszuwählen.

Umwandlung von Eingabeknoten

Wirkung von „xsl:value-of“:

- Berechnet den String-Wert des ersten Knotens in der Knotenmenge gemäß Dokumentordnung (bzw. leeren String, wenn Knotenmenge leer).
- Erzeugt Textknoten im Ausgabebaum mit diesem String (wenn String nicht leer).

Umwandlung von Eingabeknoten

Der String-Wert eines Knotens ist:

- Bei Textknoten: der Text.
- Bei Attributen: der Attributwert.
- Bei Elementen: Konkatenation aller Texte in allen Kind- und Nachfahren-Textknoten des Teilbaums.
- Bei Kommentaren: der Text zwischen dem öffnenden <!– und dem schließenden -->.

(Weitere Fälle s. XPath-Spezifikation).

Beispiel

- Eingabedaten: eine Lehrmodulbeschreibung.
- Ausgabe: Lehrmodulname (Kürzel) - Verantwortlicher

Beispiel

Lösung:

```
<xsl:template match=' / '>
    <xsl:value-of select='//LEHRVERANSTALTUNGSNAME' />
    (<xsl:value-of select='//LEHRVERANSTALTUNGSKUERZEL'/>)
    –
    <xsl:value-of select='//VERANTWORTLICHER/@dozentId' />
</xsl:template> .....
```

(aber: wieder viel Leerraum in der Ausgabe)

Text ohne Leerraum ausgeben

- Kommando „xsl:text“ gibt den in Schablonen enthaltenen Text ohne Leerraum aus.
- Funktion „normalize-space(..)“ entfernt Leerraum vorne und hinten und reduziert in der Mitte auf ein Leerzeichen.

Beispiel

Lösung ohne Leerraum:

```
....
<xsl:template match=' / '>
    <xsl:value-of select=
        'normalize-space(//LEHRVERANSTALTUNGSNAME)' />
    <xsl:text> (</xsl:text>
    <xsl:value-of select='//LEHRVERANSTALTUNGSKUERZEL' />
    <xsl:text>) – </xsl:text>
    <xsl:value-of select='//VERANTWORTLICHER/@dozentId' />
    <xsl:text> </xsl:text>
</xsl:template> ....
```

Iteration über Knoten

Ausgabekommando „xsl:for-each“

- Formaler Parameter „select“, der einen XPath-Ausdruck zur Beschreibung der Menge der Knoten des Eingabebaums enthält, über die iteriert werden soll (meistens direkte Kindknoten).
- Inhalt: eine “innere” Schablone, aufgebaut analog zum Inhalt einer Transformationsregel.
- Zusätzlich Sortierung der Knoten möglich.

Iteration über Knoten

Wirkung von „xsl:for-each“:

- Für **jeden** durch das „select“ gewählten Knoten wird die innere Schablone ausgeführt (d.h., dieser Knoten ist der Kontextknoten für diese Ausführung der inneren Schablone).
- Somit Wechsel des Kontextknotens beim jedem Betreten der inneren Schablone.

Beispiel

Nun zusätzlich alle Semester (mit Komma getrennt) angeben, in denen das Lehrmodul angeboten wurde:

```
<xsl:template match=' / '>
   <xsl:value-of select='//LEHRVERANSTALTUNGSNAME' />
   (<xsl:value-of select='//LEHRVERANSTALTUNGSKUERZEL' />) --
   <xsl:value-of select='//VERANTWORTLICHER/@dozentId' />
   <xsl:text> </xsl:text>
   <xsl:for-each select="//DURCHFUEHRUNG" >
      <xsl:value-of select='./@semester' />,
   </xsl:for-each>
</xsl:template>
```

Ein Komma am Ende zu viel ☹

Beispiel

Bessere Lösung:

```
<xsl:template match=' / '>
   <xsl:value-of select='//LEHRVERANSTALTUNGSNAME' />
   (<xsl:value-of select='//LEHRVERANSTALTUNGSKUERZEL' />) --
   <xsl:value-of select='//VERANTWORTLICHER/@dozentId' />
   <xsl:text> </xsl:text>

   <xsl:for-each select="//DURCHFUEHRUNG[ last()>position() ]">
      <xsl:value-of select='./@semester' />,
   </xsl:for-each>

   <xsl:value-of select='//DURCHFUEHRUNG[ last() = position() ]
      / @semester' />
</xsl:template>
```

UNIVERSITÄT SIEGEN

Verwaltung von Transformationsregeln

Motivierendes Beispiel

Gegeben: Unsere Adressliste als XML-Datei.

Gesuchte Abfrage: “Projektion” auf „Nachname“ und „Telefonnummer“:

- Gleicher Syntaxbaum wie Eingabebaum, durch Kopie vieler Knoten in den Ausgabebaum.
- Dabei werden manche Knoten / Teilbäume weggelassen und manche Knoten werden umbenannt.

Anforderungen an XML-Abfragesprache

- Der komplette Ausgabebaum soll in **einem Verarbeitungsschritt** aufgebaut werden.
- Vorgefertigte Routinen zum **automatischen Durchlauf** durch den Eingabebaum, dabei möglichst nur die **Abweichungen** vom Standarddurchlauf explizit angeben.
- **Kopieren** von Teilbäumen / Baumfragmenten sollte einfach sein (wenig Schreibaufwand verursachen).

Definition und Verwaltung von Transformationsregeln

```
<xsl:template match=' Knotentyp ' >
        .... Schablone ....
</xsl:template>
```

Wiederholung: Syntax einer Transformationsregel.

Inhalt einer Transformationsregel

Parameter „match“ zur Spezifikation der Knotentypen, auf die die Regel anwendbar ist:

- Einfachster und häufigster Fall: **Genau ein** Knotentyp.
- Allgemeiner Fall: **Mehrere** Knotentypen, auf die eine Regel anwendbar sein kann.
- Dazu: **Einschränkung** auf Knoten, die in einem bestimmten **Kontext** stehen (ähnlich zu XPath-Ausdrücken).

Inhalt einer Transformationsregel

Element „template“:

- **Schablone** / “Textmuster”, das für einen Knoten dieses Typs auszugeben ist (enthält Ausgabe- und Steuerkommandos).
- Darf **keine inneren Transformationsregeln** enthalten (es gibt keine “lokalen” Transformationsregeln).

Parameter „match“

Spezifikation einer Teilmenge der Knoten des Eingabebaums anhand von Selektionskriterien:

- Knotentyp.
- “Einbettung” des Knotens in Elternknoten und Vorfahren.
- Weitere inhaltliche Kriterien (z.B. Boole‘scher Ausdruck über Attributwerte).

Parameter „match“

Lösungskonzept: **LocationPathPatterns.**

- Eingeschränkte Form von Pfadausdrücken, die nur in Richtung der Blätter laufen.
- Idee: Template matcht auf einen Knoten „X“ im Eingabebaum, wenn es einen Startknoten „Y“ im Eingabebaum gibt, sodass das zugehörige „LocationPathPattern“ ausgehend von „Y“ den Knoten „X“ erreicht/findet.

Beispiele für Parameter „match“

- Dokumentwurzel: match=’ / ’
- Elemente: match=’ *Elementtypname* ’
- Attribute: match=’ @*Attributname* ’
- Textabschnitte: match=’ text() ’
- Kommentare: match=’ comment() ’
- Processing Instructions: match=’ processing-instruction() ’

Spezifikation mehrerer Knotentypen

NameTest ::= ’*’ | NCName ’:’ ’*’ | QName

- Angabe von **Namensmustern** für potentiell mehrere Elementtypen / Attribute.
- Identisch mit Namensmustern in Knotentests in Navigationsschritten eines XPath-Pfads.

Beispiele für Namensmuster

- match=’ person ’
- match=’ * ’
- match=’ einNamensraum:lokalerName ’
- match=’ einNamensraum:* ’

Einschränkungen für Namensmuster

Keine beliebigen regulären Ausdrücke über Typnamen (z.B. „abc*xyz“ ist falsch). Erlaubt:

- „*“: Selektion aller Elemente.
- „NR:*“ Selektion aller Elemente, deren Typname im Namensraum mit dem Namensraum-bezeichner NR liegt.
- „@*“: Selektion aller Attribute.
- „@NR:*“ selektiert alle Attribute (siehe „NR:“).

UNIVERSITÄT SIEGEN

Vollständige Syntax für Namensmuster

```
NCName      ::= NCNameStartChar NCNameChar*
                /* An XML Name, minus the ":“ (Non-Colon-Name) */
QName       ::= PrefixedName | UnprefixedName
NCNameChar              ::= NameChar - ’:’
NCNameStartChar         ::= Letter | ’_’
PrefixedName            ::= Prefix ’:’ LocalPart
UnprefixedName          ::= LocalPart
Prefix                  ::= NCName
LocalPart               ::= NCName
```

Aufzählung mehrerer Knotentypen

Pattern ::= LocationPathPattern | Pattern '|' LocationPathPattern

- Aufzählung mit Trennzeichen „|“.
- Reihenfolge der „LocationPathPattern“ in einem Pattern ist unerheblich.

UNIVERSITÄT SIEGEN

Beispiel für Aufzählung mehrerer Knotentypen

match=’ Name | Vorname | @* | meinNR:* ’

Pfade in LocationPathPatterns

Zusätzliche Bedingung: Knoten müssen Kind / Nachfahre anderer Knoten sein.

Beispiele für Pfade in LocationPathPatterns

- Regel für „ol/li“-Kombination (Element einer geordneten Liste) ist nur anwendbar auf „li“-Elemente, deren Elternknoten ein „ol“-Element ist.
- Regel „DURCHFUEHRUNG/@dozentId“ ist nur anwendbar auf „dozentId“-Attribute, deren Elternknoten ein „DURCHFUEHRUNG“-Element ist.

Vollständige Syntax für LocationPathPatterns

```
LocationPathPattern  ::= '/' RelativePathPattern? | '//'?
                         RelativePathPattern | .....
RelativePathPattern  ::= StepPattern | RelativePathPattern '/'
                         StepPattern | RelativePathPattern '//'
                         StepPattern
StepPattern          ::= ChildOrAttributeAxisSpecifier
                         NodeTest Predicate*
ChildOrAttributeAxisSpecifier ::= AbbreviatedAxisSpecifier |
                         ('child' | 'attribute') '::'
AbbreviatedAxisSpecifier      ::= '@'?
```

Allgemeine Form von LocationPathPatterns

- Sehen aus wie XPath-Ausdrücke und sind auch als syntaktisch korrekte XPath-Ausdrücke interpretierbar.
- Sind im Allgemeinen aber nicht als solche zu interpretieren, da es stets “im Syntaxbaum nach unten gehende” Navigationen sind.

Bedeutung eines LocationPathPatterns (LPP)

Eine Transformationsregel für ein gegebenes *LPP* ist **anwendbar** auf einen Knoten *N*, wenn im Eingabebaum ein Startknoten *S* existiert, sodass *N* in der Treffermenge von *LPP* ausgehend von *S* liegt.

Unterschiede zu Navigationsrichtungen von Xpath

- Explizit nur Navigationsrichtungen „child“ und „attribute“ (Kurzform wie üblich).
- Implizit (in abgekürzter Notation) ist „descendant“ erlaubt.

Unterschiede zu Navigationsrichtungen von XPath

- „//“ entspricht effektiv dem XPath-Navigationsschritt „/ descendant::node() /“ (inkl. Begrenzer-Schrägstriche; also NICHT wie bei XPath „/ descendant-or-self::node() /“)
- „//“ ist hier keine Abkürzung, sondern einzige erlaubte Notation.
  (die expandierte Notation „descendant::XYZ“ ist nicht erlaubt)

Unterschiede zu Knotentests von XPath

- XPath, 2.3 Node Tests: “*A node test node() is true for any node of any type whatsoever.*“
- XSLT-Spezifikation: “*node() matches any node other than an attribute node and the root node.*”

Aber: Knotentyptest „node()“ ist nur in den Navigationsrichtungen „child“ und „descendant“ erlaubt und dort können Attributknoten und die Dokumentwurzel nicht vorkommen.

Weitere Beispiele für match-Angaben

- „li [1]“ oder „li [position()=1]“: Selektiert alle „li“-Elemente, die das erste Kindelement vom Typ „li“ ihres Elternelements sind.
- “[position()=1 and self::li]”: Selektiert alle „li“-Elemente, die das erste Kind ihres Elternelements sind.
- „li [last()=2]“: Selektiert alle „li“-Elemente, deren Elternelement zwei Kindelemente vom Typ „li“ hat.

Anwendbarkeit mehrerer Transformationsregeln

```
<xsl:template match=" * " >
    <!-- allgemeine Regel fuer alle Elementtypen -->
    ....
</xsl:template>

<xsl:template match=" TelNr " >
    <!-- spezielle Regel fuer einen Elementtyp -->
    ....
</xsl:template>
```

Auf einen bestimmten Knoten können **mehrere Transformationsregeln** anwendbar sein.

Prioritäten von Transformationsregeln

- Auswahl der anzuwendenden Regel anhand von **Prioritäten.**
- Komplizierter Mechanismus zur expliziten Steuerung von Prioritäten, hier nicht vollständig dargestellt.

Prioritäten von Transformationsregeln

- Grundregel: **speziellere Regeln haben höhere Priorität.**
- Regel R1 ist **spezieller als** Regel R2, wenn R1 für **kleinere Menge von Knoten** anwendbar als R2.

Beispiel für Prioritäten

Regelmenge mit aufsteigender Priorität:

- match=" node() "
- match=" * "
- match=" li "
- match=" ol/li "
- match=" ol/li [position()=2]"

Verwaltung von Transformationsregeln

- Ein **Transformationsdokument** enthält eine („umgeordnete“) *Menge* von Transformations-regeln.
- Große Regelmengen können auf **mehrere Dateien aufgeteilt** und wieder in eine Hauptdatei zusammengeführt werden.

Verwaltung von Transformationsregeln

<xsl:include href=’uri-reference’ />

- Darf nur als Top-Level-Element (direktes Kind des „transform“-Elements) verwendet werden.
- „uri-reference“ muss zu einer Datei führen.
- Kein Unterschied zwischen inkludierten und lokal definierten Regeln.

Verwaltung von Transformationsregeln

<xsl:import href=’uri-reference’ />

- Identisch zu „xsl:include“, aber die importierten Regeln haben **geringere Priorität** als die lokal vorhandenen (im Sinne von Voreinstellungen).

Expliziter Aufruf von Transformationsregeln

```
<xsl:apply-templates select='...' />
```

- Kommando tritt innerhalb von Schablonen (z.B. von Transformationsregeln oder „for-each“-Kommandos) auf.
- Ähnlich wie „for-each“ ein Ablaufsteuerungs-kommando.
- Parameter „select“: enthält Pfad, der die zu bearbeitende Knotenmenge angibt.

Expliziter Aufruf von Transformationsregeln

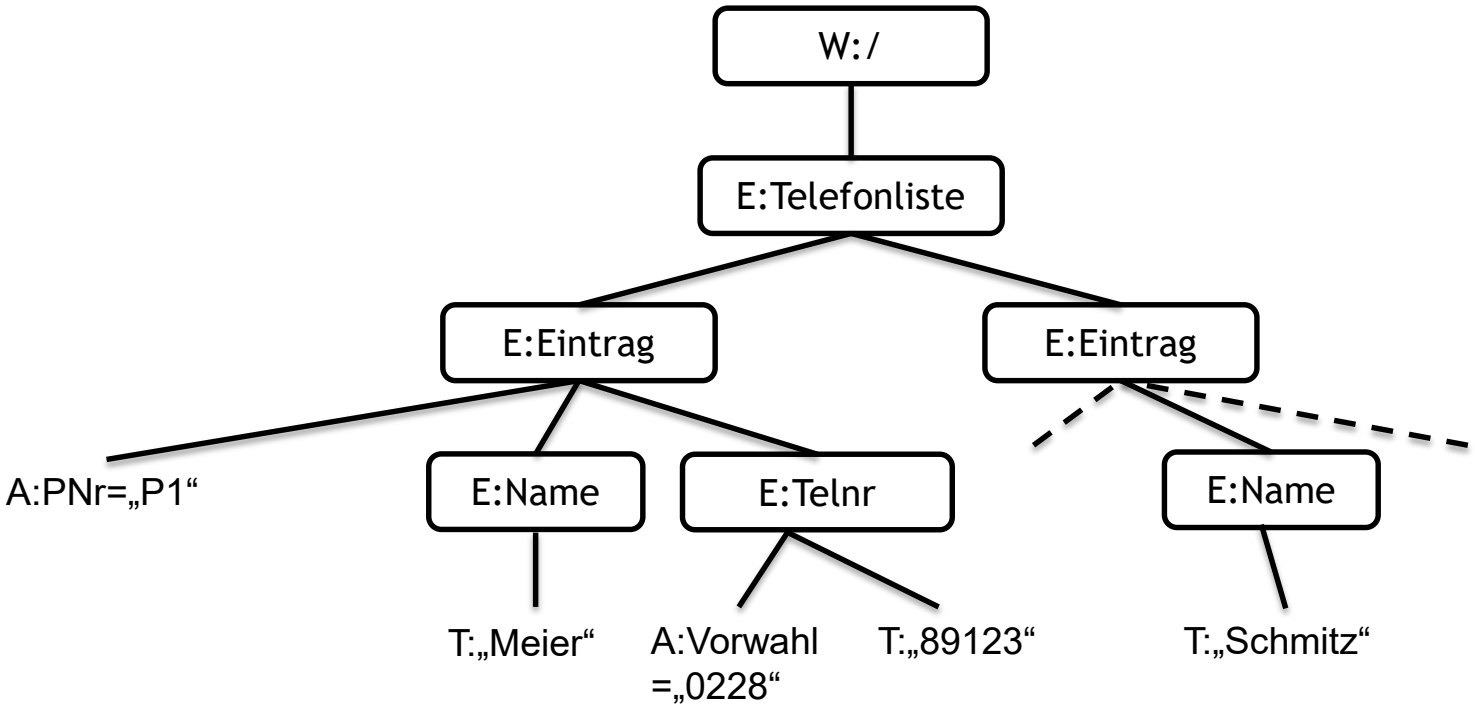
- Bestimme die Menge der zu bearbeitenden Knoten gemäß „select“-Ausdruck und bearbeite die Knoten in der Reihenfolge gemäß Eingabe.
- Für jeden zu bearbeitenden Knoten *N*:
  - bestimme den Typ von *N*.
  - bestimme die Transformationsregeln, die für Knoten dieses Typs anwendbar sind; wähle darunter *die mit der höchsten Priorität für* N.
- Rufe Schablone aus Transformationsregel mit passendem Kontext *N* auf.

Beispiel: XML-Datei

```
<?xml version=’1.0’ encoding=’ISO-8859-1’ ?>

<Telefonliste>
   <Eintrag PNr="p1" >
      <Name>Meier</Name>
      <TelNr Vorwahl="0271" >891234</TelNr>
   </Eintrag>
   <Eintrag PNr="p2" >
      <Name>Schmitz</Name>
      <TelNr Vorwahl="0228" >870887</TelNr>
   </Eintrag>
</Telefonliste>
```

Beispiel: Syntaxbaum (Ausschnitt)



Beispiel: Transformation

Aufgabe: Auch Attribut „Vorwahl“ soll separates Unterelement werden, alles andere kopieren.

Beispiel: Lösung (1)

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

<xsl:template match=’ / ’ >
    <xsl:apply-templates select=’Telefonliste’ />
</xsl:template>
…
```

Beispiel: Lösung (2)

```
…
<xsl:template match=’ Telefonliste ’ >
    <Telefonliste>
        <xsl:apply-templates select=’Eintrag’ />
    </Telefonliste>
</xsl:template>
…
```

Beispiel: Lösung (3)

```
…
<xsl:template match=’ Eintrag ’ >
   <Eintrag>
      <xsl:apply-templates select=’Name’ />
      <xsl:apply-templates select=’TelNr/@Vorwahl’ />
      <xsl:apply-templates select=’TelNr’ />
   </Eintrag>
</xsl:template>
…
```

Beispiel: Lösung (3)

```
…
<xsl:template match=’ Name ’ >
    <Name>
        <xsl:value-of select=’.’ />
    </Name>
</xsl:template>
…
```

Beispiel: Lösung (4)

```
…
<xsl:template match=’ TelNr ’ >
    <TelNr>
        <xsl:apply-templates select=’text()’ />
    </TelNr>
</xsl:template>
…
```

Beispiel: Lösung (5)

```
…
<xsl:template match=’ @Vorwahl ’ >
    <Vorwahl>
        <xsl:value-of select=’.’ />
    </Vorwahl>
</xsl:template>
…
```

Beispiel: Lösung (6)

```
…
    <xsl:template match=’ text() ’ >
        <xsl:value-of select=’.’ />
    </xsl:template>

</xsl:transform>
```

Beispiel: Diskussion

- Die Dokumentwurzel des Ausgabebaums wird implizit erzeugt, sie kann nicht explizit erzeugt werden.
- Die Transformationsregel mit match=’ / ’ enthält daher keine Anweisung, die den aktuell bearbeiten Knoten in der Ausgabe reproduziert (im Gegensatz zu den Transformationsregeln für die Elemente).

Beispiel: Diskussion

- Die inneren Knoten des Eingabebaums müssen zumeist nur **reproduziert** (kopiert) werden.
- Hierfür für **jeden Elementtyp** eine eigene Transformationsregel der Form:

```
<xsl:template match=' elementtyp ' >
   <elementtyp>
      <xsl:apply-templates select='Kinder' />
   </elementtyp>
</xsl:template>
```

Beispiel: Diskussion

- Der Kontextknoten wird jeweils “manuell” in der Ausgabe rekonstruiert (durch öffnenden und schließenden Tag).
- Die in diesen Tags liegenden Ausgabe-anweisungen erzeugen Kinder dieses Knotens.
- Das Kommando:
  „<xsl:apply-templates select=’*Kinder*’ />“
  bearbeitet alle Kinder und ruft implizit die jeweils passende Transformationsregel auf.

Beispiel: Diskussion

- Mit „<xsl:apply-templates ... />“ bis zu den Blättern des Eingabebaums laufen (siehe Regeln für „TelNr“ und „text()“).
- Alternativ: bei Elementen, die nur noch Kindknoten vom Typ „Text“ haben, schon für diese Elemente eine Transformationsregel definieren die “<xsl:value-of select=’.’ />” aufruft (siehe Regel für „Name“).

Beispiel: Diskussion

- Reihenfolge der Transformationsregeln irrelevant.
- Attribut „PNr“ ist verlorengegangen (Attribute mit Inhalt, der von der Eingabe abhängt, können wir mit dem bisher Erlernten nicht erzeugen).
- Ebenso Kommentare.

Vorgabewerte für „select“

- Der Parameter „select“ im Kommando „xsl:apply-templates“ kann weggelassen werden.
- Dann ist „child::node()“ gültiger Vorgabewert.

Testfrage

„<xsl:template>“

„<xsl:apply-templates>“

- Was ist der Unterschied zwischen diesen Elementtypen?
- Wo dürfen diese Elementtypen auftreten?

Antwort

„<xsl:template>“

- Definiert eine **Transformationsregel** für einen oder mehrere Knotentypen.
- Darf nur als Kind eines „<xsl:transform>“-Knotens (also des Wurzelelements eines Transformationsdokuments) auftreten.

Antwort

„<xsl:apply-templates>“

- Ist ein **Ablauifsteuerungskommando.**
- Gibt an, dass auf die im Parameter „select“ spezifizierten Knoten die Transformationsregel mit der jeweils höchsten Priorität angewandt werden soll.
- Darf nur in einer Schablone, also z.B. im “Rumpf” einer Transformationsregel, auftreten.

Vordefinierte Transformationsregeln

Bei der Verarbeitung der Eingabe muss **für jeden Knotentyp wenigstens eine Transformations-regel definiert sein**, auch wenn das Transformationsdokument keine passende Regel definiert.

Beispiel: Text-Knoten

Vordefinierte Transformationsregeln

Lösung: Für alle Knotentypen (außer Namensraumknoten) sind Transformationsregeln vordefiniert.

- Diese Regeln sind **sehr allgemein** und haben daher **geringste** Priorität.
- Applikationsspezifische Regeln sind **spezieller** und haben daher **höhere** Priorität.
- Vordefinierte Transformationsregeln hängen vom **Knotentyp** ab.

Vordefinierte Transformationsregeln

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

Für Elemente und die Dokumentwurzel:

- Bewirkt **rekursiven Abstieg durch den Baumstruktur** der Elemente (genauer: der Kindknoten ohne Attribute).
- Verarbeitung in der Eingabereihenfolge.

Vordefinierte Transformationsregeln

```
<xsl:template match=" text() | @* " >
    <xsl:value-of select=’ . ’ />
</xsl:template>
```

Für Textknoten und Attribute:

- Bewirkt Ausgabe des Inhalts als Text, **sofern** der Knoten überhaupt verarbeitet wird.
- Bei Attributen: nicht automatisch.

Vordefinierte Transformationsregeln

```
<xsl:template match=" processing-instruction() |
                      comment() " />
```

Für Verarbeitungsanweisungen und Kommentare:

- Werden **nicht** ausgegeben.

Vordefinierte Transformationsregeln

```
<xsl:apply-templates select=’ / ’ />
```

Implizit vorhandenes, voreingestelltes Hauptprogramm:

- Verarbeitet Dokumentwurzel des Syntaxbaums gemäß zugehöriger Transformationsregel.

Die identische Transformation

Ein Transformationsdokument, das nur die folgende Transformationsregel enthält, reproduziert den ganzen Eingabesyntaxbaum unverändert.

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

Die identische Transformation

- „node() | @*“: Kurzschreibweise für „attribute::* | child::node()“.
- „node()“ selektiert bzw. ist anwendbar auf alle Knotentypen außer Attribute und der Dokumentwurzel.
- „@*“ selektiert bzw. ist anwendbar auf alle Attribute.
- Dokumentwurzel wird speziell behandelt (hier kein Thema).

Kopier-Kommando

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

- Kopiert aktuellen Kontextknoten in den Ausgabesyntaxbaum.
- Kein „select“-Parameter.
- Schablone wird ausgeführt und alle erzeugten Knoten werden an den kopierten Knoten als Kind angehängt.

Projektionen

Entwurfsmuster für Projektionen:

- Identische Transformation als allgemeinste Regel definieren.
- Für auszublendende Attribute oder Elemente (und darunterliegende Teilbäume) **Transformationsregeln mit leerer Schablone** definieren.

Beispiel: Projektion

Aufgabe: Lösche die „TelNr“-Elemente in der Telefonliste.

Lösung

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

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

   <xsl:template match=" TelNr " />

</xsl:transform>
```

UNIVERSITÄT SIEGEN

Ausgabe- und Steuerkommandos

(Auswahl)

Ausgabe und Steuerkommandos

- Werden als Element mit Typ „xsl:...“ notiert.
- Sind entweder Steuerkommandos oder erzeugen direkt Teile des Ausgabebaums.
- Treten in Schablonen auf und haben oft einen Inhalt, der wiederum eine Schablone darstellt.

Steuerkommandos

- „xsl:for-each“: Iteriert über Knotenmenge und ruft für jeden Knoten die innere Schablone auf.
- „xsl:apply-templates“: Iteriert über Knotenmenge und wendet jeweils zutreffende Transformationsregel an.
- „xsl:if“: Bedingter Aufruf innerer Schablone.
- „xsl:choose“: Mehrere nacheinander zu testende Bedingungen (evtl. mit „otherwise“).

Ausgabekommandos

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

- „xsl:copy“: Kopiert einen Knoten beliebigen Typs des Eingabebaums (den Kontextknoten, kein „select“-Parameter).
- „xsl:copy-of select=’ XPath-Ausdruck ’“: Kopiert beliebig viele Teilbäume des Eingabebaums. (Wurzeln der Teilbäume gemäß XPath-Ausdruck)

Ausgabekommandos

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“)

Ausgabekommandos

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“)

Ausgabekommandos

Ausgabekommandos, die **Attributknoten** erzeugen:

- „xxx='...'“ Direkte Angabe von Attributname und Wert im öffnenden tag; nur möglich, wenn auch das Element direkt angegeben wird (ggf. variable Inhalte mit Attributwertschablonen).
- „xsl:attribute“: Erzeugt einen Attributknoten; Attributname und Wert in Parametern angegeben, evtl. dynamisch berechnet.

(ggf. auch xsl:copy und xsl:copy-of)

Kommando „xsl:copy-of“

<xsl:copy-of select=’ ... ’ />

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

Kommando „xsl:if“

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

- Hat einen Parameter „test“, der einen Boole‘schen Ausdruck enthält.
- Inhalt: Innere Schablone.
- Kein “else”-Zweig möglich.

Kommando „xsl:if“

Wirkung:

- Der im Parameter „test“ enthaltene Boole‘sche Ausdruck wird ausgewertet.
- Bei positivem Testergebnis wird die innere Schablone ausgeführt.
- Kontextknoten für die innere Schablone ist der gleiche wie derjenige der aufrufenden Schablone.

UNIVERSITÄT SIEGEN

Kommando „xsl:choose“

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

Attributwerte

Problem: Attribut soll keinen festen Wert erhalten, sondern berechnet werden.

- 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.

Attributwerte

Beispiel: unsere Telefonliste 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>
```

Attributwerte

Attribut „land“ kann direkt mit festem Wert angegeben werden:

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

Attributwerte

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

Beispiel: Wert von „nummer“ ist Kopie des Inhalts des Elements „Telnr“.

Attributwerte

„xsl:value-of“ kann man aber nicht benutzen:

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

Attributwerte

Begründung:

- Syntaktisch: „<“: ist in Attributwerten nicht erlaubt; Umcodieren als „&lt“ nützt nichts: dann wird das < “wörtlich” genommen und kein Kommando interpretiert.
- Strukturell: das Kommando müsste in der Schablone ein Kindelement des Attributs sein – das ist generell nicht erlaubt.

Attributwertschablonen

Ausdruck, der Knotenliste liefert.

- Angabe in geschweiften Klammern:
  { Ausdruck }
- Ausgabe: textuelle Darstellung des ersten (!) Knotens der Liste.

Attributwertschablonen

Beispiel:

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

Attributwertschablonen

Weiteres Beispiel: Jetzt nur ein Attribut mit kompletter Tel.Nr. gemäß Muster “[0049] 0271-7402611”

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

Attributwert wird durch mehrere Attributwertschablonen und feste Texte erzeugt.

Kommando „xsl:attribute“

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

- „xsl:attribute“-Anweisung muss vor Anweisungen ausgeführt werden, die den Inhalt des Elements erzeugen.
- Hat Parameter „name“, der den Namen des zu erzeugenden Attributs angibt.

Kommando „xsl:attribute“

- Innere Schablone zur Berechnung des Wertes des Attributs.
- Wert ergibt sich durch Konkatenation aller erzeugten Text-Knoten.

Kommando „xsl:attribute“

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

<DURCHFUEHRUNG semester=’Meier:2007s; Koch:2008w;’ />

Kommando „xsl:attribute“

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>
```

Kommando „xsl:attribute“

Beispiel: „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>
```

Kommando „xsl:attribute“

Ergebnis:

<alles x="
  123abc456
  "/>

Kommando „xsl:element“

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

- „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.

Kommando „xsl:element“

Beispiel: DURCHFUEHRUNG-Elemente bilden, die die „dozentId“ im Elementnamen enthalten.

Kommando „xsl:element“

Lösung:

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

Variablen

Variable = Paar (Name, Wert)

- Aber: Finale Wertzuweisung bei der Deklaration, danach keine erneute Wertzuweisung erlaubt.
- Können in verschiedenen Kontexten benutzt werden (Suchbedingungen in Pfaden, Ausgabeanweisungen usw.).
- Sind als top-level-Element zulässig, aber auch als Anweisung innerhalb von Schablonen.

Variablen

- Variablen werden u.a. benötigt, um Gruppierungen / Aggregationen und Verbunde zu berechnen.
- In XSLT Version 2 deutliche Verbesserungen im Vergleich zu XSLT Version 1
- Aber: Es gibt noch immer sehr viele Sonderfälle und Verhaltensvarianten (Verständlichkeit?)

Deklaration von Variablen

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

- Attribut „name“ muss syntaktisch korrekten Namen enthalten.
- Angabe des Werts entweder über Attribut „select“ oder durch eine innere Schablone.

Benutzung von Variablen

Benutzung von Variablen in der Form:

„$variablenname“

Wertangaben für Variablen

Über „select“: muss Ausdruck enthalten, der

- eine Zeichenkette,
- eine Zahl,
- einen Boole‘schen Wert oder
- eine Knotenmenge

liefert.

Falls Knotenmenge: Knoten können als Ausgangspunkt von Navigationen dienen.

Wertzuweisungen für Variablen

Beispiele:

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

(Zu beachten: automatische Konversionen)

Wertzuweisungen für Variablen

Weitere 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>
```

Ergebnis?

Wertzuweisungen für Variablen

Ergebnis:

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

Wertzuweisungen für Variablen

Über innere 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>
```

Wertzuweisungen für Variablen

Ergebnistyp: „result tree fragment“.

- Ausgabe mit „xsl:value-of:“ konvertiert zu einem Textknoten.
- Ausgabe mit „<xsl:copy-of select=’$xxx’ />:“ als komplette Kopie mit allen inneren Knoten.
- „Result tree fragments“ können nicht als weitere Eingabe, Startpunkt von Navigationen etc. benutzt werden (in XSLT 1.0).

Verbundbildung mit Variablen

- Ziel: Nachimplementierung eines „relationalen Verbundes“ von Hand unter Benutzung von Variablen.
- Idee: Bestimmung der Verbundpartner mit einem XPath-Ausdruck.

Verbundbildung mit Variablen

Beispiel: Lehrveranstaltungsdaten mit „dozentId“ als “Fremdschlüssel” auf Personendaten.

```
<FBINFO>
   <PERSONEN>
      <PERSON persId = ’Schmidt’ nachname= ’Schmidt’
                     vornameInit=’U.’ fachgr=’MBE’ />
       ....
   </PERSONEN>
   <LEHRVERANSTALTUNG>
      ....
      <VERANTWORTLICHER dozentId=’Schmidt’ />
      ....
   </LEHRVERANSTALTUNG>
</FBINFO>
```

Verbundbildung mit Variablen

Aufgabe: Im Element „VERANTWORTLICHER“ sollen innen „Name“, „Initialen“ und Fachgruppenzugehörigkeit eingetragen werden.

```
<FBINFO>
   ....
   <LEHRVERANSTALTUNG>
   ....
      <VERANTWORTLICHER dozentId=Schmidt' >
         Schmidt, Ulrich (MBE)
      </VERANTWORTLICHER>
   ....
   </LEHRVERANSTALTUNG>
</FBINFO>
```

Verbundbildung mit Variablen

Lösungsstrategie: Verbund manuell in drei Schritten implementieren.

1. Variable „dzId“ mit dem Fremdschlüssel-wert anlegen.
2. Variable nutzen, um in der “Zielrelation des Fremdschlüssels” den zugehörigen Eintrag zu lokalisieren und in zweiter Variable „dzElem“ Referenz auf diesen Eintrag speichern.

Verbundbildung mit Variablen

3. Von der Referenz in der zweiten Variablen aus zu den auszugebenden Daten navigieren.

Verbundbildung mit Variablen

Lösung (Ausschnitt):

```
<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>
```

Diskussion: Verbundbildung mit Variablen

In Schritt 2 wäre intuitiv, 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 Variable „@dzId“ mit den bisherigen Konzepten nicht rekonstruierbar.

Diskussion: Verbundbildung mit Variablen

- Variable „dzElem“ enthält i.A. eine Menge von Referenzen auf Knoten im Eingabebaum (weil mit Ausdruck „select=’ pfad ’“ gesetzt).
- Wenn die Daten korrekt sind, ist aber maximal ein Element möglich: von dort aus weiternavigieren.
  Beispiel: „$dzElem / @nachname“

Diskussion: Verbundbildung mit Variablen

- Variable „dzElem“ ist verzichtbar (verbessert aber Lesbarkeit):
  Jedes Auftreten von „$dzElem“ in Schritt 3 ersatzbar durch
  „select=’ // PERSON [@persId = $dzId] ’“.
- In Schritt 3 wäre „dozentId=’{ @dozentId }’“ ebenfalls richtig, aber „dozentId=’$dzId’“ wäre falsch (würde „$dzId“ wörtlich ausgeben).

Diskussion: Verbundbildung mit Variablen

Effizienzproblem:

- Implementierung des Pfadausdrucks „// PERSON [@persId = $dzId]“ kann am einfachsten durch lineare Suche implementiert werden.
- Aber: Sehr ineffizient.
- Lösung: Sekundärindex, müsste manuell angelegt werden.

Verbundergebnisse weiterverarbeiten

Weitere Verbunde bilden

- Beispiel: Über Referenz „@fachgr“ sollen weiterer Daten einer Fachgruppe ergänzt werden (z.B. Liste aller Lehrveranstaltungen einer bestimmten Fachgruppe).
- Möglich, aber sehr kompliziert und fehler-anfällig.
- Besser: temporäre XML-Datei.

Mehrere Ein- und Ausgabedateien

Bisherige Beschränkung auf genau eine 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.

Kommando „xsl:document“ (seit XSLT 1.1)

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

- Erzeugt XML-/HTML-/Text-Datei gemäß Angabe in „@method“ (siehe „xsl:output“-Elemente).
- Name der Ausgabedatei in „@href“.

Kommando „xsl:document“ (seit XSLT 1.1)

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

Kommando „xsl:document“ (seit XSLT 1.1)

Beispiel (Ausschnitt):

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

Kommando „xsl:document“ (seit XSLT 1.1)

Beispiel (zugehöriges „Hauptprogramm“):

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

Funktion „document()“ (vereinfacht)

„node-set document(uri:string)“

- Die im Argument „uri“ angegebene XML-Datei wird eingelesen und eine Referenz auf die Dokumentwurzel des Syntaxbaumes wird zurückgegeben.
- Von dort kann (wie bei einer Variablen) mit einem relativen Pfadausdruck weiternavigiert werden.

Funktion „document()“ (vereinfacht)

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>
```

Funktion „document()“ (vereinfacht)

- Diese Transformation erzeugt (bei beliebiger Eingabe) die Textausgabe “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.

Funktion „document()“ (vereinfacht)

Nutzung von „document(...)“:

- Aufsammeln von Daten aus verschiedenen Quellen.
- Vorverarbeitung der Eingabedaten mit „xsl:document“ (z.B. Verbundbildung).
- Danach Wiedereinlesen dieser Daten mit „document(...)“.

UNIVERSITÄT SIEGEN

Indizes

Sekundärindexe im XSLT-Kontext

- Wiederholung aus DBS I: Ein Sekundärindex ist ein spezieller Index, der einem Sekundärschlüsselwert eine Trefferliste zuordnet.
- Eine “Trefferliste” wäre im Kontext von XSLT eine Menge von Knoten des Eingabebaums.

Anlegen eines Sekundärindexes

```
<!-- Category: top-level-element -->
<xsl:key name ='qname'
         match='pattern'
           use ='expression' />
```

- Parameter „name“: Name des Sekundärindex (beliebig viele möglich).
- Parameter „match“: Typen der indexierten Knoten (wie „match“ in Transformationsregeln).
- Parameter „use“: Ausdruck, der zu einem Knoten den Sekundärschlüsselwert liefert.

Anlegen eines Sekundärindexes

Effekt des „xsl:key“-Kommandos:

1. Bestimme alle Knoten, die zu „match“ passen.
2. Für jeden dieser Knoten: berechne Sekundärschlüsselwert gemäß „use“ (kann aus mehreren Datenwerten mit Textfunktionen wie „concat(..)“, „substring(...)“ usw. konstruiert werden).
3. Bestimme zu jedem Sekundärschlüsselwert die Trefferknotenliste.

Verwenden von Sekundärindizes

Funktion zum Abruf der Trefferliste für einen Sekundärschlüsselwert:

key (SIName: string, SSWert: object) : node-set

- 1. Parameter: Bezeichner des Sekundärindexes.
- 2. Parameter: Sekundärschlüsselwert.
- Rückgabe: Trefferliste, also Liste von Referenzen auf Knoten des Eingabebaums.

Verwenden von Sekundärindizes

Beispiel: Zeige alle Lehrveranstaltungen in einem bestimmten Semester an.

Verwenden von Sekundärindizes

Lösungsvariante 1:

```
<xsl:key name  =’LVproSemester’
            match =’ DURCHFUEHRUNG ’
               use  =’ @semester ’ />

<xsl:template match=’ / ’>
   <xsl:for-each
               select=’ key( "LVproSemester", "2006s" ) ’ >
      <xsl:value-of select=’ @semester ’ />
      <xsl:value-of select=’ @dozentId ’ />
      <xsl:value-of select=’ .. / LEHRVERANSTALTUNGSNAME ’ />
   </xsl:for-each>
</xsl:template>
```

Verbundbildung mit Index

Lösungsvariante 2:

```
<xsl:key name  =’personendaten’
            match =’ PERSON ’
               use =’ @persId ’ />

<xsl:template match=’ VERANTWORTLICHER ’ >
   <VERANTWORTLICHER dozentId=’{ @dozentId }’ >
      <xsl:value-of select=’ key( "personendaten", @dozentId ) /
        @nachname’ />,
      <xsl:value-of select=’ key( "personendaten", @dozentId ) /
        @vornameInit’ /> ...
   </VERANTWORTLICHER>
</xsl:template>
```

Lästige Wiederholung

Verbundbildung mit Index

Lösungsvariante 3: Einsatz einer Variablen

```
<xsl:key .... s.o. .... />

<xsl:template match=' VERANTWORTLICHER ' >

   <xsl:variable name="VA"
                 select=' key( "personendaten", @dozentId )' />

   <VERANTWORTLICHER dozentId='{ @dozentId }' >
      <xsl:value-of select=' $VA / @nachname ' />,
      <xsl:value-of select=' $VA / @vornameInit ' /> ...
   </VERANTWORTLICHER>
</xsl:template>
```

Verbundbildung durch Anreichern von Elementen

Typisches Szenario:

- Vorhandenes (ggf. komplexes) Element enthält eine oder mehrere Referenzen auf andere Elemente (Verbundpartner).
- Daten von den Verbundpartnern sollen zu diesem Element “hinzukopiert” werden (z.B. zum lokalen Weiterzuverarbeiten).
- Das Element soll also komplett in die Ausgabe kopiert werden, ergänzt um zusätzliche Daten.

Verbundbildung durch Anreichern von Elementen

Lösungsschema (X = Typ des Elements):

- Für jedes Referenzattribut (bzw. einen entsprechenden Schlüsselwert) einen passenden Sekundärindex auf die Zielelemente anlegen.
- Generell identische Transformationsregel benutzen.

Verbundbildung durch Anreichern von Elementen

- Dazu: Spezielle Transformationsregel für X, die folgendes kopiert:
  - Die Attribute von X.
  - Die “hinzukopierten” Attribute von den Verbundpartnern.
  - Die Kinder von X.
  - Die Kinder der Verbundpartner.

Verbundbildung durch Anreichern von Elementen

Beispiel: Personendaten von VERANTWORTLICHER

```
<xsl:include href="identisch.xslt" />

<xsl:key name =’personendaten’ .... />

<xsl:template match=’ VERANTWORTLICHER ’ >
   <xsl:copy>
                                                    <!-- 1. lokale Attribute kopieren -->
      <xsl:apply-templates select=’ @* ’ />
                                                    <!-- 2. entfernte Attribute kopieren -->
      <xsl:apply-templates select=’
            key( "personendaten", @dozentId ) / @* ’ />
                                                    <!-- 3. lokale Kinder kopieren -->
      <xsl:apply-templates select=" node() " />
      ....
   </xsl:copy>
</xsl:template>
```

Verbundbildung durch Anreichern von Elementen

- “<xsl:apply-templates select=’ node() | @* ’ />” aus der identischen Transformation geht nicht als 1. Schritt, weil zuerst alle Attribute in der Ausgabe erzeugt werden müssen (vor den children)
- Alternativen zu „... / @*“ in Schritt 2:
  - .. / @nameEinesAttributs
  - .. / @* [name(.)=’fachgr’ or name(.)=’nachname’] („name()“ liefert Namen (Typ) eines Element oder Attributknotens)

Gruppierung und Aggregation

Beispiele mit Zählung / Summierung:

- Zahl der Module:
  <xsl:value-of select=
  ’ count(key("LVproSemester", "2006s")) ’ />
- Gesamtzahl der LP:
  <xsl:value-of select=
  ’ sum(key("LVproSemester", "2006s") / .. /
  LEISTUNGSPUNKTE / @anzahl) ’ />

Gruppierung mit Duplikateneliminierung

Beispielaufgabe: Zeige pro Semester die dort stattfindenden Lehrveranstaltungen.

Problem:

- Die Semesterkürzel treten in den Attributen „//DURCHFUEHRUNG/@semester“ mehrfach auf.
- Ausgegeben werden soll aber nur ein Eintrag pro auftretendem Datenwert (Semesterkürzel).

Gruppierung mit Duplikateneliminierung

Die wesentliche Arbeit (insbesondere die Duplikateneliminierung) wird beim Anlegen eines Sekundärindexes geleistet:

```
<xsl:key name =’semkrzl2semester’
match=’DURCHFUEHRUNG/@semester’ use =’.’ />
```

Gruppierung mit Duplikateneliminierung

- Dieser Sekundärindex enthält pro Wert, der in den Attributen „@semester“ auftritt, einen Eintrag.
- Die zugehörige Trefferliste enthält Referenzen auf alle Attribut-Knoten in der Eingabe, wo dieser Wert vorkommt.
- Problem: es gibt keinen **Iterator**, mit dem man über alle Trefferlisten iterieren könnte.

Ersatzkonstruktion für Iteratoren

Variable, die pro auftretenden Wert genau eine Referenz auf diesen Wert enthält:

```
<xsl:variable name='alleVerschiedenenSemkrzl'
    select=' // DURCHFUEHRUNG / @semester
        [ generate-id( . ) =
          generate-id( key( "semkrzl2semester", . ) [1] )
        ] ' />
```

Ersatzkonstruktion für Iteratoren

- Alle „@semester“-Knoten werden durchsucht.
- Selektiert werden die Knoten, die in der Trefferliste „(key(’semkrzl2semester’, .))“ auf Platz eins („[1]“) stehen.
- Um Referenzen auf Konten in der Eingabe vergleichen zu können, müssen die Referenzen erst in einen String umgewandelt werden („generate-id()“).

Ersatzkonstruktion für Iteratoren

Die Variable „alleVerschiedenenSemkrzl“ enthält für jeden Wert, der in der Eingabe vorkommt, genau eine Referenz auf einen „@semester-Knoten“ in der Eingabe, in dem dieser Wert steht.

Ersatzkonstruktion für Iteratoren

Testfrage: warum funktioniert die folgende Lösung ohne „generate-id()“ nicht?

```
<xsl:variable name=’alleVerschiedenenSemkrzl’
    select=’ // DURCHFUEHRUNG / @semester
        [ . =
           key( "semkrzl2semester", . ) [1]
        ] ’ />
```

Ersatzkonstruktion für Iteratoren

Antwort: Diese Lösung eliminiert Duplikate nicht.

- In dem Gleichheitsvergleich werden hier die textuellen Werte der Knoten verglichen, und nicht die Referenzen auf die Knoten im Eingabebaum.
- Die textuellen Werte der Knoten sind bei allen Einträgen in der Trefferliste definitionsgemäß gleich.

Diskussion: Indexe

- Wesentlich effizienter als viele direkte Abfragen, wenn der gleiche Datenbestand wiederholt durchsucht werden muss.
- Günstig für Verbundbildung oder Gruppierung / Aggregation.

UNIVERSITÄT SIEGEN

Diskussion

Vorteile von XSLT

- Bei sehr einfachen Web-Applikationen kommt man zumeist mit wenigen “kleinen” Transformationen (50 - 200 Zeilen) aus.
- Grundlegende Operatoren (Selektionen, Projektion, Verbund) schematisch mit Standard-Mustern sicher implementierbar.

Nachteile von XSLT

- Schwerpunkt auf Abfragen und einfache „Textverarbeitung“.
- Kaum geeignet für „vollwertige“ Algorithmen bzw. Applikationen.
- Große Mängel in der Lesbarkeit und Kompaktheit.
- Kein ausgereiftes Modulkonzept.
- Fehlende Trennung zwischen Datenextraktion und Fachlogik häufige Fehlerquelle.

Prüfungsstoff

- XSLT-Technologie einordnen und bewerten.
- Grundlegende Verarbeitungsprinzipien von XSL-Transformationen erklären.
- XSLT-Regeln definieren und anwenden.
- XSLT-Kommandos kennen und verwenden.
- Variablen-Konzept und Verbundberechnung kennen und verwenden.
- Verbundberechnung mit Indizes erklären.