CS342: Organization of Prog. Languages

Topic 14: [Language] XML and XSLT

- Extensible Markup Language – XML
- XML Namespaces
- XML Transformations
- A First Stylesheet
- Copying Elements Through
- The XT shell script
- Templates
- Patterns
- XPath Expressions
- Grammar for Location Paths

- Abbreviated Syntax for Path Location
- Constructing Output
- Conditional XSLT operations: mode=, choose, and if
- Parameters in XSLT
- Generating Detailed Text in Output
- Computing Attributes
- Computing Text Content
- Computing Tags
- Recursive application of patterns
- Functions with parameters
- Processing all children with Tail recursion

Extensible Markup Language – XML

- Defined by the World Wide Web Consortium.
  Current set of W3C Recommendations:
- `http://www.w3.org/TR/1998/REC-xml-19980210`
  XML 1.0
- `http://www.w3.org/TR/1999/REC-xml-names-19990114`
  Namespaces in XML

- References for XSLT
  `http://www.w3.org/TR/1999/REC-xslt-19991116`
  XSL Transformations (XSLT) Version 1.0
- `http://www.w3.org/TR/1999/REC-xpath-19991116`
  XML Path Language (XPath) Version 1.0

- See also
  `http://www.w3.org/XML/1999/XML-in-10-points`
  "XML in 10 Points"
- `http://www.w3.org/TR/2000/CR-xlink-20000703`
  XML Linking Language (XLink) Version 1.0

What does XML look like?

- A lot like HTML..... (both come from SGML).

```
<semantics>
   <ci><mo>rank</mo></ci>
   <annotation-xml encoding="OpenMath">
      <OMS cd="linalg1" name="rank"/>
   </annotation-xml>
</semantics>
```

- Differences:
  - Case-sensitive.
  - Every opening <X> must have a closing </X>.
    I.e., cannot be closed with </>.
  - Empty <X> </X> can be abbreviated <X/>.
  - Attributes must be quoted, i.e. <X foo="ok">.
  - Unicode.

What is XML used for?

- Marking up data of all sorts.
  Data bases.
  Documents.
  Network protocols.
  ...
- Revision of HTML as an XML application: XHTML.

Basic Vocabulary

- A piece of XML between an opening and closing pair is a *"document fragment"*.
- A document fragment is "well-formed," if all openning and closing pairs inside are properly nested, and other basic syntax is right.
- A document fragment is "valid," if additionally it satisfies a specific grammar, given by a "document type description" (DTD).
- The thing in angle brackets is called a "tag," e.g. `<ul>`.
- A document fragment inside a particular tag is called an "element," e.g.
  `<A> foo <B> blah </B> </A>`.
- An "entity" is a shorthand for some character or element, and uses the sytnax `&Name;`.

XML Namespaces

- When data for several XML applications are used at once, namespaces avoid collision in tags and attributes.
- Introduced via an `xmlns` attribute.
- An attribute of the form

  `xmlns:oj="http://orangina.com/Drink"`

  defines a prefix `oj` which can be used as prefix to a name, e.g. `oj:xxx`, within the attributed element.
- E.g.

```
<aisle xmlns:oj="http://orangina.com/Drink">
   <shelf>
     <oj:big-bottle/>
     <oj:small-bottle-case count="12"/>
   </shelf>
</aisle>
```

- An attribute of the form

  `xmlns="http://www.toyota.com/cars"`

  changes the default name space.
- E.g.

```
<mathml:annotation-xml encoding="OpenMath">
  <OMA xlink:href="id('E')"
       xmlns="http://www.openmath.org/OpenMath">
    <OMS cd="logic1" name="and" xlink:href="id('E')"/>
    <OMA xlink:href="id('E.1')">
      <OMS cd="logic1" name="xor" xlink:href="id('E.1.3')"/>
      <OMV name="a" xlink:href="id('E.1.2')"/>
      <OMV name="b" xlink:href="id('E.1.4')"/>
    </OMA>
    <OMA xlink:href="id('E.3')">
      <OMS cd="logic1" name="xor" xlink:href="id('E.3.3')"/>
      <OMV name="c" xlink:href="id('E.3.2')"/>
      <OMV name="d" xlink:href="id('E.3.4')"/>
    </OMA>
  </OMA>
</mathml:annotation-xml>
```

  Everything which is does not have a prefix is in the OpenMath namespace.

XML Transformations

- XSLT is a pattern-matching language for transforming XML documents.

- Example:

```
<xsl:template match="m:interval">
 <m:mrow>
  <xsl:choose>
   <xsl:when test="@closure='closed'">
    <m:mfenced open="[" close="]" separators=","><xsl:apply-templates/></m:mfenced>
   </xsl:when>
   <xsl:when test="@closure='open'">
    <m:mfenced open="]" close="[" separators=","><xsl:apply-templates/></m:mfenced>
   </xsl:when>
   ...
   <xsl:otherwise>
    <m:mfenced open="[" close="]" separators=","><xsl:apply-templates/></m:mfenced>
   </xsl:otherwise>
  </xsl:choose>
 </mrow>
</xsl:template>
```

A First Stylesheet

- Convert `h1` elements to `TaDa` elements, ignoring other tags.
- Sytlesheet in file `"s1.xsl"`:

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

<xsl:template match="h1">
  <TaDa> <xsl:apply-templates/> </TaDa>
</xsl:template>

</xsl:stylesheet>
```

- Input data file `"t1.xml"`:

```
<body>
  <h1>Now is the time</h1>
  <p tone="boring">This is a paragraph</p>
  <h1>Now is another time</h1>
  <ul>
    <li>Blah blah blah</li>
    <li>Blah blah blah</li>
  </ul>
</body>
```

- Unix command line:

```
xt t1.xml  s1.xsl > t1.out
```

- Output file "t1.out":

```
<?xml version="1.0" encoding="utf-8"?>

  <TaDa>Now is the time</TaDa>
  This is a paragraph

  <TaDa>Now is another time</TaDa>

    Blah blah blah
    Blah blah blah
```

- Note: all the other tags were stripped.

Copying Elements Through

- Suppose we want to transform some elements, but copy the rest through unmodified.
  Then we could add one rule and use the following stylesheet:

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

<!-- This stops the output from being 1 giant line -->
<xsl:output method="xml" indent="yes"/>

<!-- Previous rule for handling h1 elements -->
<xsl:template match="h1">
  <TaDa> <xsl:apply-templates/> </TaDa>
</xsl:template>

<!-- Rule to copy stuff through. -->
<xsl:template match="/|*|@*">
  <xsl:copy> <xsl:apply-templates select="*|@*|text()"/> </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

- This gives:

```
<?xml version="1.0" encoding="utf-8"?>
<body>
  <TaDa>Now is the time</TaDa>
  <p tone="boring">This is a paragraph</p>

  <TaDa>Now is another time</TaDa>
  <ul>
    <li>Blah blah blah</li>
    <li>Blah blah blah</li>
  </ul>
</body>
```

The XT shell script

- For the previous example, we have been using the "XT" implementation of XSLT.
- This is a freely available Java program, from `http://www.jclark.com`
- The command line we have used calls a shell script like this:

```
#!/bin/sh
# Get xt.zip and xp.zip from www.jclark.com.
# Unzip xt.zip and xp.zip into xtdir.
# Call as: xt in-file stylesheet-file

xtdir=/usr/local/Packages/xt
jars="$xtdir"/xt.jar:"$xtdir"/sax.jar:"$xtdir"/xp.jar
sax=com.jclark.xsl.sax

export CLASSPATH="$jars":$CLASSPATH

java -D$sax.parser=com.jclark.xml.sax.CommentDriver $sax.Driver $@
```

Templates

- The basic element to specify a rule is `xsl:template`.
- Templates provide rewrite rules for XML trees.
- Which templates are applied is determined by the `match="..."` attribute on `xsl:template`.

Patterns

- What goes in the `match="..."` are phrases in a pattern matching language.
- The pattern `"/"` matches the root.
- The pattern "*word*" matches a <*word*> entity.
- ... more later.

- The pattern-matching language is an extension to XPath and has two syntaxes: a long form and a short form.
- XSLT stylesheets can be somewhat confusing at first, since you need to understand at least part of three separate syntaxes: the XML syntax of XSLT, plus the long and short forms of the pattern matching language.

Example

```
<?xml version="1.0"?>

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

<xsl:template match="/">
   <html>
     <head>
       <title>Greeting</title>
     </head>
     <body>
       <p>Hello <xsl:value-of select="name"/></p>
     </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

XPath Expressions

- The principal construct in XPath is the expression.

  An expression is evaluated to an object which is one of the following: a boolean, a number, a string, or a node-set.

  A node-set is an unordered collection of nodes without duplicates.

- Expression evaluation occurs in a context which consists of: the context node, a context node list, a set of variable bindings, a function library, namespace declarations.

  The most important, for our purposes today are the context node and the variable bindings.

- The expression itself consists of "paths" matching sets of nodes and tests, function calls, and computations.

Tons of Path Examples

These are some examples of location paths using the unabbreviated syntax, (taken from the XPath specification):

- `child::para` selects the para element children of the context node
- `child::*` selects all element children of the context node
- `child::text()` selects all text node children of the context node
- `child::node()` selects all the children of the context node, whatever their node type
- `attribute::name` selects the name attribute of the context node
- `attribute::*` selects all the attributes of the context node
- `descendant::para` selects the para element descendants of the context node
- `ancestor::div` selects all div ancestors of the context node

- `ancestor-or-self::div` selects the div ancestors of the context node and, if the context node is a div element, the context node as well
- `descendant-or-self::para` selects the para element descendants of the context node and, if the context node is a para element, the context node as well
- `self::para` selects the context node if it is a para element, and otherwise selects nothing
- `child::chapter/descendant::para` selects the para element descendants of the chapter element children of the context node
- `child::*/child::para` selects all para grandchildren of the context node
- `/` selects the document root (which is always the parent of the document element)
- `/descendant::para` selects all the para elements in the same document as the context node

- `/descendant::olist/child::item` selects all the item elements that have an olist parent and that are in the same document as the context node
- `child::para[position()=1]` selects the first para child of the context node
- `child::para[position()=last()]` selects the last para child of the context node
- `child::para[position()=last()-1]` selects the last but one para child of the context node
- `child::para[position()>1]` selects all the para children of the context node other than the first para child of the context node
- `following-sibling::chapter[position()=1]` selects the next chapter sibling of the context node
- `preceding-sibling::chapter[position()=1]` selects the previous chapter sibling of the context node

- `/descendant::figure[position()=42]` selects the forty-second figure element in the document
- `/child::doc/child::chapter[position()=5]/child::section[position()=2]` selects the second section of the fifth chapter of the doc document element
- `child::para[attribute::type="warning"]` selects all para children of the context node that have a type attribute with value warning
- `child::para[attribute::type='warning'][position()=5]` selects the fifth para child of the context node that has a type attribute with value warning
- `child::para[position()=5][attribute::type="warning"]` selects the fifth para child of the context node if that child has a type attribute with value warning
- `child::chapter[child::title='Introduction']` selects the chapter children of the context node that have one or more title children with string-value equal to Introduction

- `child::chapter[child::title]` selects the chapter children of the context node that have one or more title children
- `child::*[self::chapter or self::appendix]` selects the chapter and appendix children of the context node
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last chapter or appendix child of the context node

Grammar for Location Paths

From the XPath spec:

```
LocationPath ::=
       RelativeLocationPath | AbsoluteLocationPath

AbsoluteLocationPath ::=
       '/' RelativeLocationPath?  | AbbreviatedAbsoluteLocationPath

RelativeLocationPath ::=
       Step | RelativeLocationPath '/' Step | AbbreviatedRelativeLocationPath

Step ::=
       AxisSpecifier NodeTest Predicate* | AbbreviatedStep

AxisSpecifier ::=
       AxisName '::' | AbbreviatedAxisSpecifier

AxisName ::=
      'ancestor'   | 'ancestor-or-self'   | 'attribute'  | 'child'
    | 'descendant' | 'descendant-or-self' | 'following'  | 'following-sibling'
    | 'namespace'  | 'parent'             | 'preceding'  | 'preceding-sibling'
    | 'self'
```

Abbreviated Syntax for Path Location

The pattern matching language has the concept of several different "axes" which define collections of nodes to be searched, e.g. parents, children, following siblings, etc.

Here are some examples of location paths using abbreviated syntax, from the XPath spec. Basically, these give short-hands for the most important axes.

- `para` selects the para element children of the context node
- `*` selects all element children of the context node
- `text()` selects all text node children of the context node
- `@name` selects the name attribute of the context node
- `@*` selects all the attributes of the context node
- `para[1]` selects the first para child of the context node

- `para[last()]` selects the last para child of the context node
- `*/para` selects all para grandchildren of the context node
- `/doc/chapter[5]/section[2]` selects the second section of the fifth chapter of the doc
- `chapter//para` selects the para element descendants of the chapter element children of the context node
- `//para` selects all the para descendants of the document root and thus selects all para elements in the same document as the context node
- `//olist/item` selects all the item elements in the same document as the context node that have an olist parent
- `.` selects the context node
- `.//para` selects the para element descendants of the context node
- `..` selects the parent of the context node
- `../@lang` selects the lang attribute of the parent of the context node

- `para[@type="warning"]` selects all para children of the context node that have a type attribute with value warning
- `para[@type="warning"][5]` selects the fifth para child of the context node that has a type attribute with value warning
- `para[5][@type="warning"]` selects the fifth para child of the context node if that child has a type attribute with value warning
- `chapter[title="Introduction"]` selects the chapter children of the context node that have one or more title children with string-value equal to Introduction
- `chapter[title]` selects the chapter children of the context node that have one or more title children
- `employee[@secretary and @assistant]` selects all the employee children of the context node that have both a secretary attribute and an assistant attribute

Earlier example revisited

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

<!-- This stops the output from being 1 giant line -->
<xsl:output method="xml" indent="yes"/>

<!-- Previous rule for handling h1 elements -->
<xsl:template match="h1">
  <TaDa>
    <xsl:apply-templates/>
  </TaDa>
</xsl:template>

<!-- Rule to copy stuff through. -->
<xsl:template match="/|*|@*">
  <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Constructing Output

- Output can be given as explicit XML or text included in the body of a template. E.g.

```
<xsl:template match="slide">
   <h1> Topic: </h1>
</xsl:template>
```

- Output can be text selected from the input, using `value-of`

```
<xsl:template match="...">
   <p>Hello <xsl:value-of select="name"/></p>
</xsl:template>
```

- Output can be an XML tree selected from the input

```
<xsl:template match="...">
   <xsl:copy-of select="..."/>
</xsl:template>
```

- Output can be a re-written XML tree selected from the input

```
<xsl:template match="...">
   <xsl:copy> <xsl:apply-templates select="..."/> </xsl:copy>
</xsl:template>
```

Conditional XSLT operations: mode=, choose, and if

The following stylesheet will regoranize the children of a `thing-list`.

```
<?xml version="1.0"?>

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

<xsl:output method="xml" indent="yes"/>

<!-- Sort thing-list into animal/vegtable/mineral -->

<xsl:template match="thing-list">
  <sorted-thing-list>
    <animal>   <xsl:apply-templates mode="keep-animals"/>   </animal>
    <vegtable> <xsl:apply-templates mode="keep-vegtables"/> </vegtable>
    <mineral>  <xsl:apply-templates mode="keep-minerals"/>  </mineral>
  </sorted-thing-list>
</xsl:template>
```

```
<xsl:template match="*" mode="keep-animals">
  <xsl:choose>
     <xsl:when test="self::dog">
        <xsl:copy> <xsl:apply-templates/> </xsl:copy>
     </xsl:when>
     <xsl:when test="self::wolf">
        <xsl:copy> <xsl:apply-templates/> </xsl:copy>
     </xsl:when>
     <xsl:when test="@kind='animal'">
        <xsl:copy> <xsl:apply-templates/> </xsl:copy>
     </xsl:when>
  </xsl:choose>
</xsl:template>

<xsl:template match="*" mode="keep-vegtables">
   <xsl:if test="self::beet or self::tree or @kind='vegtable'">
      <xsl:copy> <xsl:apply-templates/> </xsl:copy>
   </xsl:if>
</xsl:template>

<xsl:template match="*" mode="keep-minerals">
   <xsl:if test="self::pebble or self::gold or @kind='mineral'">
      <xsl:copy> <xsl:apply-templates/> </xsl:copy>
   </xsl:if>
</xsl:template>
```

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

</xsl:stylesheet>
```

Sample input

```
<library>
  <thing-list>
    <dog>Lassie</dog>
    <crystal kind="mineral">Quartz</crystal>
    <tree>
       <root>
            <br>A</br>
            <br> <br>B</br> <br>C</br> </br>
       </root>
    </tree>
    <wolf>Akela</wolf>
    <pebble/>
  </thing-list>

  <thing-set>
    <a/> <b/> <c/>
  </thing-set>
</library>
```

Output (spacing edited to fit page)

```
<?xml version="1.0" encoding="utf-8"?>
<library>
  <sorted-thing-list>
    <animal>
      <dog>Lassie</dog>
      <wolf>Akela</wolf>
    </animal>
    <vegtable>
      <tree>
         <root>
            <br>A</br>
            <br> <br>B</br> <br>C</br> </br>
         </root>
      </tree>
    </vegtable>
    <mineral>
       <crystal>Quartz</crystal>
       <pebble/>
    </mineral>
  </sorted-thing-list>
  <thing-set>
    <a/> <b/> <c/>
  </thing-set>
</library>
```

Parameters in XSLT

- It is possible to introduce parameters using `xsl:param` elements.
- The element

  ```
  <xsl:param name="Var" select="9"/>
  ```

  introduces a new parameter, Var, and initializes it to 9.
- The element `xsl:with-param` is used in an `xsl::apply-templates` operation to pass a parameter, over-riding one of the template's own parameters.

Example: Convert a list of numbers to a list of pairs of numbers.

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

<xsl:template match="list">
 <xsl:param name="POS" select="0"/>
 <xsl:choose>
   <xsl:when test="$POS = 0">
     <list>
       <xsl:apply-templates select=".">
         <xsl:with-param name="POS" select="1"/>
       </xsl:apply-templates>
     </list>
   </xsl:when>
   <xsl:when test="number[$POS] and number[$POS+1]">
     <pair>
       <xsl:apply-templates select="number[$POS]"/>
       <xsl:apply-templates select="number[$POS + 1]"/>
     </pair>
     <xsl:apply-templates select=".">
       <xsl:with-param name="POS" select="$POS + 2"/>
     </xsl:apply-templates>
  </xsl:when>
 </xsl:choose>
</xsl:template>
```

```
<!-- Rule to copy stuff through. -->
<xsl:template match="/|*|@*">
 <xsl:copy> <xsl:apply-templates select="*|@*|text()"/> </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Sample input:

```
<list>
   <number>10</number>
   <number>11</number>
   <number>20</number>
   <number>21</number>
   <number>30</number>
   <number>31</number>
</list>
```

Output:

```
<?xml version="1.0" encoding="utf-8"?>
<list>
<pair>
<number>10</number>
<number>11</number>
</pair>
<pair>
<number>20</number>
<number>21</number>
</pair>
<pair>
<number>30</number>
<number>31</number>
</pair>
</list>
```

Generating Detailed Text in Output

XSLT can be used to convert XML to other text formats.

An `<xsl:text>` element can be used to give literal text to be included in the output.

We show a simple stylesheet to convert a subset of XHTML to TeX.

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

<xsl:template match="/">
  <xsl:text>\documentclass{article}
\begin{document}</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>\end{document}</xsl:text>
</xsl:template>

<xsl:template match="head"/>

<xsl:template match="h1">
  <xsl:text>\section{</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>}</xsl:text>
</xsl:template>

<xsl:template match="h2">
  <xsl:text>\subsection{</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>}</xsl:text>
</xsl:template>
```

```
<xsl:template match="strong">
  <xsl:text>{\bf </xsl:text>
  <xsl:apply-templates/>
  <xsl:text>}</xsl:text>
</xsl:template>

<xsl:template match="emph">
  <xsl:text>{\em </xsl:text>
  <xsl:apply-templates/>
  <xsl:text>}</xsl:text>
</xsl:template>

<xsl:template match="p">
  <xsl:text>\par </xsl:text><xsl:apply-templates/>
</xsl:template>

<xsl:template match="br">
  <xsl:text>~\newline</xsl:text>
</xsl:template>

<xsl:template match="ol">
  <xsl:text>\begin{enumerate}</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>\end{enumerate}</xsl:text>
</xsl:template>
```

```
<xsl:template match="ul">
  <xsl:text>\begin{itemize}</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>\end{itemize}</xsl:text>
</xsl:template>

<xsl:template match="li">
  <xsl:text>\item </xsl:text>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="pre">
  <xsl:text>\begin{verbatim}</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>\emd{verbatim}</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

This converts the XHTML file:

```
<html>
<head>
 <title>Programming Languages</title>
</head>
<body bgcolor="#FFF8F8" text="#000000">
 <h1>CS 342b: Organization of Programming Languages</h1>
 <p>This is a mini <strong>XHTML</strong> page.</p>
 <ul>
   <li>Why Study Programming Languages?</li>
   <li>Computer Language Paradigms</li>
 </ul>
 <h1>Who Are We?</h1>
 <pre>
   CS    Yr 2   Yr 3   Yr 4
   Eng   Yr 2   Yr 3   Yr 4
   Other
 </pre>
</body>
</html>
```

to give TeX:

```
\documentclass{article}
\begin{document}

 \section{CS 342b: Organization of Programming Languages}
 \par This is a mini {\bf XHTML} page.
 \begin{itemize}
   \item Why Study Programming Languages?
   \item Computer Language Paradigms
 \end{itemize}
 \section{Who Are We?}
 \begin{verbatim}
   CS    Yr 2   Yr 3   Yr 4
   Eng   Yr 2   Yr 3   Yr 4
   Other
 \emd{verbatim}

\end{document}
```

Computing Attributes

Computed values may be placed in the attributes of the result tree.

The presence of the characters “{” “}” causes a value is to be inserted in an attribute.

Within “{” “}” the usual expressions are used.

E.g. Change `<num>xxx</num>` to `<number val="xxx"/>`

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

<xsl:template match="num"> <number val="{text()}"/> </xsl:template>

<!-- Rule to copy stuff through -->
<xsl:template match="/|*|@*">
  <xsl:copy> <xsl:apply-templates select="*|@*|text()"/> </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

This converts the input:

```
<text>
  The number <num><base>10</base>123</num>
  is <num>100 + 23</num>
</text>
```

to the output:

```
<?xml version="1.0" encoding="utf-8"?>
<text>
  The number <number val="123"/>
  is <number val="100 + 23"/>
</text>
```

Computing Attributes II

The following style sheet tranforms the representation of complex numbers.

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

<xsl:template match="number">
  <complex realpart="{real}"
           imagpart="{imag}"/>
</xsl:template>

<xsl:template match="data">
  <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

The input:

```
<data>
  <number><real>1.0</real><imag>1.5</imag></number>
  <number><real>3.1</real><imag>2.2</imag></number>
  <number><real>4.8</real><imag>3.8</imag></number>
  <number><real>0.0</real><imag>9.0</imag></number>
</data>
```

gives the output:

```
<?xml version="1.0" encoding="utf-8"?>
<data>
  <complex realpart="1.0" imagpart="1.5"/>
  <complex realpart="3.1" imagpart="2.2"/>
  <complex realpart="4.8" imagpart="3.8"/>
  <complex realpart="0.0" imagpart="9.0"/>
</data>
```

Computing Text Content

We have seen how to make a computation on content, attribute and parameter values and place the result as an attribute value.

Similarly, we can compute new content with
`<xsl:value-of>`

Example:

```
<?xml version="1.0"?>

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

<xsl:template match="complex">
  <number>
     <real><xsl:value-of select="@realpart"/></real>
     <imag><xsl:value-of select="@imagpart"/></imag>
  </number>
</xsl:template>

<xsl:template match="data">
  <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Sample input:

```
<data>
  <complex  realpart="1.0" imagpart="1.5"/>
  <complex  realpart="3.1" imagpart="2.2"/>
  <complex  realpart="4.8" imagpart="3.8"/>
  <complex  realpart="0.0" imagpart="9.0"/>
</data>
```

Computed output:

```
<?xml version="1.0" encoding="utf-8"?>
<data>
  <number><real>1.0</real><imag>1.5</imag></number>
  <number><real>3.1</real><imag>2.2</imag></number>
  <number><real>4.8</real><imag>3.8</imag></number>
  <number><real>0.0</real><imag>9.0</imag></number>
</data>
```

Computing Tags

The names of tags can be computed using `<xsl:element>`
Likewise, attributes can be computed using `<xsl:attribute>`

Example:

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

<xsl:template match="doit">
  <xsl:element name="{hisName}">
     <xsl:attribute
         name="{hisAttribute}">Good</xsl:attribute>
     <xsl:apply-templates match="*"/>
  </xsl:element>
</xsl:template>

<xsl:template match="hisName"/>
<xsl:template match="hisAttribute"/>

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

</xsl:stylesheet>
```

Sample input:

```
<doc>
  <doit>
      <hisName>font</hisName>
      <hisAttribute>style</hisAttribute>
      <color>Red</color>
  </doit>
  <doit>
      <hisName>lunch</hisName>
      <hisAttribute>flavour</hisAttribute>
      <time>noon</time>
  </doit>
</doc>
```

Output:

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
  <font style="Good">
      <color>Red</color>
  </font>
  <lunch flavour="Good">
      <time>noon</time>
  </lunch>
</doc>
```

Recursive application of patterns

When rules are applied recursively, it is not necessary to know the depth of the input tree.

The next example takes as input mathematical expressions made up of

```
<num>k</num> <var>v</var>
<plus>a b c ...</plus> <times>a b</times>
<pow>a n</pow>
<cos>a</cos> <sin>a</sin>
```

and differentiates them with respect to x.

Sample input:

```
<doc>
<times><num>3</num><var>x</var></times>

<times><num>3</num><var>y</var></times>

<pow><var>x</var><num>4</num></pow>

<plus>
  <power><var>x</var><num>4</num></power>
  <times><num>3</num><var>x</var></times>
  <num>2</num>
  <sin><pow><var>x</var><num>3</num></pow></sin>
</plus>
</doc>
```

We use the following rules:

$$\frac{dk}{dx} = 0, \text{for} \quad \text{constant } k$$

$$\frac{dx}{dx} = 1 \qquad \frac{dz}{dx} = 0, z \neq x$$

$$\frac{d}{dx}(A + B) = \frac{dA}{dx} + \frac{dB}{dx}$$

$$\frac{d}{dx}(A \times B) = \frac{dA}{dx} \times B + A \times \frac{dB}{dx}$$

$$\frac{d}{dx}A^n = n \times A^{n-1}\frac{dA}{dx}$$

$$\frac{d}{dx}\sin(A) = \quad \cos(A) \times \frac{dA}{dx}$$

$$\frac{d}{dx}\cos(A) = -\sin(A) \times \frac{dA}{dx}$$

etc

The stylesheet is a transliteration of these rules:

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

<xsl:param name="x">x</xsl:param>

<!-- D(k) = 0 -->
<xsl:template match="num"> <num>0</num> </xsl:template>

<!-- D(var) = 1 if var = x, 0 otherwise -->
<xsl:template match="var">
  <xsl:choose>
    <xsl:when test="text() = $x"> <num>1</num> </xsl:when>
    <xsl:otherwise> <num>0</num> </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- D(a + b) = D(a) + D(b) -->
<xsl:template match="plus">
  <plus>
    <xsl:apply-templates/>
  </plus>
</xsl:template>
```

```
<!-- D(a * b) = D(a)*b + a*D(b) -->
<xsl:template match="times">
  <plus>
    <times>
       <xsl:apply-templates select="*[1]"/>
       <xsl:copy-of select="*[2]"/>
    </times>
    <times>
       <xsl:copy-of select="*[1]"/>
       <xsl:apply-templates select="*[2]"/>
    </times>
  </plus>
</xsl:template>

<!-- D(a^n) = n*a^(n-1)*D(a) -->
<xsl:template match="pow">
  <times>
    <xsl:copy-of select="*[2]"/>
    <pow>
       <xsl:copy-of select="*[1]"/>
       <num><xsl:value-of select="*[2]-1"/></num>
    </pow>
    <xsl:apply-templates select="*[1]"/>
  </times>
</xsl:template>
```

```
<!-- D(sin(a)) = cos(a)*D(a) -->
<xsl:template match="sin">
  <times>
    <cos><xsl:copy-of select="*[1]"/></cos>
    <xsl:apply-templates select="*[1]"/>
  </times>
</xsl:template>

<!-- D(cos(a)) = -sin(a)*D(a) -->
<xsl:template match="cos">
  <times>
    <num>-1</num>
    <sin><xsl:copy-of select="*[1]"/></sin>
    <xsl:apply-templates select="*[1]"/>
  </times>
</xsl:template>
</xsl:stylesheet>
```

Output for sample, reformatted:

```
<?xml version="1.0" encoding="utf-8"?>

<plus>
   <times><num>0</num><var>x</var></times>
   <times><num>3</num><num>1</num></times>
</plus>

<plus>
   <times><num>0</num><var>y</var></times>
   <times><num>3</num><num>0</num></times>
</plus>

<times>
  <num>4</num>
  <pow><var>x</var><num>3</num></pow>
  <num>1</num>
</times>
```

```
<plus>
  <times>
     <num>4</num>
     <pow><var>x</var><num>3</num>
     </pow><num>1</num>
  </times>
  <plus>
     <times><num>0</num><var>x</var></times>
     <times><num>3</num><num>1</num></times>
  </plus>
  <num>0</num>
  <times>
     <cos><pow><var>x</var><num>3</num></pow></cos>
     <times> <num>3</num> <pow><var>x</var><num>2</num></pow> <num>1</num> </times>
  </times>
</plus>
```

Functions with parameters

So far we have seen templates applied by matching patterns.

They may also be called explicitly, by name, with `<xsl:call-template>`.

Recall that a "parameter" in a template is declared with `xsl:param` and is given a default value, e.g. `<xsl:param name="diameter" select="12">`

The template selection (`<xsl:apply-templates>` or `<xsl:call-template>`) over-rides the default value by including a `xsl:with-param` element.

The following stylesheet calls a template explicitly to replace `<factorial>` elements with their computed values.

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

<xsl:template match="factorial">
  <number>
    <xsl:call-template name="do-factorial">
       <xsl:with-param name="N" select="text()"/>
    </xsl:call-template>
  </number>
</xsl:template>

<xsl:template name="do-factorial">
  <xsl:param name="Prod" select="1"/>
  <xsl:param name="N"    select="1"/>
  <xsl:choose>
     <xsl:when test="$N &gt; 1">
       <xsl:call-template name="do-factorial">
         <xsl:with-param name="Prod" select="$Prod * $N"/>
         <xsl:with-param name="N"    select="$N - 1"/>
       </xsl:call-template>
     </xsl:when>
     <xsl:otherwise>
       <xsl:value-of select="$Prod"/>
     </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

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

</xsl:stylesheet>
```

Here is the same thing using pattern matching, rather than explicit calls:

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

<xsl:template match="factorial">
  <xsl:param name="N" select="text()"/>
  <xsl:param name="Prod" select="1"/>
  <xsl:choose>
     <xsl:when test="$N &gt; 1">
        <xsl:apply-templates select=".">
           <xsl:with-param name="Prod" select="$Prod * $N"/>
           <xsl:with-param name="N"    select="$N - 1"/>
        </xsl:apply-templates>
     </xsl:when>
     <xsl:otherwise>
        <number> <xsl:value-of select="$Prod"/> </number>
     </xsl:otherwise>
  </xsl:choose>
</xsl:template>

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

</xsl:stylesheet>
```

In either case, the stylesheet converts input such as this:

```
<body>
<h1>Title</h1>
<factorial>4</factorial>
<factorial>6</factorial>
<factorial>10</factorial>.
blah blah.
</body>
```

to this

```
<?xml version="1.0" encoding="utf-8"?>
<body>
<h1>Title</h1>
<number>24</number>
<number>720</number>
<number>3628800</number>.
blah blah.
</body>
```

Processing all children with Tail recursion

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

<!-- Count the number of feet on a farm. -->
<xsl:template match="farm">
  <xsl:param name="Ix"    select="0"/>
  <xsl:param name="Feet"  select="0"/>

  <xsl:choose>
    <xsl:when test="$Ix = 0">
       <farm-summary>
          <xsl:apply-templates select=".">
             <xsl:with-param name="Ix" select="$Ix+1"/>
          </xsl:apply-templates>
       </farm-summary>
    </xsl:when>

    <xsl:when test="*[position() = $Ix]">
      <!-- Handle current child -->
      <xsl:apply-templates select="*[position() = $Ix]"/>
      <!-- Recurse on self for rest of children -->
      <xsl:apply-templates select=".">
         <xsl:with-param name="Ix"   select="$Ix   + 1"/>
         <xsl:with-param name="Feet" select="$Feet + *[position()=$Ix]/@feet"/>
      </xsl:apply-templates>
    </xsl:when>
```

```
    <xsl:otherwise>
      <!-- Base of recursion, no children left -->
      <mouths count="{$Ix - 1}"/>
      <feet count="{$Feet}"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- Rule to copy everything, including attributes -->
<xsl:template match="/|*|@*">
  <xsl:copy>
    <xsl:apply-templates select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

The input:

```
<farm>
  <human feet="2">Fred</human>
  <human feet="2">Ethel</human>
  <cow feet="4">Bossie</cow>
  <pig feet="4" smell="bad">Charlotte</pig>
  <dog feet="4">Lassie</dog>
  <fish feet="0">Goldie</fish>
</farm>
```

Gives the output:

```
<?xml version="1.0" encoding="utf-8"?>
<farm-summary>
<human feet="2">Fred</human>
<human feet="2">Ethel</human>
<cow feet="4">Bossie</cow>
<pig feet="4" smell="bad">Charlotte</pig>
<dog feet="4">Lassie</dog>
<fish feet="0">Goldie</fish>
<mouths count="6"/>
<feet count="16"/>
</farm-summary>
```