3.1 Additional Features

- XPath support for
  - arithmetics
  - processing ID/IDREF cross-references
  - manipulation of strings
- Generating text
  - for content
  - for attribute values
- Repetition, sorting and conditional processing
- Generating numbers

XPath: Arithmetical Operations

- Operators for double-precision (64 bit) floating-point numbers
  `+, -, *, div, mod` (same as `%` in Java)
- Rounding numbers up, down, and to the closest integer:
  `floor(x), ceiling(x),round(x)`
- Formatting numbers as strings (e.g.):
  - » `format-number(-1.2534, "0.0") = "-1.3"`
  - XSLT function; applies Java decimal format patterns

Aggregate Functions

- Counting nodes
  - » `count(`node-set`)`
  - and summing them as numbers
    - » `sum(`node-set`)`
- Example:
  - Average of course grades below current node:

```
sum(.//course/@grade) div
count(.//course)
```


Cross-referencing

- Function `id` selects elements by their unique ID
  - **NB**: ID attributes must be declared in DTD (See an example later)
- Examples:
  - `id('sect:intro')` selects the element with unique ID `"sect:intro"`
  - `id('sect:intro')/para[5]` selects the fifth `para` child of the above element
  - `id('sect1 sect2 sect3')` selects 3 sections (with corresponding ID values)

String manipulation

- Equality and inequality of strings can be tested with operators = and `!=`
  - `"foo" = 'foo';` (NB alternative quotes)
  - `"foo" != "Foo"`
- Testing for substrings:
  - `starts-with("dogbert", "dog") = true()`
  - `contains("dogbert", "gbe") = true()`
- Concatenation (of two or more strings),
  - `concat("dog", "bert") = "dogbert"`

XPath: more string functions

- `substring-before("ftp://a", "//") = substring-before("ftp://a", "/") = "ftp:"`
- `substring-after("ftp://a", "/")= "/a"`
- `substring(`*string, start, length?*`)`:
  - » `substring("dogbert", 1, 3) = "dog"`
  - » `substring("dogbert", 3) = "gbert"`
- `string-length("dogbert")=7`
- `translate(`*Str, Replaced, Replacing*`)`:
  - » `translate("doggy","dgo","Ssi")="Sissy"`

Generating Text

- The string-value of an expression can be inserted in the result tree by instruction
  `<xsl:value-of select="`*Expr*`" />`
  - if *Expr* evaluates to a node-set, **value of the first node** in document order is used
- Consider transforming source elements like

```
<name alias="Bird">
 <first>Charlie</first><last>Parker</last>
</name>
```

to the form

```
Charlie ("Bird") Parker
```


Computing generated text (2)

- This can be specified by template rule

```
<xsl:template match="name">
 <xsl:value-of select="first" />
 ("<xsl:value-of select="@alias" />")
 <xsl:value-of select="last" />
 <xsl:text>
 </xsl:text>
</xsl:template>
```

- Verbatim text (like the white-space above) can be inserted using `xsl:text`

Attribute value templates

- The string-value of an expression can be inserted in an attribute value by surrounding the expression by braces { and }
- Consider transforming source element

```
<photo>
  <file>Mary.jpg</file>
  <size width="300"/>
</photo>
```

into form

```
<img src="/images/Mary.jpg"
     width="300" />
```


Attribute value templates (2)

- This can be specified by template rule

```
<xsl:template match="photo">
 <img src="/images/{file}"
      width="{size/@width}" />
</xsl:template>
```

- Expressions `{file}` and `{size/@width}` are evaluated in the context of the current node (the `photo` element)

XSLT: Repetition

- Nodes can be "pulled" from source for processing using

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

  - *Template* is applied to each of the selected nodes (0, 1 or more), each node in turn as the `current()` node
    - » in document order, unless sorted using `xsl:sort` instructions (see later)

Example (of `xsl:for-each`)

- Consider formatting the below document as HTML:

```
<!DOCTYPE document [ <!ATTLIST section id ID #IMPLIED> ]>
<document> <title>The Joy of XML</title>
<section id="Intro"><title>Getting Started</title>
 <name><first>Helen</first> <last>Brown</last></name>
 says that processing XML documents is fun.
 <name><first>Dave</first> <last>Dobrik</last></name> agrees.
</section>
<section><title>Family affairs</title>
 <name><first>Bob</first> <last>Brown</last></name> is the
 husband of  <name><first>Helen</first>
 <last>Brown</last></name>.  </section>
<section><title>Finishing Up</title>
As we discussed in <title-ref idref="Intro" />, processing XML
documents is fun. </section></document>
```


Example: Table of contents

- A table of contents can be formed of section titles:

```
<xsl:template match="/">
<HTML><HEAD> <TITLE><xsl:value-of
        select="document/title"/></TITLE></HEAD>
 <BODY>
  <H2>Table of Contents</H2>
  <OL> <!-- Pull each section title: -->
     <xsl:for-each select="//section/title">
       <LI><xsl:apply-templates /></LI>
     </xsl:for-each>
  </OL> <!-- then process the sections: -->
  <xsl:apply-templates select="document/section"/>
 </BODY> </HTML>
</xsl:template>
```


Example (cont; Cross references)

- Cross references (to sections) can also be processed using `xsl:for-each`:

```
<xsl:template match="title-ref">
  <xsl:for-each select="id(@idref)">
    Section  (<xsl:value-of
    select="substring(title, 1, 8)" />...)
  </xsl:for-each>
</xsl:template>
```

- With this rule the source fragment

```
As we discussed in <title-ref idref="Intro"/>
```

  becomes

  As we discussed in Section (Getting …)

XSLT Sorting

- A sorted order for the processing of nodes with `xsl:for-each` and `xls:apply-templates` can be specified by <**`xsl:sort`**/>
- controlled by attributes of `xsl:sort` like
  - `select`: expression for the sort key (default: ".")
  - `data-type`: `"text"` (default) or `"number"`
  - `order`: `"ascending"` (default) or `"descending"`
- The first `xsl:sort` specifies the primary sort key, the second one the secondary sort key, and so on.

Example (cont; Sorted index of names)

- All names can be collected in a last-name-first-name order using the below template

```
<H2>Index</H2> <UL>
  <xsl:for-each select="//name">
    <xsl:sort select="last" />
    <xsl:sort select="first" />
    <LI><xsl:value-of select="last"
        />, <xsl:value-of select="first"/></LI>
  </xsl:for-each>
</UL>
```

- This creates an UL list with items

```
<LI>Brown, Bob</LI>
<LI>Brown, Helen</LI>
<LI>Brown, Helen</LI>
<LI>Dobrik, Dave</LI>
```

Possible to eliminate duplicates? Yes, but a bit tricky. See next

Conditional processing

- A template can be instantiated or ignored based on the value of a `test` Boolean expression, using

```
<xsl:if test="Expression">
      Template
</xsl:if>
```

- Example: a comma-separated list of names:

```
<xsl:template match="namelist/name">
  <xsl:apply-templates/>
  <xsl:if test="position() &lt; last()"
   >, </xsl:if>
</xsl:template>
```


An aside: Meaning of `position()`

- Evaluation wrt the current node list. Above rule applied to a source with

```
<namelist><name>a</name><name>b</name></namelist>
<namelist><name>c</name><name>d</name></namelist>
```

by invocation

```
<xsl:apply-templates select="//name" />
```

yields "a,b,c,d" (a single node list);

With invocation

```
<xsl:template match="namelist">
      <xsl:apply-templates select="name" />
```

it yields "a,b" and "c,d" (Clever, and tricky!)

Conditional processing (2)

- Also a case-like construct (~ `switch` in Java):

```
<xsl:choose>
  <!-- The first 'when' whose test=true() is
  instantiated: -->
  <xsl:when test="Expr1"> … </xsl:when>
  <xsl:when test="Expr2"> … </xsl:when>
     …
  <!-- If no 'when' applies, an optional
  'otherwise' is instantiated: -->
  <xsl:otherwise> … </xsl:otherwise>
</xsl:choose>
```


Example (cont; Eliminating duplicate names)

- No access to other nodes (except `current()`) in the current node list
  - But can refer to other nodes in the source tree
  - Process just the first one of duplicate `name`s:

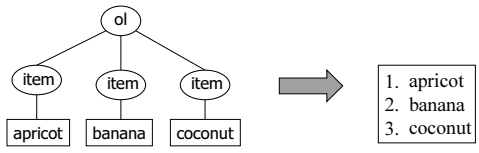
```
<xsl:for-each select="//name">
  <xsl:sort select="last"/>
  <xsl:sort select="first" />
  <xsl:if test="not(
        preceding::name[first=current()/first
                  and last=current()/last] )">
     <LI><xsl:value-of select="last"
       />, <xsl:value-of select="first"/></LI>
  </xsl:if>
  </xsl:for-each>
```


Generating Numbers

- Formatted numbers can be inserted in the result tree by element `<xsl:number />`
  - by the position of the current node in the source tree
  - nodes to be counted specified by a `count` pattern
  - common numbering schemes supported: single-level, hierarchical, and sequential ignoring levels
- Typical cases in following examples
  - » (Complete specification rather complex)
- Example 1: Numbering list items

Generating numbers: Example 1

```
<xsl:template match="ol/item">
  <!-- default: count similar siblings (items) -->
  <xsl:number format="1. "/>
  <xsl:apply-templates/>
<xsl:template>
```

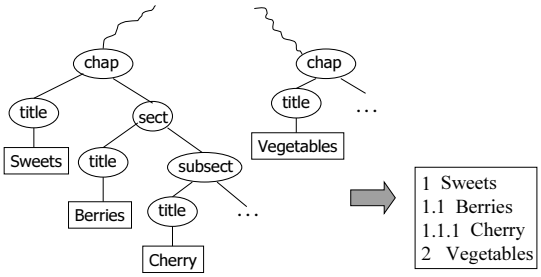


Generating numbers: Example 2

- Hierarchical numbering (1, 1.1, 1.1.1, 1.1.2, …) for titles of chapters, titles of their sections, and titles of subsections:

```
<xsl:template match="title">
  <xsl:number level="multiple"
     count="chap|sect|subsect"
     format="1.1 "/>
  <xsl:apply-templates/>
</xsl:template>
```


Generating numbers: Example 2

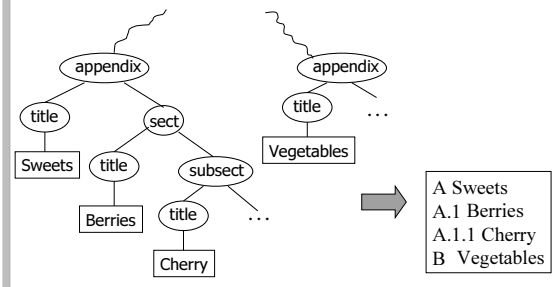


Example 2: Variation

- As above, but number titles within appendices with A, A.1, A.1.1, B.1 etc:

```
<xsl:template match="appendix//title">
  <xsl:number level="multiple"
     count="appendix|sect|subsect"
     format="A.1 "/>
  <xsl:apply-templates/>
</xsl:template>
```


Example 2: Variation

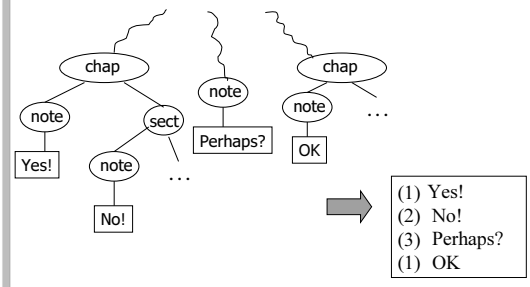


Generating numbers: Example 3

- Sequential numbering of `notes` within `chap`ters: (more precisely: starting anew at the start of any chapter)

```
<xsl:template match="note">
  <xsl:number level="any"
              from="chap"
              format="(1) "/>
  <xsl:apply-templates/>
</xsl:template>
```


Ex 3: Sequential numbering `from` chaps



3.2 Computing with XSLT

- XSLT is a declarative rule-based language
  - for a special purpose: XML transformations
  - Could we use XSLT for procedural computing?
  - What is the exact computational power of XSLT?
- We've seen some programming-like features:
  - iteration over source nodes (**`xsl:for-each`**)
  - conditional evaluation (**`xsl:if`** and **`xsl:choose`**)

Computing with XSLT

- Further programming-like features:
  - **variables** (names bound to *non-updatable* values):

```
<xsl:for-each select="//name">
 <xsl:variable name="LAndF"
       select="concat(last, ', ', first)" />
  ...
```

  - callable **named templates** with **parameters**:

```
 <xsl:call-template name="process-name">
  <xsl:with-param name="pname" select="$LAndF" />
 </xsl:call-template>
  ...
</xsl:for-each>
```


Visibility of Variable Bindings

- The binding is **visible** in following siblings of **`xsl:variable`**, and in their descendants:

```
<xsl:for-each select="//name">
  <xsl:variable name="LAndF"
          select="concat(last, ', ', first)" />
  ...
  <xsl:call-template name="process-name">
     <xsl:with-param name="pname" select="$LAndF" />
  </xsl:call-template>

  ...

</xsl:for-each>
<TABLE> . . . </TABLE>
```


A Real-Life Example

- We used LaTeX to format an XML article. For this, we needed to map source table structures

```
<tgroup cols="3">
...
</tgroup>
```

  to corresponding LaTeX environments:

```
\begin{tabular}{lll} %3 left-justified cols
 ...
\end{tabular}
```

- How to do this?

Possible solution (for up to 4 columns)

```
<xsl:template match="tgroup">
  \begin{tabular}{l<xsl:if test="@cols > 1">l</xsl:if
             ><xsl:if test="@cols > 2">l</xsl:if
             ><xsl:if test="@cols > 3">l</xsl:if>}
      <xsl:apply-templates />
  \end{tabular}
</xsl:template>
```

- OK, but inelegant
- How to accept any number of columns?

More General Solution (1/2)

- Pass the column-count to a named template which generates the requested number of '`l`'s:

```
<xsl:template match="tgroup">
\begin{tabular}{ <xsl:call-template name="gen-cols">
       <xsl:with-param name="count" select="@cols" />
       <xsl:with-param name="symb" select="'l'" />
      </xsl:call-template> }
    <xsl:apply-templates />
\end{tabular}
</xsl:template>
```


Solution 2/2: Recursive `gen-cols`

```
<xsl:template name="gen-cols">
  <xsl:param name="count" />
  <xsl:param name="symb" />
  <xsl:if test="$count > 0">
      <xsl:value-of select="$symb" />
      <xsl:call-template name="gen-cols">
        <xsl:with-param name="count"
            select="$count - 1" />
        <xsl:with-param name="symbol"
            select="$symbol" />
      </xsl:call-template>
  </xsl:if>
</xsl:template>
```

} formal parameters

Stylesheet Parameters

- Stylesheets can get parameters from command line, or through JAXP `Transformer.setParameter()`:

```
<xsl:transform ... >
  <xsl:output method="text" />
  <xsl:param name="In" select="0"/> <!-- default -->
  <xsl:template match="/">
      <xsl:value-of select="2*$In"/> </xsl:template>
</xsl:transform>

$ java -jar saxon.jar dummy.xml double.xslt In=120
240
```


Computational power of XSLT

- XSLT seems quite powerful, but how powerful is it?
  - Implementations provide extension mechanisms, e.g., to call arbitrary Java methods
  - Are there limits to XSLT processing that we can do *without* extensions?
- **Any** algorithm can be shown computable with plain XSLT
  - by simulating Turing machines by a recursive named template with string parameters

What does this mean?

- XSLT has **full algorithmic power**
  - (It is "Turing-complete")
  - Is this intentional?
    - » Inconvenient as a general-purpose programming language!
  - Impossible to recognise non-terminating transformations automatically
    (<– the "halting problem" has no algorithmic solution)
    - » could attempt "denial-of-service" attacks with non-terminating style sheets