LV Textkodierung und Textanalyse mit TEI, UE

Einheit 4: XPath und XSLT-Basisstylesheets

Sarah Lang

Universität Wien, Jänner 2021

Zentrum für Informationsmodellierung, Graz

Übersicht

Präliminaria

Lehr- & Lernziele

1. Kennenlernen von XPath und XSLT
2. Im Zuge von Übungen sowohl XPath als auch XSLT mindestens 1x selbst verwendet haben
3. Überblick, was XPath und XSLT an Mehrwert bieten
4. Grundsätzliches Verständnis der Funktionsweise von XSLT und XPath
5. In diesem Rahmen *nicht* möglich: Aneignung fundierter Kenntnis und Beherrschung von XSLT und XPath

Hausübung

Eigenständiges Umsetzen (bzw. Modifizieren) einer simplen Transformation in XSLT, im Zuge dessen auch XPath zur Anwendung kommt. Ein Basisstylesheet wird zur Verfügung gestellt, auf dem aufgebaut werden kann.

Vorstellungsrunde

Leitfragen

1. Name, Studienrichtung
2. Warum Interesse an Veranstaltung?
3. Erfahrungen mit Computer und/oder Digital Humanities & aktueller Kenntnisstand

Sarah Lang

- urspr. aus Aalen (DE/BW)
- Studium: Latein & Französisch & Geschichte (Lehramt), Bachelor in Archäologie, Master in Religionswissenschaft, KFU Graz
- Arbeit am Institut für Religionswissenschaft: u.a. Monster, Horrorfilm - später Alchemie
- DH-Zertifikat - Arbeit am Zentrum für Informationsmodellierung (ZIM-ACDH) Graz
- Moralische Wochenschriften → *gams.uni-graz.at/mws*
- Repositorium zu antiken Fabeln → *gams.uni-graz.at/graf*
- DH-Dissertation: Wissensrepräsentation und automatisierte Annotation alchemischer Symbolsprache (Bsp: Korpus des Chymikers Michael Maier, 1568–1622)

Abfragen mit XPath

Intro i

Mit XPath kann in XML-Dokumenten navigiert werden.
//persName = ‘Gib mir alle Personennamen / alle Personen.’
→ alle Funde werden visuell markiert. Ich habe sie damit ‘gefunden’ und ‘angesprochen’. Mit dieser Abfrage kann ich diese Elemente ‘anwählen’, z.B. zur Weiterverarbeitung in XSLT.

Intro ii

Um mit XPath etwas zu finden:

1. Auf Deutsch in einem vollständigen Satz formulieren, was man erreichen will. ‘Ich möchte alle Vorkomnisse der Person Mina.’
2. Auf technische Struktur des Dokuments umlegen: Diese Vorkomnisse sind alle als `persName ref="#Mina"` kodiert, womit ich das Mittel habe, sie per XPath zu addressieren.
3. Möchte ich Elemente ausgegeben haben, ihre Inhalte? Attribute, ihre Inhalte? Oder gar das Ergebnis einer Funktion (‘Zähle alle Vorkomnisse und gib mir die Anzahl.’)?
4. Gibt es Bedingungen, also will ich nur bestimmte Ergebnisse oder alles, was gefunden wird?

Jedes Vorkommen von Mina wäre ‘übersetzt’ also: Alle `persName`, die `@ref='Mina'` haben, *im ganzen Dokument*.

Pfade & Scope i

Scope: relative & absolute Pfade

Ich kann, je nach Abfrage, **im ganzen Dokument** oder **in einem bestimmten Gültigkeitsbereich** suchen. Dazu unterscheide ich

relative Pfade nicht alles wird explizit aufgeschlüsselt, Lokalisierung abhängig davon, wo ich gerade im Dokument ‘stehe’. → Das ist z.B. nützlich, weil ich nicht wissen muss, wo genau ich bin. Ist aber auch schlecht, weil ich evtl. nicht genau weiß, wo ich gerade bin und andere Ergebnisse bekomme, als ich erwarte. Falls man sich nicht sicher ist, lieber volle Pfade. In gewissen Funktionen von XSLT später unumgänglich und keine große Sache.

Pfade & Scope ii

Exkurs: Scope Es kann auch sein, dass ich z.B. innerhalb einer `<xsl:for-each>`-Abfrage mit einer Sub-Menge des Dokuments weiterarbeite. Darin navigiere ich dann relativ, weil ich ja womöglich gar nicht wissen *kann*, wo die Elemente, auf die etwas zutrifft, eigentlich im Dokument stehen.

Pfade & Scope iii

Ansonsten kann ein relativer Pfad aber auch dazu führen, dass man ‘unerwünschte’ Dinge findet: evtl. hat man sich verkalkuliert, man findet mehr oder weniger als gewollt. Vielleicht steht man an der falschen Stelle und die gewünschte Abfrage ergibt im aktuell zur Verfügung stehenden Umfang (*scope*) keine Ergebnisse.

absolute Pfade Jede Angabe ist explizit aufgeschlüsselt. Ich bekomme wirklich nur den erwünschten Pfad. Wird komplizierter, wenn die Ergebnismenge aus mehreren unterschiedlichen Bereich kommt (hier dann Oder-Verknüpfungen verwenden ||).

XPath Funktionsweise i

Abfrageresultat = Knotenmenge Eine Pfad-Abfrage gibt eine Menge zurück. Mit einem Punkt kommt man an den Inhalt. Z.B. `matches(.,'([0-9]4)')`: Enthält ‘alles’ = aktueller Knoteninhalt eine vierstellige (Jahres-)Zahl? Antwort, ja/nein. Kein Zugriff auf die gefundenen Inhalte selbst, nur Ja/Nein-Antwort (!). `//*[matches(.,'([0-9]{4})')]` = ‘beliebige Knoten im ganzen Dokument, wo eine vierstellige Zahl vorkommt’ = gibt eine Knotenmenge zurück, der Elemente auf die die Antwort ”ja”= ”true”lautet.

XPath Funktionsweise ii

Prädikate = Bedingungen Prädikate, also Bedingungen, sind in eckigen Klammern. Dort drin kann ich eine Funktion abfragen, die ein Ergebnis liefert (z.B.) ja/nein.

Funktionen Funktionen können auch um den ganzen Ausdruck herum, doch dann wird im Normalfall auch keine Knotenmenge ausgegeben, sondern eine Antwort (abhängig von der speziellen Funktion). Wie eine Funktion in Mathe hat sie ein Ergebnis. Funktion ungleich Ergebnismenge, sondern Lösung.

```
function(//lokalisierung, parameter)
```

XPath Funktionsweise iii

Unterschied zwischen String und Knoten Wenn er mal im substring ist, hat er nur noch Zeichen, keine Elemente mehr. Auch bei Punkt(.).

Manche Funktionen geben eine Antwort, manche würden etwas verändern: die gehen nur in XSLT. Im XPath-Viewer geht nur das, was ich anzeigen kann. *`XPath failed due to: A sequence of more than one item is not allowed as the first argument of`*. Reinklicken kann man außerdem nur, wenn man eine Knotenmenge gewählt hat. Manche Funktionen nehmen nicht mehrere Knoten an, weil sie genau 1 Antwort geben sollen.

XPath i

```
//person//surname
//persName[@ref]
//persName/@ref
//persName[@ref="bla"]
//persName[contains(.,'bla')]
contains(.,'bla') = ja/nein, keine Knotenausgabe
```

Vorteil gegenüber normalen Suchen und Ersetzen: man kann Bedingungen abfragen (“gibt mir alle Vorkommnisse von Mina, aber nur in Kapiteln, wo Dracula nicht vorkommt”).

absoluter und relativer Pfad: in ein Div reinklicken und nur ’p’ suchen.

XPath ii

Absoluter expliziter Pfad: z.B. `title`, wenn im Header auch bibliograph. Angaben sind: `//title` gibt alle Resultate, aber man will ja nur den Dokumententitel, daher

```
/TEI/teiHeader/ [hier entweder jeden Schritt
eingeben oder z.B:]//titleStmt/title
```

Negation mit `!=` oder `not`

```
//persName[@ref]
//persName[not(@ref)] -- Personenname ohne ref-Attribut

//persname[@ref !='Mina'] -- alle außer Mina
```

XPath-Vorschau es ist möglich, dass nicht alle korrekten XPath-Abfragen auch in der Vorschau angezeigt werden können. Hier kommt dann keine Fehlermeldung und in XSLT sollte es gehen.

`substring()` Die String-Operationen (siehe ‘Zeichenketten-Funktionen’) gehen immer nur auf genau eine Zeichenkette und nicht etwa eine Menge.

Einstieg & Tipps ii

Vorgehen Immer erst in einem normalen Satz formulieren, was man will. Das dann dahingehend zerlegen was davon Pfad, was Bedingung und was Funktion sein könnte. Dann erst den Ausdruck schreiben.

Lokalen / globalen Scope testen Ausprobieren den Mauszeiger in ein `<div>` zu stellen und dann nur `p` in die XPath-Abfrage einzugeben: Man findet Knoten im lokalen Kontext. `//` gibt immer vom globalen Kontext aus an.

Antwort(menge) verstehen `not(//persname[@ref="#Mina"])` gibt 'false' zurück
→ weil er fragt:

Einstieg & Tipps iii

Gibt es im ganzen Dokument einen *persname[@ref="#Mina"]*?
Die Antwort ist ja.
Daraufhin verneint *not()* alles, d.h. es stimmt nicht, dass es keinen *persname[@ref="#Mina"]* im Dokument gibt.

Ergo → die Antwort ist *true*.

Beispiel zum Einstieg: Ausprobieren!

1. `//div[@type="chapter"]  -- count(//head)`

2. `//head -- //div[@type='chapter']/head`

3. `substring-after(//div[type=....],'Chapter')`

4. `distinct-values(//placeName/@xml:id)`
   `ODER distinct-values(//placeName)`
   `--> was ist der Unterschied?`

5. `distinct-values(//persName[@ref="#Mina"]`

7. `//p[contains(.,'Christmas')]`
   `ODER //p[matches(.,[A-Z][a-z])]`

8. `//div[@type="chapter"][@n="4"]/head`
   `ODER  //div[@type="chapter"][4]/head`

9. `//persName[@ref != "#Mina"]`
   `ODER \\persname[not(@ref='#Mina')]`

Beispiel zum Einstieg: Ausprobieren! ii

```
10. //div[@type='chapter'][1]/persName |
//div[@type='chapter'][1]/placeName
ODER //person | //place
```

XPath i

XML-Dokument = Baum
→ Zugriff / Navigation über Pfade

Wurzelknoten (/) ≠ Wurzelelement, dieses ist Kind des Wurzelknotens.
Weiters gibt es Attributknoten, Textknoten, Elementknoten, (+ Namensraum + Kommentar + Verarbeitungsanweisung)

Abfrage = Pfadstruktur, Ergebnis = Knotenmenge / Wert.
absolut (*/person/name*)
oder
relativ (*../../title*)

XPath ist Teil der XSL-Sprachfamilie (XPath, XSLT, XSL-FO, ...XQuery).

XPath ii

Syntax *Achsename::Knotentest[Prädikat]*

Achse = Beweungsrichtung

Knotentest: Welcher Knoten?

Prädikate: Bedingungen.

/child::person[attribute::gender]/child::name Langform

/person[@gender]/name ..... Kurzform

XPath iii

Achsen

::self ..... der aktuelle Knoten selbst (Kurzform: .)

::ancestor ..... alle Vorfahren des aktuellen Knotens

attribute:: ..... die Attribute des aktuellen Knotens (**Kurzform: @**)

child:: ..... direkte Kindelemente des aktuellen Knotens

descendant:: ..... Nachkommen des aktuellen Knotens (**Kurzform: //**)

parent:: ..... Elternknoten des aktuellen Knotens (**Kurzform: ../**)

preceding-sibling:: . vorhergehende Geschwister des aktuellen Knotens

following-sibling:: .. nachfolgende Geschwister des aktuellen Knotens

Navigieren in XML i

Zusätzliche Infos finden sich in den IDE Kurzreferenzen.

XPATH

1. im Kontext ausgewertet
2. absolute und relative Positionsangabe möglich (*dynamic & static context*
3. Namespaces
4. Funktionen, Variablen

Navigieren in XML ii

XPath-Basics

`/` ..... Wurzelknoten / Schritt weiter im Baum (nächster Lokalisierungsschritt)

`::*` ..... ::* = alle Elementtypen (kein bestimmter Elementname)

`@` ..... Attribut

`[Zahl]` ..... das wievielte Element (aus der Ergebnismenge)

`text()` ..... Textinhalt eines Elementes

`::Elementtyp` ..... Bedingung für den Elementtypen (Knotentest)

`//` ..... beliebig tief im Baum

`[text() = 'Textinhalt']` ..... Bedingung (Prädikat)

`.` ..... self

`(/../` ..... Eine Hierarchiestufe (mit beliebigem Namen) überspringen

`pfad//unterpfad` ..... beliebige Tiefe dazwischen

XPath Prädikate i

Bedingungen, die Submengen definieren = Prädikate

```
//person[@gender="female"]
//person[firstname = "Stefan"]
//person[firstname != "Tanja"]
//person[1]
//person[last()]/firstname
//person[position() = 2]
/participants/person[position()=5]/firstname
/participants/person[last()]/firstname
count(child::*)
```

XPath-Funktionen i

Funktionen – Knotenmenge

`position()` ..... Position des aktuellen Knotens, return: Zahl

`count()` ..... Anzahl der Knoten der übergebenen Knotenmenge

`last()` ..... letzter Knoten der gewählten Knotenmenge

XPath-Funktionen ii

Zeichenkettenfunktionen

substring-before(value, substring) .... Teilzeichenkette eines Textknotens (*value*), die vor dem angegebenen Zeichen (*substring*)

substring-before(., 'mina') ..... Bsp.

substring-after(value, substring) ..... wie before

substring(value, start, length) ..... mit Start- und Länge

string-length(.) ..... Länge des aktuellen Textknoten

concat(value1, value2, ...) ..... zusammenfügen

concat(nachname, ', ', vorname) ..... Bsp.

translate(Ziel, string, Ersetzung) ..... Ersetzen

noramlize-space(Ziel) ..... Entfernt trailing whitespace

XPath-Funktionen i

Boolsche Funktionen

`starts-with(value, substring)` ..... return: wahr/falsch

`starts-with(., 'D')` ..... wahr/falsch

`contains(value, substring)` ..... ist die Zeichenkette im aktuellen Textknoten enthalten, wird der Wert wahr zurückgeliefert

`not(Vergleich)` ..... gibt ja, falls nicht XY

RegEx-fähige Funktionen

`matches(Ziel, 'RegEx')` ..... wie `contains()`, aber nur exakte Treffer

`replace()` ..... wie `translate()` nur RegEX-fähig

`tokenize()` ..... Text in Einzelwörter auftrennen

Problembehebung

1. [Nichts geht] Im Zweifelsfall ist das Problem immer der **Namespace**. Oder man steht an einer anderen Stelle, als man dachte (bei der Verwendung relativer Pfade) → sicherheitshalber **absolute Pfade** wählen.
2. [Nichts gefunden] XML ist **hierarchisch**! Stehe ich an der **richtigen Stelle**? Erreiche ich den Grad an **Tiefe**, den ich will (im Zweifelsfall `/../` einbauen, um etwas unabhängiger von der Hierarchie zu sein.

Fortgeschrittene Anwendungen ii

3. [Zu wenig gefunden?] **Sprünge in der Pfad-Tiefe:** Ein Pfadausdruck (ohne zwischengeschaltetes *`pfad//unterpfad`*) überspringt keine Ebenen bzw. geht genau 1 Pfad weiter: Sind Zwischenebenen drin, die ich übersehen habe (z.B. eine weitere *`div`*-Verschachtelung)? Mit `//` dazwischen ist die Tiefe dann aber beliebig – falls eine spezielle gewünscht ist (z.B. 3 Stufen), dann lieber einen absoluten Pfad mit `/../../../` bauen.
4. [Ausdruck geht, passt das Ergebnis zur Anforderung?] Möglichst spezifische Abfragen formulieren: Finde ich alles, was ich suche? Matcht die Abfrage womöglich auch noch andere, unerwünschte Elemente?
5. [Relevant für XSLT:] Es gibt vordefinierte Verarbeitungsregeln, d.h. alle Textinhalte werden ausgegeben, außer man unterdrückt es explizit.

Vorbereitung von Transformationen

Mithilfe von XPath zuerst alle im Dokument vorkommenden möglicherweise zu verarbeitenden Dinge ausgeben lassen (je 1x, (*distinct-values()*-Funktion) :

distinct-values(//@/name())* ..... jede Art von Attribut
distinct-values(//@)* ..... jeder Attributwert
distinct-values(//@rend) ..... jeder *@rend*-Wert
distinct-values(//name()) ..... Elementtypen

Und Action! Macht das XPath-Übungsblatt durch. Behelft Euch gern mit Cheatsheet und Folien.

Transformationen mit XSLT

Was ist XSLT? i

XSL (*eXtensible Stylesheet Language*) ist eine Programmiersprache zur Transformation von XML-Daten. Diese erlaubt die Speicherung von abstrakten Basisdaten, aus denen dann unterschiedlichste Repräsentationen, z.B. die Strukturinformationen für HTML, automatisch generiert werden können (*single source*-Prinzip). **XSLT** (=XSL-Transformations) wird häufig synonym dazu verwendet, ist aber eigentlich nur ein Teil neben **XSL-FO** (*Formatting Objects*, zur Beschreibung von Druckdokumenten als PDF), das seit 2012 nicht mehr weiterentwickelt wird.

XSLT besteht aus Verarbeitungsmustern (sog. 'Schablonen' oder *templates*), wobei das XML-Dokument durchgegangen wird und Schablonen angewendet werden, sobald eine auf den aktuell angefundenen Inhalt passt. Die Verarbeitung durch den Parser

Was ist XSLT? ii

beginnt beim Wurzelknoten, weswegen es immer eine erste Schablone gibt, die auf alles passt:

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

Wird für angetroffene Inhalte kein Template gefunden, so werden die *Built-in*-Regeln angewendet, die die Inhalte der Elemente (inkl. Unterelemente!) ausgeben. Dies begegnet einem oft in dem Umstand, dass einfach der ganze Dokumenteninhalt drauflosgeprinted wird, was oft nicht eigentlich erwünscht ist. Z.B. möchte man oft ja nur den `body`-Inhalt eines Dokuments und nicht alle Metainformationen des `teiHeader`s unsortiert ausgedruckt haben. Dieses grundlegende Verhalten muss man daher bewusst unterbinden.

XSL(T): eXtensible Stylesheet Language (Transformations)

.XSL

Ein XSLT-Stylesheet (ist selbst ein XML-Dokument und) beschreibt Regeln für den Transformationsprozess einer Eingabedatei (XML-Dokument) in eine oder mehrere Ausgabedateien (XML, XHTML, Text).

Ausgabeformate sind z.B....

- **(x)HTML**
- **XML** → z.B. Word, TEI und andere XML-Standards, RDF, SVG
- **Text** → z.B. LaTeX (→ PDF), RTF, MARC.

Warnung am Rande: LaTeX z.B. ist auch Markup, hat aber andere geschützte Entitäten, bzw. andere Konventionen, wie man Dinge, wie etwa den Ampersand escaped. `xsl:output` und nachkontrollieren!

XSLT Ressourcen i

IDE Kurzreferenzen ● W3Schools-Tutorial

Einstieg i

In Oxygen sowohl ein zu transformierendes XML als auch ein neues XSL-Stylesheet eröffnen.

Auto-generiertes Start-Dokument

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:xs="http://www.w3.org/2001/XMLSchema"
4      exclude-result-prefixes="xs"
5      version="2.0">
6
7  </xsl:stylesheet>
```

Einstieg ii

Dann in die XML-Datei reinklicken und ein Transformationsszenario konfigurieren (Schraubenzieher-Symbol):

1. ‘Neu’ auswählen (XSLT-Transformation).
2. Reiter XSLT: XML-Datei aussuchen, XSL-Datei aussuchen (kann man sonst auch auf massenweise Dokumente anwenden (Projekt-Optionen).
3. Transformator wählen: Saxon (9er Version, sonst egal welcher).
4. Reiter XSL-FO: Hier egal, weil das eine andere Transformation als XSLT ist (zum PDFs machen, was alternativ zu LaTeX dort gehen würde).
5. Reiter Ausgabedatei: ‘Datei speichern unter’ → Grünen Pfeil anklicken (*`${cfn}`* auswählen. Dann *`-transform.xml`* hinzufügen, damit die Orignaldatei nicht überschrieben wird. (Bei erneuter Transformation wird diese Datei immer wieder überschrieben, außer man benennt sie um).
6. Wahlweise ‘Im Editor öffnen’ und anzeigen als XML, falls XML oder XHTML, falls HTML.
7. Ok, dann anwenden.

Einstieg iii

Damit wird nun unser, bishe noch sehr leeres, Stylesheet auf die Datei angewendet. Was fällt auf?

Das sich zeigende Verhalten ist das Standard-Verhalten von XSLT, wenn es keine zutreffenden Verarbeitungsregeln findet.

Standard-Transformationsablauf i

Leeres Match-Root-Template

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:xs="http://www.w3.org/2001/XMLSchema"
4      exclude-result-prefixes="xs"
5      version="2.0">
6
7      <xsl:template match="/">
8          <!-- und jetzt? -->
9      </xsl:template>
10
11 </xsl:stylesheet>
```

Mit dem folgenden äußern wir, dass wir möchten, dass die Standard-Verarbeitungsweise wieder verwendet wird. Der angefundene Text wird ohne Elemente hineingeschrieben. Achtung, das ist jetzt nicht eigentlich ein XML-Dokument, weil es keine Elemente oder hierarchische Struktur hat!

Standard-Transformationsablauf i

Standard-Regeln mit `apply-templates`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:xs="http://www.w3.org/2001/XMLSchema"
4      exclude-result-prefixes="xs"
5      version="2.0">
6
7      <xsl:template match="/">
8          <xsl:apply-templates/>
9      </xsl:template>
10
11 </xsl:stylesheet>
```

Standard-Transformationsablauf ii

Struktur ‘hardcoden’

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:xs="http://www.w3.org/2001/XMLSchema"
4      exclude-result-prefixes="xs"
5      version="2.0">
6
7      <xsl:template match="/">
8          <xml>
9              <xsl:apply-templates/>
10         </xml>
11     </xsl:template>
12
13 </xsl:stylesheet>
```

XSLT-Standardablauf II i

Statt die Elementenhierarchie mühsam händisch wieder reinschreiben zu müssen, können wir auch Templates (‘Schablonen’) definieren, die immer aktiv werden, wenn ein bestimmtes Element gefunden wird. Lies: Wann immer Du *<p/>* findest, mach XY. Wir deklarieren vorher noch einen Namespace (*xmlns*), damit unsere TEI-Dateien erkannt werden:

```
xmlns:t="http://www.tei-c.org/ns/1.0"
```

XSLT-Standardablauf II ii

Elemente müssen jetzt immer mit *t:* davor angesprochen. *t:p* lies: *<p>* aus dem Namespace, der mit *t:* abgekürzt ist (siehe Stylesheet-Deklaration oben).

XSLT-Standardablauf II iii

p-Elemente matchen

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:xs="http://www.w3.org/2001/XMLSchema"
4      xmlns:t="http://www.tei-c.org/ns/1.0"
5      exclude-result-prefixes="xs"
6      version="2.0">
7
8      <xsl:template match="/">
9          <xml>
10             <xsl:apply-templates/>
11         </xml>
12     </xsl:template>
13
14     <xsl:template match="t:p">
15         <neu>
16             <xsl:apply-templates/>
17         </neu>
18     </xsl:template>
19
20 </xsl:stylesheet>
```

Alle *p*s, inklusive ihrem Inhalt (dank `apply-templates`) werden erhalten, aber sie heißen nun statt `<p>` `<neu>` (sinnfrei & zu Demonstrationszwecken). Dieses Vorgehen jedenfalls nennt sich das sogenannte **Push-Paradigma**.

Das Gegenstück dazu, das **Pull-Paradigma** sehen wir jetzt beim hinzugefügten `<head>`. In dieses neu erzeugte Element holen wir uns bewusst den Inhalt vom `<title>`-Element rein (Achtung, hier XPath-Wissen anwenden: Wenn es mehrere `<title>`-Elemente gibt, nimmt er *alle*. Im Zweifelsfall explizit und mist absoluten Pfaden arbeiten. Außerdem ist `value-of select=` nicht dasselbe wie `apply-templates`. `apply-templates` schaut erst, ob es (weiter *unten* im Code!) noch Regeln gibt, die auf die Elemente zutreffen, die im gefundenen Ding eingeschachtelt sind und wendet ggf. diese an.

`value-of select=` kopiert nur den aktuell angesprochenen Textinhalt; tiefere Verschachtelungen gehen verloren. `value-of select="."` gibt den aktuellen Knoteninhalt (Text). `attribut={@rend}` ist eine Kurzschreibweise dafür bei Attributen.

XSLT-Basics III iii

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:t="http://www.tei-c.org/ns/1.0"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <xml>
            <head><xsl:value-of select="//t:title"/></head>
            <xsl:apply-templates/>
        </xml>
    </xsl:template>

    <xsl:template match="t:p">
        <neu attribut={@rend}>
            <xsl:apply-templates/>
        </neu>
    </xsl:template>

</xsl:stylesheet>
```

Inhalte löschen i

Per Automatismus werden in XSLT alle Inhalte (Werte) erhalten – aber wir haben schon gesehen, dass man mit leeren Regeln bewusst löschen kann. Wir löschen im Beispiel also probehalber alle *<hi>* (inklusive Inhalt!). Hier darauf achten, dass man nicht versehentlich auf die Quelldatei schreibt, sonst ist alles weg (siehe Transformationsszenario konfigurieren).

Bewusstes Löschen durch leere Regel

```
1  <xsl:template match="t:hi">
2      <!-- loeschen -->
3  </xsl:template>
```

Inhalte löschen ii

Das sind im Grunde schon alle Regeln, die man wissen muss. Daneben gibt es natürlich noch mehr Funktionen, falls man kompliziertere Sachen machen will. Diese finden sich im Folgenden erklärt, werden aber evtl. gar nicht gebraucht.

Folgend noch ein Beispiel...

Beispiel i

Beispiel ii

Gedicht → HTML-Webansicht

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml" version="2.0">

    <xsl:template match="/">
        <html>
            <head>
                <title><xsl:value-of select="div/head" /></title>
            </head>
            <body>
                <xsl:apply-templates select="div" />
            </body>
        </html>
    </xsl:template>

    <xsl:template match="div">
        <h1><xsl:value-of select="head" /></h1>
        <div>
            <xsl:apply-templates select="lg" />
        </div>
    </xsl:template>

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

    <xsl:template match="l">
        <xsl:value-of select="." />
        <br />
    </xsl:template>
</xsl:stylesheet>
```

Variablen

```
<xsl:variable name="platzhalter" select="inhalt">
```

XSLT führt bei Variablen einen sog. *pass by value*, nicht *by reference* aus, d.h. es wird nur der Wert übergeben, den das ausgewählte Element zu einem Zeitpunkt hat. Wenn ich die Variable zu früh setze, wähle ich dabei womöglich einen fixierten Wert. Also am besten immer so nah wie möglich am ‘Ziel’.

Weitere Funktionen ii

Attribute `<xsl:attribute>` dient zum Setzen von Attributen im Resultat-Dokument. Dies ist nicht nötig, wenn man nur einen Inhalt will, dann kann auch die Kurzschreibweise verwendet werden.

plain text `<xsl:text>` setzt Text ‘as is’, d.h. fügt nicht unerwünschte Leerzeichen ein, was sonst passieren. Überhaupt geht XSLT mit Whitespace relativ frei um, außer man verbietet es explizt (`preserve-space`.

Sortieren Es gibt auch `<xsl:sort>` zum Sortieren.

Bedingungen `<xsl:choose>` zum Auswählen unterschiedlicher Verarbeitung je nachdem, was angefunden wird. `<xsl:if>` führt Anweisungen nur aus, wenn eine Bedingung erfüllt ist (z.B. falls es Element XY gibt, oder ein Element Inhalt XY hat).

Weitere Funktionen iii

Personenliste in HTML mit `for-each`

```
1  <ul> <!-- ungeordnete Liste in HTML -->
2  <xsl:for-each select="t:persName">
3      <li> <!-- Listenelement -->
4      <xsl:value-of select="." />
5      </li>
6  </xsl:for-each>
7  </ul>
```

Weitere Funktionen iv

Attribute setzen

```
<xsl:for-each select="t:persName">
    <span class={@rend}><xsl:value-of select="."></span>
</xsl:for-each>
<!-- ODER -->

<span>
    <xsl:attribute name="id">
        <xsl:value-of select="@rend"><xsl:text>-person</xsl:text>
    <xsl:attribute>
    <xsl:attribute name="interpretation">
        <xsl:value-of select="concat(@rend,@ana)">
    <xsl:attribute>
    <xsl:apply-templates /> <!-- für den eig. Inhalt -->
<span>
<!-- Resultat z.B.
    <span id="monster-person" interpretation="monster-boese">
    Weird Sisters </span> -->
```

Weitere Funktionen v

For-each-group

```
1  <!-- könnte man auch nach Alphabet sortieren oder so. Nur gezielt verwenden, macht in
   ↪  komplexen Dokumenten oft ominöse Fehler. -->
2
3  <xsl:for-each-group select"dings" group-starting-with="dings[@rend="Startdings]]"
4      <xsl:apply-templates select="current-group()">
5      <xsl:value-of select="current()/dingens">
6  </xsl:for-each-group>
```

Mehrere Quelldokumente zusammenführen (document( ))

```
1  <xsl:apply-templates select="document('Letter1_TEI.xml')/tei:TEI/tei:text/tei:body/tei:div"/>
```

Beispiel: Wähle jede Person (`//persName`) und generiere eine Aufzählung (`<ul>`) der Nachnamen (`lastname`):

Schleifen: for-each

```
1  <ul>
2  <xsl:for-each select="//persName">
3      <xsl:sort select="lastname" order="ascending" />
4      <li> <xsl:value-of select="lastname"/> </li>
5  </xsl:for-each>
6  </ul>
```

`xsl:if` führt Anweisungen nur aus, wenn die Bedingung in `@test` erfüllt ist.

Bedingungen I: Falls

```
1  <xsl:if test=" xpath-ausdruck "> ... </xsl:if>
2
3  <xsl:for-each select="//book">
4      <xsl:if test=" author = 'Cicero' ">
5          <li><xsl:value-of select="title"/></li>
6      </xsl:if>
7  </xsl:for-each>
```

Auch die Unterscheidung mehrerer Fälle ist möglich. So kann ich z.B. für das HTML unterschiedlichen Personen unterschiedliche Farben zuweisen. Dazu frage ich, in *@test* welche ID ein angetroffener *persName* hat und verarbeite dann je nachdem unterschiedlich.

Bedingungen II: Auswählen

```
1  <xsl:choose>
2      <xsl:when test=" xpath-ausdruck "> ... </xsl:when>
3      <xsl:otherwise> ... </xsl:otherwise>
4  </xsl:choose>
```

Man kann zudem sortieren (*xsl:sort*), kopieren (*xsl:copy - xsl:copy-of*)und Variablen verwenden.

XSL-Paradigmen: Push i

Push-Paradigma

```
1   <xsl:apply-templates select="xpath-ausdruck"/>
```

Geeignet vor allem: Für das Verarbeiten des Text-Bodys. Hier werden angetroffene Dinge einfach verarbeitet, wann immer sie auftreten. Man muss deren Reihenfolge nicht kennen. Die Dokumentenstruktur des ‘Herkunftsdokuments’ wird weitgehend erhalten (soweit es eben in den Template-Regeln definiert ist).

Push-Paradigma

```
1   <xsl:template match="/">
2       <xsl:apply-templates/>
3   </xsl:template>
```

XSL-Paradigmen: Push ii

Push Processing Jedes Element bekommt eine Regel oder *built-ins* werden verwendet. Nützlich falls Input- und Outputdokument dieselbe Struktur haben sollen.

Call Template / Push-Methode

```
1  <xsl:template match="/">
2      <xsl:call-template name="irgendein-name ">
3  </xsl:template>
4  <xsl:template name="irgendein-name">
5      ... macht etwas ...
6  </xsl:template>
```

XSL-Paradigmen: Pull i

Pull-Paradigma

```
1   <xsl:call-templates name="name-eines-templates "/>
```

Geeignet vor allem: Herausziehen von Metadaten aus dem *teiHeader*, um sie irgendwo extra anzuzeigen. Nochmals ‘über das Dokument drübergehen’ und z.B. einen Personenindex machen, den man an einer bestimmten Stelle hingesetzt haben will (*<xsl:for-each>* ist z.B. ein typischer Befehl der Push-Methode.

XSL-Paradigmen: Pull ii

Pull Processing Gewisse Knoten explizit auswählen. Kontrolle darüber, nur bestimmte Knoten im Output-Dokument weiterzuverwenden. Gut geeignet, falls das Output-Dokument eine völlig andere Struktur haben soll, als der Inhalt oder nur sehr selektiv Inhalte übernommen werden sollen (z.B. nur eine Liste aller enthaltenen Personennamen braucht nicht den ganzen restlichen Text dazu verarbeiten).

Pull-Paradigma

```
1  <xsl:value-of select=''pattern''/>
2  <xsl:apply-templates select=''pattern''/>
3  <xsl:for-each select=''pattern''/>
```

Paradigmen in der Praxis i

In der Praxis wird meist eine Kombination von beiden verwendet: Die Metadaten des *teiHeader* werden, z.B. in der Form eines Zitiervorschlags, mit dem Pull-Paradigma ausgelesen. Dann geht man an die Stelle des Output-Dokuments, wo der hauptsächliche Inhalt (*body*) hin soll und wählt aus, dass gar nicht mehr das ganze Dokument automatisch per Pull-Paradigma verarbeitet werden soll, sondern nur alles ab dem *body*.

apply-templates select

```
1  <xsl:apply templates select="//t:body" />
2  <xsl:apply-templates select="head" mode="toc"/>
```

Anderes Beispiel: Kapitelüberschriften innerhalb des Buches per Push-Paradigma, für das Inhaltsverzeichnis im Pull-Paradigma.

Paradigmen in der Praxis ii

Mithilfe von *mode*s kann man zudem noch unterscheiden, dass in gewissen Situationen andere Templateregeln angewandt werden.

<xsl:template match=""> verarbeitet ein Element, wo immer es angetroffen wird.

<xsl:value-of select=""> gibt den Inhalt eines Elements als String aus – wenn noch Kindelemente sind, gehen die Knoten/Auszeichnungen verloren, weil nur mehr plain text kopiert wird.

<xsl:apply-templates> geht die Regeln durch, wo er dann alles findet, was unter *"match"* definiert ist – er sucht aber Unter-Verschachtelungen nur weiter, wenn man das explizit so angibt (mit weiterem *apply-templates*).

Facts i

Vorgehen bei Stylesheets Regelwerk nach Top-down-Prinzip, ausgehend vom Wurzelelement des Quelldokuments.

Wurzelelement `xsl:stylesheet` XSLT ist selbst XML, daher hat es auch ein alles umgebendes Wurzelelement. Hier werden einige Dinge deklariert (Namensräume, XSLT-Processor, etc.).