Chapter 8
The Transformation Language XSL

8.1 XSL: Extensible Stylesheet Language

- developed from
  - CSS (Cascading Stylesheets) scripting language for transformation of data sources to HTML or any other optical markup, and
  - DSSSL (Document Style Semantics and Specification Language), stylesheet language for SGML. Functional programming language.
- idea: rule-based specification how elements are transformed and formatted *recursively*:
  - Input: XML
  - Output: XML (special case: HTML)
- declarative/functional: XSLT (XSL Transformations)

APPLICATIONS

- XML → XML
  - Transformation of an XML instance into a new instance according to another DTD,
  - Integration of several XML instances into one,
  - Extraction of data from an XML instance,
  - Splitting an XML instance into several ones.
- XML → HTML
  - Transformation of an XML instance to HTML for presentation in a browser
- XML → anything
  - since no data structures, but only ASCII is generated, LaTeX, postscript, pdf can also be generated
  - ... or transform to XSL-FO (Formatting objects).

THE LANGUAGE(S) XSL

Partitioned into two sublanguages:

- functional programming language: XSLT
  "understood" by *XSLT-Processors* (e.g. xt, xalan, saxon, xsltproc ...)
- generic language for document-markup: XSL-FO
  "understood" by XSL-FO-enabled *browsers* that transform the XSL-FO-markup according to an internal specification into a direct (screen/printable) presentation.
  (similar to LaTeX)

- XSL itself is written in XML-Syntax.
  It uses the namespaces "xsl:" and "fo:"
- XSL programs can be seen as XML data.
- it can be combined with other languages that also have an XML-Syntax (and an own namespace).

APPLICATION: XSLT FOR XML → HTML

- the prolog of the XML document contains an instruction that specifies the stylesheet to be used:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="mondial-simple.xsl"?>
<!DOCTYPE mondial SYSTEM "mondial.dtd">
<mondial>
    ...
</mondial>
```

- if an (XSL-enabled) browser finds an XML document with a stylesheet instruction, then the XML document is processed according to the stylesheet (by the browser's own XSLT processor), and the result is shown in the browser.
  (e.g., `http://ap34.ifi.informatik.uni-goettingen.de/XSLT/mondial-with-stylesheet.xml`)
- Remark: not all browsers support the full functionality (id()-function)
- in general, for every DTD there is a suitable stylesheet how to present such documents.

8.2 XSLT: Syntax and Semantics

- Each XSL-stylesheet is itself a valid XML document,

  <?xml version="1.0">
  <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  ...
  </xsl:stylesheet>

- contains elements of the namespace xsl: that specify the transformation/formatting,
- contains literal XML for generating elements and attributes of the resulting document,
- uses XPath expressions for accessing nodes in an XML document. XPath expressions (mostly) occur as attribute values of `<xsl:...>` elements, (e.g., `<xsl:copy-of select='`*xpath*`'>`)
- XSL stylesheets/programs recursively generate a result tree from an XML input tree.

8.2.1 XSLT: Flow Control by Templates

The stylesheet consists mainly of *templates* that specify the instructions *how* elements should be processed:

- xsl:template:

  <xsl:template match="*xsl-pattern*">
  content
  </xsl:template>

- *xsl-pattern* is an XPath expression without use of "*axis*::" (cf. Slide 196). It indicates for which elements (types) the template is applicable:
  a node x satisfies *xsl-pattern* if there is some ancestor node k of x, such that x is in the result set of *xsl-pattern* for k as context node.

  (another selection takes place at runtime when the nodes are processed for actually deciding to apply a template to a node).
- *content* contains the XSL statements for generation of a fragment of the result tree.

TEMPLATES

- <xsl:template match="city">
  <xsl:copy-of select="current()"/>
  </xsl:template>
  is a template that can be applied to cities and copies them unchanged into the result tree.
- <xsl:template match="lake|river|sea"> ... </xsl:template>
  can be applied to waters.
- <xsl:template match="country/province/city"> ... </xsl:template>
  can be applied to city elements that are subelements of province elements that in course are subelements of country elements.
- <xsl:template match="id('D')"> ... </xsl:template>
  can be applied to the element whose ID is "D".
- <xsl:template match="city[population > 1000000]"> ... </xsl:template>
  can be applied to city elements that have more than 1000000 inhabitants.

EXECUTION OF TEMPLATES: "TREE WALKING"

- xsl:apply-templates:
  <xsl:apply-templates select="*xpath-expr*"/>
- *xpath-expr* is an XPath expression that indicates for which elements (starting from the node where the current template is applied as context node) "their" template should be applied.
  Note that elements are processed in order of the final axis of the select expression.
- By <xsl:apply-templates> elements inside the content of <xsl:template> elements, the hierarchical structure of XML documents is processed
  - simplest case (often in XML → HTML): depth-first-search
  - can also be influenced by the "select" attribute: "tree jumping"
- if all subelements should be processed, the "select" attribute can be omitted.
  <xsl:apply-templates/>

TEMPLATES

- <xsl:apply-templates select="country"/>
  processes all country subelements of the current context element.
- <xsl:apply-templates select="country/city"/>
  processes all city subelements of country subelements of the current context element,
- <xsl:apply-templates select="/mondial//city[population > 1000000]"/>
  processes all city elements that are contained in Mondial and whose population is more than 1000000,
- <xsl:apply-templates select="id(@capital)"/>
  processes the element whose ID equals the value of the capital-(reference) attribute of the current context element.

TEMPLATES

- One template must be applicable to the root element for initiating the processing:
  - <xsl:template match="*name_of_the_root_element*">
  - <xsl:template match="/">
  - <xsl:template match="*">

RULE-BASED "PROGRAMMING"

- local semantics: templates as "rules"
- global semantics: built-in implicit tree-walking combines rules

Templates: Example

Presentation of the country information as a table (→ HTML)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="mondial">
<html> <body> <table>
    <xsl:apply-templates select="country"/>
  </table>  </body> </html>
</xsl:template>

<xsl:template match="country">
<tr><td> <xsl:value-of select="name"/> </td>
  <td> <xsl:value-of select="@car_code"/> </td>
  <td align="right"> <xsl:value-of select="population"/> </td>
  <td align="right"> <xsl:value-of select="@area"/> </td>
 </tr>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-simple.xsl]

Templates: Example

Presentation of the country and city information as a table:

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="mondial">
 <html><body><table>
     <xsl:apply-templates select="country"/>
 </table></body></html>
</xsl:template>

<xsl:template match="country">
<tr valign="top">
  <td><xsl:value-of select="name"/></td>
  <td><xsl:value-of select="@car_code"/></td>
  <td align="right"><xsl:value-of select="population"/></td>
  <td align="right"><xsl:value-of select="@area"/></td>
  <td valign="top">
    <table><xsl:apply-templates select=".//city"/></table>
  </td>
</tr>
</xsl:template>

<xsl:template match="city">
<tr> <td width="100"><xsl:value-of select="name"/></td>
     <td align="right" width="100">
        <xsl:value-of select="population[1]"/>
     </td>
</tr>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-nested.xsl]

TEMPLATES: EXAMPLE

The following (transformation: XML → XML) stylesheet copies all country and city elements from Mondial and outputs first all country elements, and then all city elements as top-level elements:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/TR/WD-xsl">
   <!-- Template that copies elements -->
   <xsl:template match="city|country">
      <xsl:copy-of select="current()"/>
   </xsl:template>
   <template match="mondial">
      <!-- apply templates: first countries -->
      <xsl:apply-templates select="/mondial/country">
      <!-- apply templates: then cities -->
      <xsl:apply-templates select="//country/city | //country/province/city"/>
   </template>
</xsl:stylesheet>
```


HANDING NON-XSLT NAMESPACES IN XSLT

declare the namespace: in the surrounding <xsl:stylesheet> element:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ht="http://www.w3.org/1999/xhtml"
                version="1.0">
```

tells the XSL processor that the namespace bound to 'http://www.w3.org/1999/xhtml' is denoted by "ht:" in this document.
(and <ht:body> is different from <body>)

```
<!-- XSLT Stylesheet for XPath queries against remote documents
 call: xalan -xsl web-queries.xsl -in some-xml-file
       xsltproc web-queries.xsl some-xml-file -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ht="http://www.w3.org/1999/xhtml"
                version="1.0">
<xsl:template match="*">
 <result>
 <!-- *** query follows here, must start with document('...') *** -->
 <xsl:copy-of select="
  document('http://www.dbis.informatik.uni-goettingen.de/index.html')//ht:li
   "/>
 <!-- *** query ends here *** -->
 </result>
</xsl:template>
</xsl:stylesheet>

<!--
Queries
  against a remote XML document:
  document('http://www.dbis.informatik.uni-goettingen.de/Mondial/mondial.xml')//country[@car_code='D']
  against an XHTML document (note the use of a namespace! - definition above)
  document('http://www.dbis.informatik.uni-goettingen.de/index.html')//ht:li
-->
```

[Filename: XPath/web-queries.xsl]

TEMPLATES

Difference between:

1. <xsl:template match="*xsl-pattern*">
   content
   </xsl:template>
2. <xsl:apply-templates select="*xpath-expr*"/>

- select="..." is evaluated wrt. the current context node (selects which elements are addressed by the given XPath expression),
- match="..." is evaluated wrt. the document structure starting from "below" (checks if the document structure matches with the pattern),
- xsl:apply-templates selects nodes for application by its *xpath-expr*, and then the suitable templates are applied,
- the order of templates has no effect on the order of application (document order of the selected nodes).

Templates

Exercise 8.1
Describe the difference between the following stylesheet fragments:

1. <xsl:template match="city">
   <xsl:copy-of select="current()"
   </xsl:template>
   <xsl:apply-templates select="//country/city"/>
   <xsl:apply-templates select="//country/province/city"/>
2. <xsl:template match="country/city">
   <xsl:copy-of select="current()"
   </xsl:template>
   <xsl:template match="country/province/city">
   <xsl:copy-of select="current()"
   </xsl:template>
   <xsl:apply-templates select="//country/city|//country/province/city"> □

Conflicts between Templates

When using non-disjoint match-specifications of templates (e.g. *, city, country/city, city[population>1000000]) (including possibly templates from imported stylesheets), several templates are probably applicable.

- in case that during processing of an <xsl:apply-templates>-command several templates are applicable, the one with the most specific match-specification is chosen.
- defined by *priority rules* in the XSLT spec (that also define priorities between incomparable patterns)

Overriding (since XSLT 2.0)

The above effect is similar to *overriding* of methods in object-oriented concepts: always take the most specific implementation

- <xsl:next-match>: apply the next-lower-specific rule (among those defined in the same stylesheet)
- <xsl:apply-imports>: apply the next-lower-specific rule (among those defined in imported stylesheets (see later))

Resolving Template Conflicts Manually

Process a node with different templates depending on situation:

- associating "modes" with templates and using them in apply-templates

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="mondial">
  <xsl:apply-templates select="country[@area>1000000]"/>
  ... and now the second part ...
  <xsl:apply-templates select="country[@area>1000000]" mode="bla"/>
</xsl:template>
<xsl:template match="country">
  <firsttime> <xsl:value-of select="name"/> </firsttime>
</xsl:template>
<xsl:template match="country" mode="bla">
  <secondtime> <xsl:value-of select="name"/> </secondtime>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-modes.xsl]

Named Templates

Named templates serve as macros and can be called by their name.

- xsl:template with "name" attribute:

  <xsl:template name="*name*">
  content
  </xsl:template>

  - *name* is an arbitrary name
  - *content* contains xsl-statements, e.g. xsl:value-of, which are evaluated against the current context node.

- xsl:call-template

  <xsl:call-template name="*name*"/>

- Example: Web pages – templates for upper and left menus etc.

8.2.2 XQuery and XSLT

- both are declarative, functional languages ...
- ... with completely different strategies:
  - XQuery: nesting of the return-statement directly corresponds to the structure of the result
  - XSLT: the nested processing of templates yields the structure of the result.

XSLT

- modular structure of the stylesheets
- extensibility and reuse of templates
- flexible, data-driven evaluation

XQuery

- better functionality for joins (for $a in ..., $b in ...)
- XSLT: joins must be programmed explicitly as nested loops (xsl:for-each)

TRANSLATION XSLT → XQUERY

- each template is transformed into an FLWR statement,
- inner template-calls result in nested FLWR statements inside the return-clause
- genericity of e.g. <apply-templates/> cannot be expressed in XQuery since it is not known which template is activated

⇒ the more flexible the schema (documents), the more advantages show up for XSLT.

Exercise 8.2

- Give XQuery queries that do the same as mondial-simple.xsl and mondial-nested.xsl.
- Give an XQuery query that does the same as the stylesheet on Slide 308. □

8.2.3 XSLT: Generation of the Result Tree

Nodes can be inserted into the result tree by different ways:

- literal XML values and attributes,
- copying of nodes and values from the input tree,
- generation of elements and attributes by constructors.

Configuring Output Mode

- recommended, top level element (see xsl doc. for details):
  <xsl:output method="xml|html|xhtml|text" indent="yes|no"/>
  (not yet supported by all XSLT tools; saxon has it)

Generation of Structure and Contents by Literal XML

- All tags, elements and attributes in the content of a template that do not belong to the xsl-namespace (or to the local namespace of an xsl-tool), are literally inserted into the result tree.
- with <xsl:text> *some_text*</xsl:text>, text can be inserted explicitly (whitespace, e.g. when generating IDREFS attributes).

Generation of the Result Tree

Copying from the Input Tree

- <xsl:copy>*contents*</xsl:copy>
  copies the current context node (i.e., its "hull"): all its namespace nodes, but *not* its attributes and subelements (note that contents can then be generated separately).
- <xsl:copy-of select="*xpath-expr*"/>
  copies the result of *xpath-expr* (applied to the current context) unchanged into the result tree.
- <xsl:value-of select="*xpath-expr*" [separator="*char*"]/>
  generates a text node with the value of *xpath-expr*.
  Applied to multiple nodes, the partial results are separated by *char* (default: space).
  [note: the latter changed from XSLT 1.0 (apply only to 1st node) to 2.0]

Exercise 8.3
Consider the differences between <xsl:copy/>, <xsl:copy-of select="current()"/> and <xsl:value-of select="current()"/>.
In which cases do two commands have the same result? □

GENERATION OF THE RESULT TREE

Example:

```
<xsl:template match="city">
   <mycity>
      <xsl:value-of select="name"/>
      <xsl:copy-of select="longitude|latitude"/>
   </mycity>
</xsl:template>
```

- generates a mycity element for each city element,
- the name is inserted as #PCDATA content,
- the subelements longitude and latitude are copied:

```
<mycity>Berlin
  <longitude>13.3</longitude>
  <latitude>52.45</latitude>
</mycity>
```


GENERATION OF THE RESULT TREE

For inserting attribute values,

<xsl:value-of select="*xpath-expr*"/>

cannot be used *directly*. Instead, XPath expressions have to be enclosed in {...}:

```
<xsl:template match="city">
   <mycity id="{@id}">
      <xsl:value-of select="name"/>
      <xsl:copy-of select="longitude|latitude"/>
   </mycity>
</xsl:template>
```

GENERATION OF THE RESULT TREE

Example:

```
<xsl:template match="city">
  <mycity source="mondial"
      country="{ancestor::country/name}">
    <xsl:apply-templates/>
  </mycity>
</xsl:template>
```

- generates a “mycity” element for each “city” element,
- constant attribute “source”,
- attribute “country”, that indicates the country where the city is located,
- all other attributes are omitted,
- for all subelements, suitable templates are applied.

XSLT: GENERATION OF THE RESULT TREE

Generation of Elements and Attributes

- <xsl:element name=“*xpath-expr*”>
  content
  </xsl:element>
  generates an element of element type *xpath-expr* in the result tree, the content of the new element is *content*. This allows for computing element names.
- <xsl:attribute name=“*xpath-expr*”>
  content
  </xsl:attribute>
  generates an attribute with name *xpath-expr* and value *content* which is added to the surrounding element under construction.
- With <xsl:attribute-set name=“*name*”> xsl:attribute* </xsl:attribute-set>
  attribute sets can be predefined. They are used in xsl:element by
  use-attribute-sets=“attr-set$_1$ ... attr-set$_n$”

8.2.4 XSLT: Control Structures

... so far the “rule-based”, clean XSLT paradigm with implicit recursive semantics:

- templates: recursive control of the processing

... further control structures inside the content of templates:

- iterations/loops
- branching

Design of XSLT Command Elements

- semantics of these commands as in classical programming languages (Java, C, Pascal, Basic, Cobol, Algol)
- Typical XML/XSLT design: element as a command, further information as attributes or in the content (i.e., iteration specification, test condition, iteration/conditional body).

Iterations

For processing a list of subelements or a multi-valued attribute, local iterations can be used:

<xsl:for-each select=“*xpath-expr*”>
    content
</xsl:for-each>

- inside an iteration the “iteration subject” is not bound to a variable (like in XQuery as `for $x in` *xpath-expression*), but
- the current node is that from the xsl:for-each, not the one from the surrounding xsl:template
- an xsl:for-each iteration can also be used for implementing behavior that is different from the templates “matching” the elements (instead of using modes).

GENERATION OF IDREFS ATTRIBUTES

- XML source: “border” subelements of “country” with an IDREF attribute “country”:
  <border country=“*car_code*” length=“...”>
- result tree: IDREFS attribute country/@neighbors that contains all neighboring countries
- three ways how to do this (note: 1st and 2nd require XSLT 2.0)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"> <!-- version! -->
<xsl:template match="*"><xsl:apply-templates select="country"/></xsl:template>
<xsl:template match="country">
  <country neighbors1="{border/@country}"> <!-- note: adds whitespace as separator -->
    <xsl:copy-of select="attribute::*"/>
    <xsl:attribute name="neighbors2">
      <xsl:value-of select="border/@country"/> <!-- default separator: whitespace -->
    </xsl:attribute>
    <xsl:attribute name="neighbors3">
      <xsl:for-each select="border">
        <xsl:value-of select="@country"/><xsl:text> </xsl:text> <!-- explicit -->
      </xsl:for-each>
    </xsl:attribute>
   </country>
 </xsl:template></xsl:stylesheet>
```

[Filename: XSLT/mondial-neighbors.xsl]

XSLT: CONDITIONAL PROCESSING

- Simple Test:
  <xsl:if test=“*predicate*”> *content* </xsl:if>

Example:

```
<xsl:template match="country">
   <table>
      <tr> <th colspan="2"> <xsl:value-of select="name"> </th>
      </tr>
      <xsl:if test="@area">
         <tr>
            <td> Area: </td>
            <td> <xsl:value-of select="@area"> </td>
         </tr>
      </xsl:if>
      ⋮
   </table>
</xsl:template>
```

XSLT: CONDITIONAL PROCESSING

- Multiple alternatives:

<xsl:choose>
<xsl:when test="*predicate*$_1$">
content$_1$
</xsl:when>
<xsl:when test="*predicate*$_2$">
content$_2$
</xsl:when>
...
<xsl:otherwise>
content$_{n+1}$
</xsl:otherwise>
</xsl:choose>

8.2.5 XSLT: Variables and Parameters

Variables and parameters serve for binding values to names.

VARIABLES

- variables can be assigned only once (in their definition). A later re-assignment (like in C or Java) is not possible.
- variables can be defined as top-level elements which makes them visible in the whole document (as a constant).
- a variable definition can take place at an arbitrary position inside a template - such a variable is visible in all its siblings
  - e.g., a variable inside a <xsl:for-each>,
  - gets a new value for each iteration to store an intermediate value.

BINDING AND USING VARIABLES

- value assignment either by a "select" attribute (value is a string, a node, or a set of nodes)

  <xsl:variable name="*var-name*" select="*xpath-expr*"/>

- or as element content (then, the value can be a tree which is generated dynamically by XSLT)

  <xsl:variable name="*var-name*">
  content
  </xsl:variable>

- Usage: by select="$*var-name*"

Example: Variables

A simple, frequent use is to "keep" the outer current element when iterating by an xsl:for-each:

- Consider the previous "border"-example
- now: generate a table of neighbors

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="*">
  <table><xsl:apply-templates select="country"/></table>
</xsl:template>
<xsl:template match="country">
  <xsl:variable name="country" select="."/>
  <xsl:for-each select="border">
    <tr>
     <td><xsl:value-of select="$country/@car_code"/></td>
    <td><xsl:value-of select="@country"/></td>
   </tr>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-neighbors-table.xsl]

PARAMETERS

- ... similar to variables
- values are communicated to called templates by parameters,
- the definition of parameters is allowed *only at the beginning* of xsl:template elements. The defined parameter is then visible everywhere in the template body.
- the assignment of a value takes place in the calling <xsl:apply-templates> or <xsl:call-template> element.
- pure call-by-value, no call-by-reference possible.

Remark: since a parameter can be an element with substructures, theoretically, a single parameter is always sufficient.

COMMUNICATION OF PARAMETERS TO TEMPLATES

- Parameters are declared at the beginning of a template:

```
<xsl:template match="...">
   <xsl:param name="param-name"
              select="xpath-expr/">    <!-- with a default value -->
   ⋮
</xsl:template>
```

- the parameter values are then given with the template call:

```
<xsl:apply-templates select="xpath-expr1">
   <xsl:with-param name="param-name"
                   select="xpath-expr2"/>
</xsl:apply-templates>
```

- Often, parameters are propagated downwards through several template applications/calls.
  This can be automatized (since XSLT 2.0) by
  <xsl:param name="*param-name*" select="*xpath-expr*" tunnel="yes">

Example: Parameters

Generate a table that lists all organizations with all their members. The abbreviation of the organisation is communicated by a parameter to the country template which then generates an entry:

→ next slide

[Filename: orgs-and-members.xsl]

Exercise 8.4

- Extend the template such that it also outputs the type of the membership.
- Write an equivalent stylesheet that does not call a template but works explicitly with <xsl:for-each>.
- Give an equivalent XQuery query (same for the following examples). □

EXAMPLE (CONT'D)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="mondial">
    <html><body> <h2>Membership Table</h2>
      <table> <xsl:apply-templates select="organization"/>
      </table></body></html>
  </xsl:template>
  <xsl:template match="organization">
    <tr><td colspan="2"><xsl:value-of select="name"/></td></tr>
    <xsl:apply-templates select="id(members/@country)">
       <xsl:with-param name="the_org" select="name/text()"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="country">
    <xsl:param name="the_org"/>
    <tr><td><xsl:value-of select="$the_org"/></td>
        <td><xsl:value-of select="name/text()"/></td></tr>
  </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/orgs-and-members.xsl]

Example/Comparison of Mechanisms

Example: This example illustrates the implicit and explicit iterations, and the use of variables/parameters
[use file:XSLT/members1.xsl and develop the other variants]

- Generate a list of the form

```
<organization> EU <member>Germany</member>
                  <member>France</member> ... </organization>
```

  - using template-hopping [Filename: XSLT/members1.xsl]
  - using xsl:for-each [Filename: XSLT/members2.xsl]

- Generate a list of the form

```
<membership organization="EU" country="Germany"/>
```

  based on each of the above stylesheets.

  - template hopping: requires a parameter [Filename: XSLT/members3.xsl]
  - iteration: requires a variable [Filename: XSLT/members4.xsl]

A Powerful Combination: Variables and Control

<xsl:variable name="*var-name*">
content
</xsl:variable>

Any structure that is generated in *content* is then bound to the variable.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="mondial">
    <xsl:variable name="berlin">
      <bla>
       <xsl:copy-of select="//city[name='Berlin']"/>
      </bla>
    </xsl:variable>
    <xsl:copy-of select="$berlin"/>
    <xsl:copy-of select="$berlin/bla/city/name"/>
  </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/var-1.xsl]

Even more powerful

Anything inside the *contents* is bound to the variable – this allows even to generate complex structures by template applications:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="mondial">
    <xsl:variable name="bigcities">
      <xsl:apply-templates select="//city"/>
    </xsl:variable>
    <xsl:copy-of select="$bigcities"/>
    <xsl:copy-of select="$bigcities//name"/>
  </xsl:template>

  <xsl:template match="city">
    <xsl:if test='number(population)>1000000'>
    <xsl:copy-of select="current()"/>
   </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/var-2.xsl]

External Parameters

Stylesheets can be called with external parameters (e.g., from the shell, or from a Java environment):

- define formal parameters for the stylesheet:

  <xsl:stylesheet ...>
  <xsl:parameter name="*name*$_1$"/>
  <xsl:parameter name="*name*$_2$"/>
  stylesheet contents
  (parameters used as $name$_i$*)*
  </xsl:stylesheet ...>

- call e.g. (with saxon)
  saxonXSL -s bla.xml bla.xsl *name*$_1$=*value*$_1$ *name*$_2$=*value*$_2$

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 <xsl:param name="country"/>
 <xsl:template match="mondial">
  <xsl:copy-of select="//country[name=$country]"/>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/external-param.xsl]

8.2.6 XSLT: Miscellaneous

SORTING

For the set-based XSLT elements

- xsl:apply-templates and
- xsl:for-each

it can be specified whether the elements should be processed in the order of some key:

```
<xsl:sort select="xpath-expr"
    data-type = {"text"|"number"}
    order = {"descending"|"ascending"}/>
```

- "select" specifies the values according to which the nodes should be ordered (evaluated wrt. the node as context node)
- "data type" specifies whether the ordering should be alphanumeric or numeric,
- "order" specifies whether the ordering should be ascending or descending,
- if an "xsl:apply-templates"- or "xsl:for-each" element has multiple "xsl:sort" subelements, these are applied in a nested way (as in SQL).

Grouping (since XSLT 2.0)

Extends the <xsl:for-each> concept to groups:

<xsl:for-each-group select="*xpath-expr*" group-by="*local-key*">
content
</xsl:for-each-group>

Inside the content part:

- current element is the *first* element of the current group
  ⇒ for accessing/returning the whole group, something else must be used:
- current-group() returns the sequence of all elements of the current group (e.g., current-group()/name for all their names); can e.g. be used for aggregation
- current-grouping-key() returns the current value of the grouping key
- position() returns the number of the current group

Grouping (Example)

Example

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="mondial">
  <xsl:for-each-group select="country" group-by="encompassed/@continent">
   <continent nr="{position()}">
    <xsl:copy-of select="id(current-grouping-key())/name"/>
    <xsl:copy-of select="current-group()/name"/>
   </continent>
  </xsl:for-each-group>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/for-each-group.xsl]

Exercise 8.5
Do the same in XQuery (note: use "let" for the group). □

Using Functions from XQuery Functions and Operators

- the functions and operators from "XQuery Functions and Operators" (e.g., aggregations) are also available in XSLT.
- Namespace: http://www.w3.org/2005/xpath-functions

Example

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fn="http://www.w3.org/2005/xpath-functions" version="1.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="mondial">
  <xsl:for-each select="country">
   <country name="{name/text()}">
    <xsl:copy-of select="fn:sum(.//city/population[1])"/>
   </country>
  </xsl:for-each>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/fn-sum.xsl]

User-Defined Functions (since XSLT 2.0)

<xsl:function name="*local-ns*:*fname*">
    <xsl:param name="*param*$_1$"/>
        :
    <xsl:param name="*param*$_n$"/>
    contents
</xsl:function>

- the *local-ns* must be declared by xmlns:*local-ns*=*uri* in the xsl:stylesheet element; also xmlns:xs for XML Schema must be declared there
- function can then be used with n parameters in xsl:value-of, or in any XPath expression. e.g.,
  <xsl:value-of select="*local-ns*:*fname*(*value*$_1$,...,*value*$_n$)"/>

Import Mechanisms

XSL-stylesheets can import other stylesheets (i.e., they import their rules):

- <xsl:include href="*url*"/>
  for conflict-free stylesheets,
- <xsl:import href="*url*"/>
  definitions of the importing document have higher priority than definitions from the imported documents,
  the xsl:import subelements must precede all other subelements.

Example: DBIS Web pages

- general macros, frames etc. as templates in separate files
- individual page content in XML
- stylesheets generate Web pages from XML content file

Access to Data from Multiple Documents

- using the document()-function from XPath:

```
<xsl:template match="*">
 <html>
   <xsl:copy-of select="document('file:/home/may/paper.html')//h5"/>
 </html>
</xsl:template>
```

Examples: the XSLT stylesheets that have been used for XPath queries

Generation of Multiple Instances

- controlling output to different files (since XSLT 2.0):
  <xsl:result-document href="*output-uri*">

```
<?xml version="1.0" encoding="utf-8"?>
<!-- xsl:document is saxon/xslt1.1 for xslt2.0's xsl:result-document -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="*">
  <xsl:result-document href="file:/tmp/countries.xml">
    <countries><xsl:apply-templates select="country"/></countries>
  </xsl:result-document>
  <xsl:result-document href="file:/tmp/organizations.xml">
    <organizations><xsl:apply-templates select="organization"/></organizations>
  </xsl:result-document>
 </xsl:template>
 <xsl:template match="country"><xsl:copy-of select="name"/></xsl:template>
 <xsl:template match="organization"><xsl:copy-of select="name"/></xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/redirected-output.xsl]

8.3 XSL-FO

XSL-FO specifies *formatting objects*, that are added to a result tree and describe the later formatting

- page layout, areas, frames, indentation,
- colors, fonts, sizes,
- structuring, e.g. lists, tables ...

XSL-FO provides similar concepts as known from LaTeX.

FO-objects are e.g. (Namespace fo:) fo:block, fo:character, display-graphic, float, footnote, inline-graphic, list-block, list-item, list-item-body, list-item-label, multi-case, page-number, page-number-citation, region-before/after, region-body, simple-link, table, table-and-caption, table-body, table-caption, table-cell, table-column, table-footer, table-header, table-row.

- Each of these objects has appropriate attributes.

XSL-FO

- result tree contains *formatting objects* elements
- the result tree is then input to a formatter that generates HTML/LaTeX/RTF/PDF etc.
- currently only understood by
  - FOP (originally by James Tauber, now by Apache), a Java program that translates XML documents that include XSL-FO-markup to PDF:
    `http://xml.apache.org/fop/`
  - Adobe Document Server (XML → PDF)

8.4 XSLT: Language Design in the XML-World

- XSLT is itself in XML-Syntax
- there is a DTD for XSLT: `http://www.w3.org/TR/xslt#dtd`

⇒ Analogously, there is an XML syntax for XQuery: XQueryX, `http://www.w3.org/TR/xqueryx`, W3C Recommendation since 23 January 2007.

- XSLT uses an own *namespace*, xsl:....
- there are several further languages of this kind (programming languages, markup languages, representation languages ...):
  XLink, XML Schema,
  SOAP (Simple Object Access Protocol)
  WSDL (Web Services Description Language)
  OWL (Web Ontology Language)
  DocBook
  ... lots of application-specific languages.

8.5 Concepts

(cf. Slide 10)

- XML as an abstract *data model* (Infoset) with an *abstract datatype* (DOM) and several implementations (*physical level*),
- High-level *declarative*, *set-oriented* query language *logical level*: XPath/XQuery
- new: XSLT: transformational language
- two possibilities to define *views*:
  - XQuery: views as queries,
  - XSLT: views by transformations, especially XHTML views to the user.