XSL Transformations

Previously we used DOM and SAX to extract data from an XML document and present that information as plain text, as HTML, or organized in a new XML document.

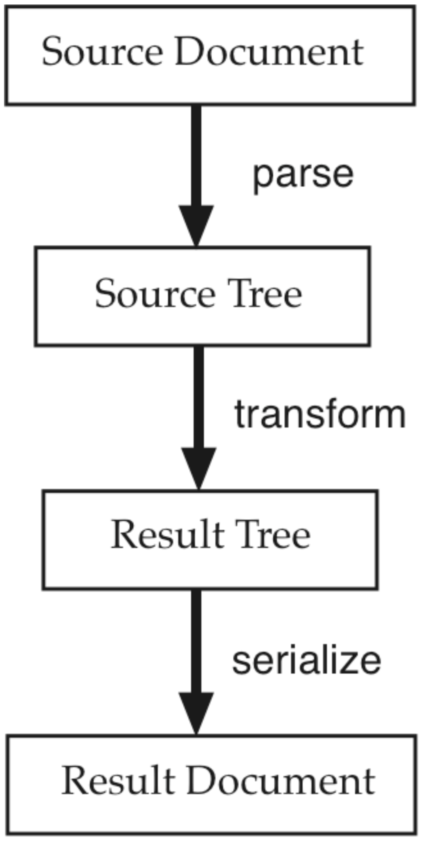
XSLT supplies another way to perform these kinds of tasks.

XSL (Extensible Stylesheet Language) is an application of XML that provides tools for transforming an XML document into some other textual form.

Features of XSLT

- XSL is written entirely in XML.
- Its elements are found in a namespace with the URI "http://www.w3.org/1999/XSL/Transform".
- XSL is a declarative programming language: we describe what we want, but we do not prescribe how to get it. SQL takes the same approach to programming.
- It is complete programming language with variables, methods, and parameters but no assignment command to change the value of a variable.
- The result of a transformation will be another XML document, an HTML document, or a text document.
- The XSLT processor parses the XML document to produce a source tree that the transformation definition works on to produce a result tree.
- XSLT uses XPath expressions to specify nodes and values in the source tree.

XSLT Process



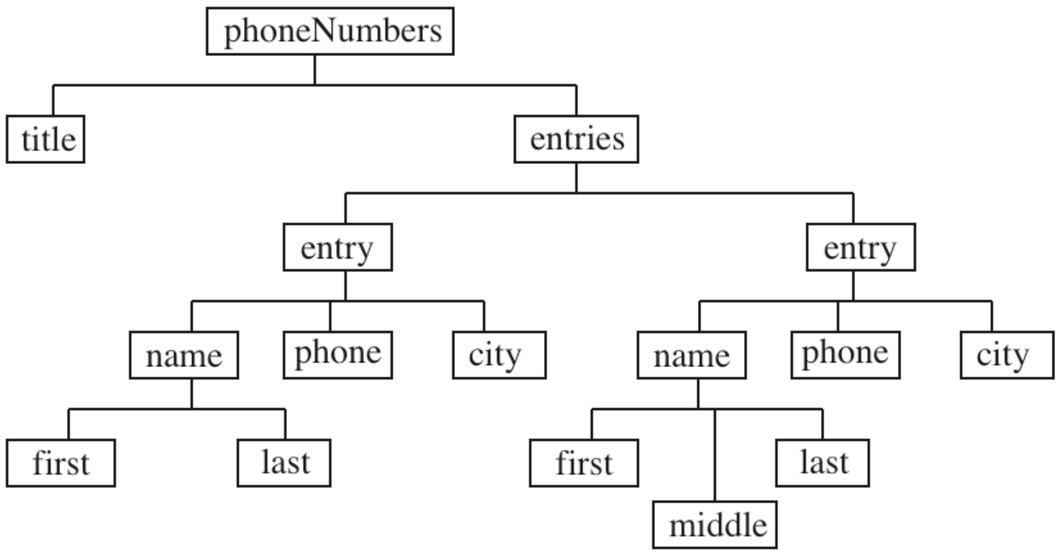
Structure of an XSLT Document

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <!-- series of templates that match items in the
         source tree and define the content to be placed in
         the result tree -->
</xsl:stylesheet>
```

Simple XSLT Examples

The basic behavior of the XSLT processor will be illustrated by a series of simple examples that work on the XML document *phoneA.xml*.

As a reminder, here is the structure of the elements found in this document. Remember that the root of the source tree lies above this tree.



An XSLT Processor

Before we are done, we will investigate several different tools for transforming an XML document using XSL.

For our first examples, we use a utility *xsltproc* that is available on the Linux machines in the CS department.

To see the various options allowed by *xsltproc* try

```
% man xsltproc
```

We invoke the program in two basic ways.

To see the result document on the screen (standard output):

```
% xsltproc ph1.xsl phoneA.xml
```

To put the result document into a file:

```
% xsltproc -o result ph1.xsl phoneA.xml
        or
% xsltproc ph1.xsl phoneA.xml >result
```

Two Top-level Items

In our first example of an XSL program we have two different items at the top level.

1. An instruction that tells the XSLT processor to produce plain text as its result (no XML declaration and no tags).
2. An example of a template rule, which has the following form

```
<xsl:template match="XPath expression">
        Text and values to place in the result document.
</xsl:template>
```

In this first example we use the XPath expression "/" that matches the root of the source tree (not the root element).

File: ph1.xsl

```
<?xml version="1.0"?>
<!-- ph1.xsl -->
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <xsl:output method="text"/>

   <xsl:template match="/">
     Found the root of the source tree.
   </xsl:template>

</xsl:stylesheet>
```

This style sheet matches the root with its template rule and sends the string "Found the root of the source tree." along with some mysterious white space to the result document.

Since the template rule has no instructions to continue processing the source tree, the string is all we get.

File: ph2.xsl

In this style sheet we ask the XSLT processor to continue looking for matches in the source tree directly below the document root where the element *phoneNumbers* lies.

This request is made by the empty element shown here.

```
<xsl:apply-templates/>
```

The element *xsl:apply-tempates* always lies inside of a template rule. Its purpose is to direct the processor to continue down into the tree looking for templates to match each of the child nodes of the current node, the one that matched the enclosing template rule.

```
<?xml version="1.0"?>
<!-- ph2.xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    Found the root of the source tree.
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="phoneNumbers">
    Found the phoneNumbers element.
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="title">
    Found the title element.
  </xsl:template>
  <xsl:template match="entries">
    Found the entries element.
  </xsl:template>

</xsl:stylesheet>
```

Observe that this style sheet continues processing in the template rule that matches the element *phoneNumbers* and stops only when it reaches the elements *title* and *entries*.

Applying ph2.xsl to phoneA.xml

```
Found the root of the source tree.

Found the phoneNumbers element.


Found the title element.


Found the entries element.
```

File: ph3.xsl

In this style sheet we add another element *xsl:value-of*, which is used to extract or compute a value for the result document.

Its *select* attribute specifies the value as an XPath.

In this particular example, the first *xsl:value-of* chooses the value of a child element of the current node and the second chooses the current node itself using the abbreviation ".". The value of an element is the concatenation of all of the textual content found inside of the element.

```
<?xml version="1.0"?>
<!-- ph3.xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
     Found the root of the source tree.
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="phoneNumbers">
     Found the phoneNumbers element.
     Title1: <xsl:value-of select="title"/>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="title">
     Found the title element.
    Title2: <xsl:value-of select="."/>
  </xsl:template>

  <xsl:template match="entries">
     Found the entries element.
    <xsl:apply-templates/>     <!-- investigate all children -->
  </xsl:template>              <!--  of entries -->

  <xsl:template match="entry">
     Found an entry element.
  </xsl:template>

</xsl:stylesheet>
```

Applying ph3.xsl to phoneA.xml

```
 Found the root of the source tree.

 Found the phoneNumbers element.
Title1: Phone Numbers

 Found the title element.
Title2: Phone Numbers

 Found the entries element.


 Found an entry element.
Found an entry element.

Found an entry element.

Found an entry element.
```

File: ph4.xsl

In this style sheet we reach down into the source tree from the root to specify a couple of particular values using XPath expressions.

```
<?xml version="1.0"?>
<!-- ph4.xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    Phone: <xsl:value-of
            select="phoneNumbers/entries/entry/phone"/>
    City: <xsl:value-of
            select="phoneNumbers/entries/entry/city"/>
  </xsl:template>

</xsl:stylesheet>
```

Applying ph4.xsl to phoneA.xml

```
Phone: 335-0055
City: Iowa City
```

Two observations

1. Selecting an element in an *xsl:value-of* element chooses the text found in the content of the selected element.

2. The element *xsl:value-of* chooses only the first instance of a node specified by the XPath.

File: ph5.xsl

The element *xsl:apply-templates* allows an attribute *select* that directs the XSLT processor to a certain set of nodes as it continues to use template rules to match elements.

In this style sheet the first template points the processor to the set of *entry* nodes.

When each of these nodes is matched, four lines of text are moved to the result document.

Inside of the template that matches *entry* elements, the current node or the context node is the *entry* element being processed.

The *xsl:value-of* elements select values of children of *entry* or an attribute that belongs to the *name* child of *entry*.

Further processing is directed to the *name* element so that its template can extract two parts from each name.

```
<?xml version="1.0"?>
<!-- ph5.xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:apply-templates
                 select="phoneNumbers/entries/entry"/>
  </xsl:template>
```

```
   <xsl:template match="entry">
      Name: <xsl:apply-templates select="name"/>
      Gender: <xsl:value-of select="name/@gender"/>
      Phone: <xsl:value-of select="phone"/>
      City: <xsl:value-of select="city"/>
   </xsl:template>

   <xsl:template match="name">
      <xsl:value-of select="first"/><xsl:value-of select="last"/>
   </xsl:template>

</xsl:stylesheet>
```

Applying ph5.xsl to phoneA.xml

```
Name: RustyNail
Gender:
Phone: 335-0055
City: Iowa City
Name: JustinCase
Gender: male
Phone: 354-9876
City: Coralville
Name: PearlGates
Gender: female
Phone: 335-4582
City: North Liberty
Name: HelenBack
Gender: female
Phone: 337-5967
City:
```

Observe that there are no spaces between the first and last names. We deal with this problem later.

Alter the Logic

In the next example observe how changing the structure of the stylesheet alters the result document.

Instead of constructing a node set containing the *entry* elements and then processing their children, this next version constructs node sets with the *name* elements, the *phone* elements, and then the *city* elements.

The result document shows the same information, but in a different order.

File: ph5a.xsl

```
<?xml version="1.0"?>
<!-- ph5a.xsl -->
<xsl:stylesheet  version="1.0"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:apply-templates
          select="phoneNumbers/entries/entry/name"/>
    <xsl:apply-templates
          select="phoneNumbers/entries/entry/phone"/>
    <xsl:apply-templates
          select="phoneNumbers/entries/entry/city"/>
  </xsl:template>

  <xsl:template match="name">
    Name: <xsl:value-of select="first"/>
              <xsl:value-of select="last"/>
    Gender: <xsl:value-of select="@gender"/>
  </xsl:template>

  <xsl:template match="phone">
    Phone: <xsl:value-of select="."/>
  </xsl:template>

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

Applying ph5a.xsl to phoneA.xml

```
Name: RustyNail
Gender:
Name: JustinCase
Gender: male
Name: PearlGates
Gender: female
Name: HelenBack
Gender: female
Phone: 335-0055
Phone: 354-9876
Phone: 335-4582
Phone: 337-5967
City: Iowa City
City: Coralville
City: North Liberty
```

An Experiment

In each of the previous examples, the style sheet has specified carefully exactly which nodes to examine.

As an experiment we delete the template rule that matches the root and thereby omit the XPath expression that leads the processor to the *entry* elements.

File: ph6.xsl

```
<?xml version="1.0"?>
<!-- ph6.xsl -->
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
```

```
<xsl:template match="entry">
   Name: <xsl:apply-templates select="name"/>
   Gender: <xsl:value-of select="name/@gender"/>
   Phone: <xsl:value-of select="phone"/>
   City: <xsl:value-of select="city"/>
</xsl:template>

<xsl:template match="name">
   <xsl:value-of select="first"/><xsl:value-of select="last"/>
</xsl:template>

</xsl:stylesheet>
```

Applying ph6.xsl to phoneA.xml

```
Phone Numbers

  Name: RustyNail
  Gender:
  Phone: 335-0055
  City: Iowa City
  Name: JustinCase
  Gender: male
  Phone: 354-9876
  City: Coralville
  Name: PearlGates
  Gender: female
  Phone: 335-4582
  City: North Liberty
  Name: HelenBack
  Gender: female
  Phone: 337-5967
  City:
```

Observe that the XSLT processor was still able to find the *entry* elements and apply the templates for *entry* and *name*.

It even found the content of the *title* element somehow.

Default Template Rules

The XSLT processor always executes in an environment that includes a set of predefined template rules.

When two template rules apply to the same node in a source tree, the processor chooses the more specific of the two. That is why the template rules in our examples have been followed.

But when our style sheet has no template rule for a particular element, the predefined default rules are applied.

The next table show the behavior of the predefined template rules.

Node Type	**Rule**
Root	Apply templates to children
Elements	Apply templates to children
Text	Copy text to result tree
Attributes	Copy value of attribute to result tree
Comments	Do nothing
Processing instructions	Do nothing
Namespace	Do nothing

File: nothing.xsl

```
<?xml version="1.0"?>
<!-- nothing.xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
</xsl:stylesheet>
```

Applying nothing.xsl to phoneA.xml

```
        Phone Numbers



              Rusty
              Nail

            335-0055
            Iowa City



              Justin
              Case

            354-9876
            Coralville



              Pearl
              E.
              Gates

            335-4582
            North Liberty



              Helen
              Back

            337-5967
```

Observe that the result document contains all of the textual content contained in the original XML document, but not the attribute values.

The strange formatting can be explained by setting the result document next to the source document. Notice that the text (and intervening white space) match up closely.

```
                          <phoneNumbers>
Phone Numbers               <title>Phone Numbers</title>
                            <entries>
                              <entry>
                                <name>
    Rusty                         <first>Rusty</first>
    Nail                          <last>Nail</last>
                                </name>
  335-0055                      <phone>335-0055</phone>
  Iowa City                     <city>Iowa City</city>
                              </entry>
                              <entry>
                                <name gender="male">
    Justin                        <first>Justin</first>
    Case                          <last>Case</last>
                                </name>
  354-9876                      <phone>354-9876</phone>
  Coralville                    <city>Coralville</city>
                              </entry>
                              <entry>
                                <name gender="female">
    Pearl                         <first>Pearl</first>
    E.                            <middle>E.</middle>
    Gates                         <last>Gates</last>
                                </name>
  335-4582                      <phone>335-4582</phone>
  North Liberty                 <city>North Liberty</city>
                              </entry>
                              <entry>
                                <name gender="female">
    Helen                         <first>Helen</first>
    Back                          <last>Back</last>
                                </name>
  337-5967                      <phone>337-5967</phone>
                              </entry>
                            </entries>
                          </phoneNumbers>
```

Removing the White Space

XSLT understands a top-level element that directs the processor to remove all the ignorable white space from the content of particular elements.

```
        <xsl:strip-space elements="name phone city"/>
```

The attribute value is a list of elements delimited by white space.

Example

Add this new element immediately after the *xsl:output* element in *nothing.xsl*.

```
        <xsl:strip-space elements="*"/>
```

In the attribute value, * indicates that the ignorable white space from all elements is to be removed.

Now the result document will contain all of the textual content of *phoneA.xml* on a single line with no white space between the element content.

```
      Phone NumbersRustyNail335-0055Iowa CityJustinCase
            354-9876CoralvillePearlE.Gates335-4582North
            LibertyHelenBack337-5967
```

File: default.xsl

This style sheet approximates what the default template rules do when we have not defined a more specific rule for an element.

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

```
  <xsl:output method="text"/>  <!-- not default -->

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

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

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

</xsl:stylesheet>
```

Abbreviations Used

- In the first template rule, * matches every kind of element node in the document.
- In the second rule, *text()* matches every text node and *@** matches every attribute.
- In the third rule, the two XPath expressions match the two obvious kinds of nodes, but this template has no content.

If this style sheet is applied to *phoneA.xml*, the result documents will be identical to the one above.

Question: Why are the attribute values not produced by the default rules?

Although all attributes are matched by the second template rule and the *xsl:apply-templates* element in the first rule directs the processor to the child nodes of the current node, attribute nodes are not consider to be children of element nodes.

To get the attributes we need an *xsl:apply-templates* element or an *xsl:value-of* element that selects the attribute like the one in *ph5.xsl*.

```
Gender: <xsl:value-of select="name/@gender"/>
```

XSLT Processing

XSLT transformation begins with a current nodes list that contains a single entry: the root, which is the entire XML document and is represented by the "/" pattern.

Processing proceeds as follows

1. For each node X in the current nodes list, the processor searches for all *<xsl:template match="pattern">* elements in the style sheet that match that node. From this list of templates, the one with the best (most specific) match is selected.
2. The selected template is instantiated using node X as its current node. This template typically copies data from the source document to the result tree or produces brand new content in combination with data from the source.
3. If the template contains an element *<xsl:apply-templates select="newPattern"/>*, a new current node list is created and the process repeats recursively. The select pattern is defined relative to node X, rather than the root.

As the XSLT transformation process continues, the current node and current node list are constantly changing.

Conditional Processing 1

When processing an XML document to construct a result document, we sometimes have situations where the choice of what text to move to the result tree depends on what we have found in the source tree.

One way to implement this kind of decision making is the *xsl:if* element.

It requires an attribute *test* whose value is an XPath expression that is interpreted as a boolean value. If the value is true, the contents of the *xsl:if* element are processed to create text for the result document. If it is false, the entire *xsl:if* element is ignored.

XPath expression result types

Recall that XPath expressions have one of four data types.

- Node set
- String
- Number
- Boolean

Each of these XPath types can be viewed as a boolean value.

Node set

An empty node set is false, and a non-empty node set is true.

This property means that an XPath that specifies a particular element or attribute is false if that element or attribute is missing.

String

An empty string (no characters) is false, and a non-empty string is true.

Number

Zero is false, NaN (Not a Number) is false, and any other value is true.

Boolean

XPath has a number of boolean operations that can be used in decision making, including =, !=, <, <=, >, >=, *and*, *or*, *true()*, *false()*, and *not(XPathExpr)*. Remember to use an entity reference for < and maybe for >.

When a string is needed inside of an attribute value delimited by quote characters, used apostrophes to delimit the string.

Sample Boolean Expressions

test="true()"	always succeeds
test="true"	succeeds if there is a child element *true* in the current context
test="'true'"	always succeeds
test="'false'"	always succeeds
test="not(5)"	always fails
test="count(entry) &lt; 3"	succeeds if there are less than 3 child elements *entry* in the current context
test="name/@gender"	succeeds if there is an attribute *gender* belonging to a *name* child in the current context
test="not('false')"	always fails

Applying Decision Making

Remember that when we used the stylesheet *ph5.xsl* to transform *phoneA.xm*l, the Gender and City labels were entered in the result document even if these items were missing.

In the next stylesheet, we solve this problem using *xsl:if* elements.

A second improvement in this XSLT definition will involve taking more control over the format of the result document.

The *xsl:text* element instructs the processor to insert the textual content of this element into the result document as is.

We use this feature to place a space between the first and last names and to add a new line after each entry.

File: phif.xsl

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

  <xsl:template match="/">
    <xsl:apply-templates
            select="phoneNumbers/entries/entry"/>
  </xsl:template>

  <xsl:template match="entry">
    Name: <xsl:apply-templates select="name"/>
    <xsl:if test="name/@gender">
    Gender: <xsl:value-of select="name/@gender"/>
    </xsl:if>
    Phone: <xsl:value-of select="phone"/>
    <xsl:if test="city">
    City: <xsl:value-of select="city"/>
    </xsl:if>
    <xsl:text>
    </xsl:text>
  </xsl:template>

  <xsl:template match="name">
    <xsl:value-of select="first"/><xsl:text> </xsl:text>
    <xsl:value-of select="last"/>
  </xsl:template>

</xsl:stylesheet>
```

The lines with "Gender: " and "City: " are not indented so that when they are transferred to the result document, the four labels line up on the left.

Applying phif.xsl to phoneA.xml

```
Name: Rusty Nail
Phone: 335-0055
City: Iowa City

Name: Justin Case
Gender: male
Phone: 354-9876
City: Coralville

Name: Pearl Gates
Gender: female
Phone: 335-4582
City: North Liberty

Name: Helen Back
Gender: female
Phone: 337-5967
```

Controlling Whitespace

In these examples, the positions of the lines produced by the stylesheet are determined by the positions of the literal text ("Name: ", "Gender: ", and so on) in the stylesheet itself.

We can take complete control of this positioning by placing all literal strings inside of *xsl:text* elements.

In this case, we must supply any needed whitespace, for example new lines, explicitly.

In the next stylesheet observe how new lines are providing without any indenting of the following line of text.

File: phtext.xsl

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

  <xsl:template match="/">
    <xsl:apply-templates
               select="phoneNumbers/entries/entry"/>
  </xsl:template>

  <xsl:template match="entry">
    <xsl:text>Name: </xsl:text>
               <xsl:apply-templates select="name"/>
    <xsl:text>
</xsl:text>
    <xsl:if test="name/@gender">
      <xsl:text>Gender: </xsl:text>
               <xsl:value-of select="name/@gender"/>
      <xsl:text>
</xsl:text>
    </xsl:if>
    <xsl:text>Phone: </xsl:text>
               <xsl:value-of select="phone"/>
    <xsl:text>
</xsl:text>
    <xsl:if test="city">
      <xsl:text>City: </xsl:text><xsl:value-of select="city"/>
      <xsl:text>
</xsl:text>
    </xsl:if>
    <xsl:text>
</xsl:text>
  </xsl:template>
```

```
   <xsl:template match="name">
      <xsl:value-of select="first"/><xsl:text> </xsl:text>
      <xsl:value-of select="last"/>
   </xsl:template>
</xsl:stylesheet>
```

Applying phtext.xsl to phoneA.xml

```
Name: Rusty Nail
Phone: 335-0055
City: Iowa City

Name: Justin Case
Gender: male
Phone: 354-9876
City: Coralville

Name: Pearl Gates
Gender: female
Phone: 335-4582
City: North Liberty

Name: Helen Back
Gender: female
Phone: 337-5967
```

Iteration

To process all of the nodes that meet some criterion, we can use the *xsl:for-each* element with its *select* attribute.

This element lets us choose a set of nodes, and then it applies the instructions found as the content of the *xsl:for-each* element to each node in the set in document order.

In effect, the content of the *xsl:for-each* element forms a template that is applied to each element in the node set defined by its *select* attribute.

To illustrate this new element, we rewrite the stylesheet *phif.xsl* using *xsl:for-each* in place of *xsl:apply-templates*.

Observe that we choose the name elements *first* and *last* directly rather than using a separate template and an *xsl:apply-templates* element.

File: phfor.xsl

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

  <xsl:template match="/">
    <xsl:for-each select="phoneNumbers/entries/entry">
      Name: <xsl:value-of select="name/first"/>
            <xsl:text> </xsl:text>
            <xsl:value-of select="name/last"/>
      <xsl:if test="name/@gender">
      Gender: <xsl:value-of select="name/@gender"/>
      </xsl:if>
      Phone: <xsl:value-of select="phone"/>
      <xsl:if test="city">
      City: <xsl:value-of select="city"/>
      </xsl:if>
      <xsl:text>
      </xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The result document will be identical to that produced with the *phif.xsl* stylesheet.

To have the header labels line up in the result document, we must have them line up in the stylesheet.

Transforming to XML

In the next example we want to translate *phoneA.xml* into another XML document with the same structure, but we want to replace all of the element tags and attribute names with their German equivalents.

Here is an XML document with only two entries that shows the format we want to produce.

```
<?xml version="1.0"?>
<Telefonnummern>
  <Titel>Phone Numbers</Titel>
  <Eintraege>
   <Eintrag>
     <Name>
        <Vorname1>Rose</Vorname1>
       <Nachname>Bush</Nachname>
     </Name>
     <Telefonnummer>335-6545</Telefonnummer>
     <Stadt>Iowa City</Stadt>
   </Eintrag>
   <Eintrag>
     <Name Geschlect="female">
        <Vorname1>Bertha</Vorname1>
        <Vorname2>D.</Vorname2>
        <Nachname>Blues</Nachname>
     </Name>
     <Telefonnummer>335-2862</Telefonnummer>
   </Eintrag>
  </Eintraege>
</Telefonnummern>
```

Observe that a *name* element may have no *gender* attribute and an *entry* element may have no *city* child.

Strategy

The style sheet needs a template for every element in the XML document *phoneA.xml*.

- The content of each template will contain start and end tags for the element matched except that the tag names will be in German.
- The templates for elements that have children that are elements will contain an *xsl:apply-templates* element to force the XSLT processor to continue matching template for those children elements.
- The templates for elements that have textual content will contain an *xsl:value-of* element selecting the current node to move that text to the result tree.

The *gender* attribute needs to be handled in a special way.

In the result document we expect to see start tags of the form:

```
<Name Geschlect="male">
<Name Geschlect="female">
```

Here is the template that we might expect would work for the *name* element.

```
<xsl:template match="name">
    <Name Geschlect="@gender">
      <xsl:apply-templates/>
    </Name>
  </xsl:template>
```

The problem is that the attribute value "@gender" is interpreted as literal text, so the start tag will look like this.

```
<Name Geschlet="@gender">
```

XSLT has a special syntax, called *attribute value templates*, that can be used to force the evaluation of an XPath expression: Place the expression inside braces.

The correct version of the template for the *name* element must use those braces.

```
<xsl:template match="name">
    <Name Geschlect="{@gender}">
      <xsl:apply-templates/>
    </Name>
  </xsl:template>
```

File: german.xsl

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

  <xsl:output method="xml"/>    <!-- a redundant element -->

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

  <xsl:template match="phoneNumbers">
    <Telefonnummern>
      <xsl:apply-templates/>
    </Telefonnummern>
  </xsl:template>

  <xsl:template match="title">
    <Titel>
       <xsl:value-of select="."/>
    </Titel>
  </xsl:template>
```

```
<xsl:template match="entries">
  <Eintraege>
     <xsl:apply-templates/>
  </Eintraege>
</xsl:template>

<xsl:template match="entry">
  <Eintrag>
     <xsl:apply-templates/>
  </Eintrag>
</xsl:template>

<xsl:template match="name">
   <xsl:if test="@gender">
     <Name Geschlect="{@gender}">
       <xsl:apply-templates/>
     </Name>
  </xsl:if>
   <xsl:if test="not(@gender)">
     <Name>                    <!-- no attribute -->
       <xsl:apply-templates/>
     </Name>
  </xsl:if>
</xsl:template>

<xsl:template match="first">
   <Vorname1>
      <xsl:value-of select="."/>
   </Vorname1>
</xsl:template>

<xsl:template match="middle">
   <Vorname2>
      <xsl:value-of select="."/>
   </Vorname2>
</xsl:template>

<xsl:template match="last">
  <Nachname>
      <xsl:value-of select="."/>
  </Nachname>
</xsl:template>
```

```
    <xsl:template match="phone">
      <Telefonnummer>
        <xsl:value-of select="."/>
      </Telefonnummer>
    </xsl:template>
    <xsl:template match="city">
      <Stadt>
        <xsl:value-of select="."/>
      </Stadt>
    </xsl:template>
  </xsl:stylesheet>
```

Notes on *german.xsl*

Observe how the *xsl:if* element is used to expression an if-then-else structure.

```
      <xsl:if test="@gender">
        <Name Geschlect="{@gender}">
          <xsl:apply-templates/>
        </Name>
      </xsl:if>
      <xsl:if test="not(@gender)">
        <Name>                    <!-- no attribute -->
          <xsl:apply-templates/>
        </Name>
      </xsl:if>
```

Conditional Processing 2

The logic in the example above can be handled by the *xsl:choose* element as well.

In addition, *xsl:choose* can express the more complicated logic of a case or switch structure.

The *xsl:choose* element has the following form:

```
<xsl:choose>
  <xsl:when test="condition1">
    output for result tree when condition1 is true
  </xsl:when>
  <xsl:when test="condition2">
    output for result tree when condition2 is true
  </xsl:when>
  <xsl:otherwise>
    output for result tree when both conditions are false
  </xsl:otherwise>
</xsl:choose>
```

For the German translation we can use the following code.

```
<xsl:choose>
  <xsl:when test="@gender">
    <Name Geschlect="{@gender}">
      <xsl:apply-templates/>
    </Name>
  </xsl:when>
  <xsl:otherwise>
    <Name>                <!-- no attribute -->
      <xsl:apply-templates/>
    </Name>
  </xsl:otherwise>
</xsl:choose>
```

More on Attribute Value Templates

The value used in the attribute value template may be an element value as well as an attribute value.

For example, change the preceding example to define the phone number as an attribute for the *entry* (*Eintrag*) element and remove the actual *phone* element.

Change the templates for the *entry* element and for the *phone* element.

```
<xsl:template match="entry">
  <Eintrag Telefonnummer="{phone}">
    <xsl:apply-templates/>
  </Eintrag>
</xsl:template>

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

Observe how the *phone* element is removed completely from the result tree by supplying an *xsl:template* element with no content.

Another Way to Create Elements

In the previous example translating *phoneA.xml* into an XML document with German tags, we used literal text to form elements and attributes in the result tree.

XSLT has elements of its own for creating elements and attributes in the result tree.

Create an Element

```
<xsl:element name="elementTag">
    content of element
</xsl:element>
```

Create an Attribute

```
<xsl:attribute name="attributeName">
    value of attribute
</xsl:attribute>
```

The *xsl:attribute* element(s) must occur as the first item(s) inside of an element that is being created.

Using these XSLT elements, we can write a stylesheet for transforming *phoneA.xml* into its German equivalent.

The *xsl:output* element of this stylesheet contains an attribute *indent* to improve the formatting of the result document slightly.

File: g.xsl

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

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="phoneNumbers">
    <xsl:element name="Telefonnummern">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="title">
    <xsl:element name="Titel">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="entries">
    <xsl:element name="Eintraege">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="entry">
    <xsl:element name="Eintrag">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
```

```
<xsl:template match="name">
  <xsl:choose>
    <xsl:when test="@gender">
      <xsl:element name="Name">
        <xsl:attribute name="Geschlect">
          <xsl:value-of select="@gender"/>
        </xsl:attribute>
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:when>
    <xsl:otherwise>
      <xsl:element name="Name">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template match="first">
  <xsl:element name="Vorname1">
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:template>
<xsl:template match="middle">
  <xsl:element name="Vorname2">
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:template>
<xsl:template match="last">
  <xsl:element name="Nachname">
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:template>
<xsl:template match="phone">
  <xsl:element name="Telefonnummer">
    <xsl:value-of select="."/>
  </xsl:element>

</xsl:template>
```

```
    <xsl:template match="city">
      <xsl:element name="Stadt">
        <xsl:value-of select="."/>
      </xsl:element>
    </xsl:template>
  </xsl:stylesheet>
```

The result document is essentially the same as with the previous stylesheet *german.xsl*.

Using the elements *xsl:element* and *xsl:attribute* contributes no more capability than using literal elements and attributes in this example.

The power of these elements comes from situations where we need to discover or compute the names of the new elements and attributes dynamically, which means they depend on the contents of the source tree.

Creating Elements and Attributes Dynamically

To illustrate the power of XSLT, we transform an XML document by creating entirely new attributes and elements that depend on the original document.

For testing purposes, we start with a simple XML document that contains multiple elements and attributes.

File: dynamic.xml

```
<?xml version="1.0"?>
<!-- dynamic.xml -->
<root>
  <items>
    <item position="01" code="a25">
      <id>FX483</id>
      <name>Element1</name>
      <description>Debris</description>
    </item>
    <item position="02" code="b38">
```

```
        <id>FH390</id>
        <name>Element2</name>
        <description>Junk</description>
      </item>
      <item position="03" code="a88">
        <id>FA881</id>
        <name>Element3</name>
        <description>Trash</description>
      </item>
    </items>
  </root>
```

The first stylesheet will be concerned only with creating elements from the attributes in the source document.

Meanwhile, the existing simple elements (those with textual content only) will be deleted.

File: mkElements.xsl

```
<?xml version="1.0"?>
<!-- mkElements.xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <newRoot>
      <newItems>
        <xsl:apply-templates select="root/items/item"/>
      </newItems>
    </newRoot>
  </xsl:template>
  <xsl:template match="item">
    <newItem>
      <xsl:for-each select="@*">
        <xsl:element name="{name()}">
          <xsl:value-of select="."/>
        </xsl:element>
      </xsl:for-each>
      </newItem>
    </xsl:template>
</xsl:stylesheet>
```

The strategy of this stylesheet is to process each of the *item* elements in the source tree, iterating through the attributes for each of these elements, creating a corresponding element whose content is the attribute value.

Observe that the name of the new element results from an attribute value template containing the name of the attribute being processed.

The XPath expression *name()* is a call to a predefined function that returns the name of the node currently being processed.

Applying mkElements.xsl to dynamic.xml

```
<?xml version="1.0">
<newRoot>
  <newItems>
    <newItem>
      <position>01</position>
      <code>a25</code>
    </newItem>
    <newItem>
      <position>02</position>
      <code>b38</code>
    </newItem>
    <newItem>
      <position>03</position>
      <code>a88</code>
    </newItem>
  </newItems>
</newRoot>
```

In the second example we change the existing elements that are children of *item* elements into attributes of the *newItem* elements.

Existing attributes of the *item* elements will again be made into elements that are children of the *newItem* elements as in the stylesheet *mkElements.xsl*.

Remember that the attributes of a new element must be created before the content of the element is defined.

File: mkEandA.xsl

```
<?xml version="1.0"?>
<!-- mkEandA.xsl -->
<xsl:stylesheet  version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" indent="yes"/>
   <xsl:template match="/">
     <newRoot>
       <newItems>
          <xsl:apply-templates select="root/items/item"/>
       </newItems>
     </newRoot>
   </xsl:template>
   <xsl:template match="item">
     <newItem>
        <xsl:apply-templates select="*"/>  <!-- need attribute -->
        <xsl:for-each select="@*">
          <xsl:element name="{name()}">
            <xsl:value-of select="."/>
          </xsl:element>
        </xsl:for-each>
     </newItem>
   </xsl:template>
   <xsl:template match="*">
      <xsl:attribute name="{name()}">
        <xsl:value-of select="."/>
      </xsl:attribute>
   </xsl:template>
</xsl:stylesheet>
```

Notes on mkEandA.xsl

In the template rule that matches the element *item*, we first apply the stylesheet templates to the children elements (*id*, *name*, and *description*). The *select* attribute is required to avoid an error from the XSLT processor.

The last template rule matches these children elements and creates an attribute for each.

In the second part of the template rule that matches the element *item*, we iterate through the attributes of the element, creating a new element corresponding to each as in the previous example.

Applying mkEandA.xsl to dynamic.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<newRoot>
  <newItems>
    <newItem id="FX483" name="Element1"
                                description="Debris">
      <position>01</position>
      <code>a25</code>
    </newItem>
    <newItem id="FH390" name="Element2"
                                description="Junk">
      <position>02</position>
      <code>b38</code>
    </newItem>
    <newItem id="FA881" name="Element3"
                                description="Trash">
      <position>03</position>
      <code>a88</code>
    </newItem>
  </newItems>
</newRoot>
```

XSLT in Java

The tools for processing XSLT in Java can be found in two packages.

Classes in *javax.xml.transform*

TransformerFactory

Transformer

TransformerException

Classes in *javax.xml.transform.stream*

StreamSource

StreamResult

XSLT Processing Step-by-Step

1. Get two StreamSource objects that encapsulate the XML source document and the XSLT stylesheet.

   This class has several constructors that take different forms of input specification. Check the documentation for other versions.

   ```
   File oldFile = new File(args[0]);
   StreamSource oldStream = new StreamSource(oldFile);

   File xslFile = new File(args[1]);
   StreamSource xslStream = new StreamSource(xslFile);
   ```

2. Get a StreamResult object that encapsulates the result XML document.

   This class has several constructors that take different forms of output specification. See documentation.

   ```
   File newFile = new File(args[2]);
   StreamResult newStream = new StreamResult(newFile);
   ```

3. Get a TransformerFactory object.

```
TransformerFactory factory =
                    TransformerFactory.newInstance();
```

4. Get a Transformer object that encapsulates the XSLT stylesheet.

```
Transformer transformer =
               factory.newTransformer(xslStream);
```

5. Call the *transform* method on the Transformer object to transform the source document into the result document.

```
transformer.transform(oldStream, newStream);
```

File: MyTransform.java

```
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import java.io.*;

public class MyTransform
{
   public static void main(String [] args)
                     throws TransformerException,
                            FileNotFoundException
   {
      if (args.length != 3)
      {
         System.out.print ("Usage: java MyTransform ");
         System.out.println("old.xml style.xsl new.xml");
         return;
      }
```

```
          File oldFile = new File(args[0]);
          StreamSource oldStream =
                                  new StreamSource(oldFile);

          File xslFile = new File(args[1]);
          StreamSource xslStream =
                                  new StreamSource(xslFile);

          File newFile = new File(args[2]);
          StreamResult newStream =
                                  new StreamResult(newFile);

          TransformerFactory factory =
                      TransformerFactory.newInstance();

          Transformer transformer =
                      factory.newTransformer(xslStream);

          transformer.transform(oldStream, newStream);
     }
  }
```

Running MyTransform.java

```
% java MyTransform phoneA.xml gernan.xsl gPhone.xml
```

XSLT in Saxon

Saxon is an open source program for processing XML documents written by Michael Kay.

It is now at version 8.8.

Installing Saxon

1. Download the file *saxon8.8.jar* from the course ftp site.
2. Move *saxon8.8.jar* into your *bin* directory.
3. In your *.cshrc* or *.tcshrc* file, add a path specification at the end of the "setenv CLASSPATH" declaration.

   ```
   :/space/userId/bin/saxon8.8.jar
   ```

4. Execute *source .cshrc* or *source .tcshrc* for immediate use of Saxon.

Executing Saxon

The transform operation is found in the class net.sf.saxon.Transform, which expects two command-line arguments, the XML source document followed by the XSLT stylesheet.

```
% java net.sf.saxon.Transform phoneA.xml german.xsl
Warning: Running an XSLT 1.0 stylesheet with an XSLT 2.0
processor.

<?xml version="1.0" encoding="UTF-8"?><Telefonnummern>
   <Titel>Phone Numbers</Titel>
   <Eintraege>
     <Eintrag>
       <Name>
          <Vorname1>Rusty</Vorname1>
         <Nachname>Nail</Nachname>
       </Name>
```

```
        <Telefonnummer>335-0055</Telefonnummer>
        <Stadt>Iowa City</Stadt>
    </Eintrag>
    <Eintrag>
        <Name Geschlect="male">
            <Vorname1>Justin</Vorname1>
            <Nachname>Case</Nachname>
        </Name>
        <Telefonnummer>354-9876</Telefonnummer>
        <Stadt>Coralville</Stadt>
    </Eintrag>
            :
```

Saxon 8 has implemented XSLT 2.0, so a warning message is given when the stylesheet says "version=1.0".

If the warning troubles you, just change the *stylesheet* attribute to "version=2.0".

To write the result document to a file, use the Unix redirection feature.

```
% java net.sf.saxon.Transform ph.xml g.xsl >g.xml
Warning: Running an XSLT 1.0 stylesheet with an XSLT 2.0
processor.
```

To avoid writing the qualified class name every time you execute Saxon, add an alias to your *.cshrc* file.

```
alias saxon 'java net.sf.saxon.Transform \!:1 \!:2'
```

Then you can run the Saxon XSLT processor using the alias.

```
% saxon ph.xml g.xsl | cat >g.xml
```

The link *Saxon Information* on the course web page leads to Saxonica, a web page with more information about Saxon.

XSLT Processors: Information

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

  <xsl:template match="/">
    Version Supported: <xsl:value-of
                    select="system-property('xsl:version')"/>
    XSLT Processor: <xsl:value-of
                  select="system-property('xsl:vendor')"/>
    More Information: <xsl:value-of
                   select="system-property('xsl:vendor-url')"/>
  </xsl:template>
</xsl:stylesheet>
```

Three Results

xsltproc

```
Version Supported: 1.0
XSLT Processor: libxslt
More Information: http://xmlsoft.org/XSLT/
```

Java

```
Version Supported: 1.0
XSLT Processor: Apache Software Foundation (Xalan XSLTC)
More Information: http://xml.apache.org/xalan-j
```

Saxon8

```
Version Supported: 2.0
XSLT Processor: SAXON 8.2 from Saxonica
More Information: http://www.saxonica.com/
```

Using XSLT to Produce HTML

The information in an XML document is defined in terms of the semantics of the data, not in terms of display properties.

Presenting that information on a web page can be handled in several ways.

Cascading Stylesheets (CSS)

This technique of defining presentation characteristics was developed for HTML originally, but CSS works even better when applied to XML documents.

The idea is that a CSS specification contains a number of rules for formatting the various elements in the XML document.

Example

```
entry {  display: block;
         font-size: 16pt;
         color: black;
         background-color: yellow;
      }
```

CSS Has Many Limitations

- Implementations are not uniform.
- Only limited content can be added to the document.
- The content cannot be transformed, say by sorting.
- Cannot decide which elements should be displayed and which should be omitted.
- Cannot calculate values such as sums and averages.

Presenting Phone Numbers using CSS

Create a CSS stylesheet that describes the format of the data for a browser.

File: phone.css

```
/* File: phone.css */

phoneNumbers  { background-color: yellow;
              }

title  { display: block;
         color: navy;
         margin-top: 0.5in ;
         font-size: 32;
         font-variant: small-caps;
         font-weight: bold
       }

entry { display: block;
        border: 3pt solid black;
        width: 60%;
        background-color: aqua;
        border-color: teal;
        font-size: 18;
        margin: 0.5cm;
        padding: 5mm
      }

name   { display: block;
         color: navy;
         font-style: italic
       }
```

```
name:before  { content: "Name: " }

phone { display: block;
        color: red;
        text-decoration: blink;
        text-indent: 1cm
      }

phone:before { content: "Phone: " }

city  { display: block;
        color: red;
        text-indent: 1cm
      }

city:before { content: "City: " }
```

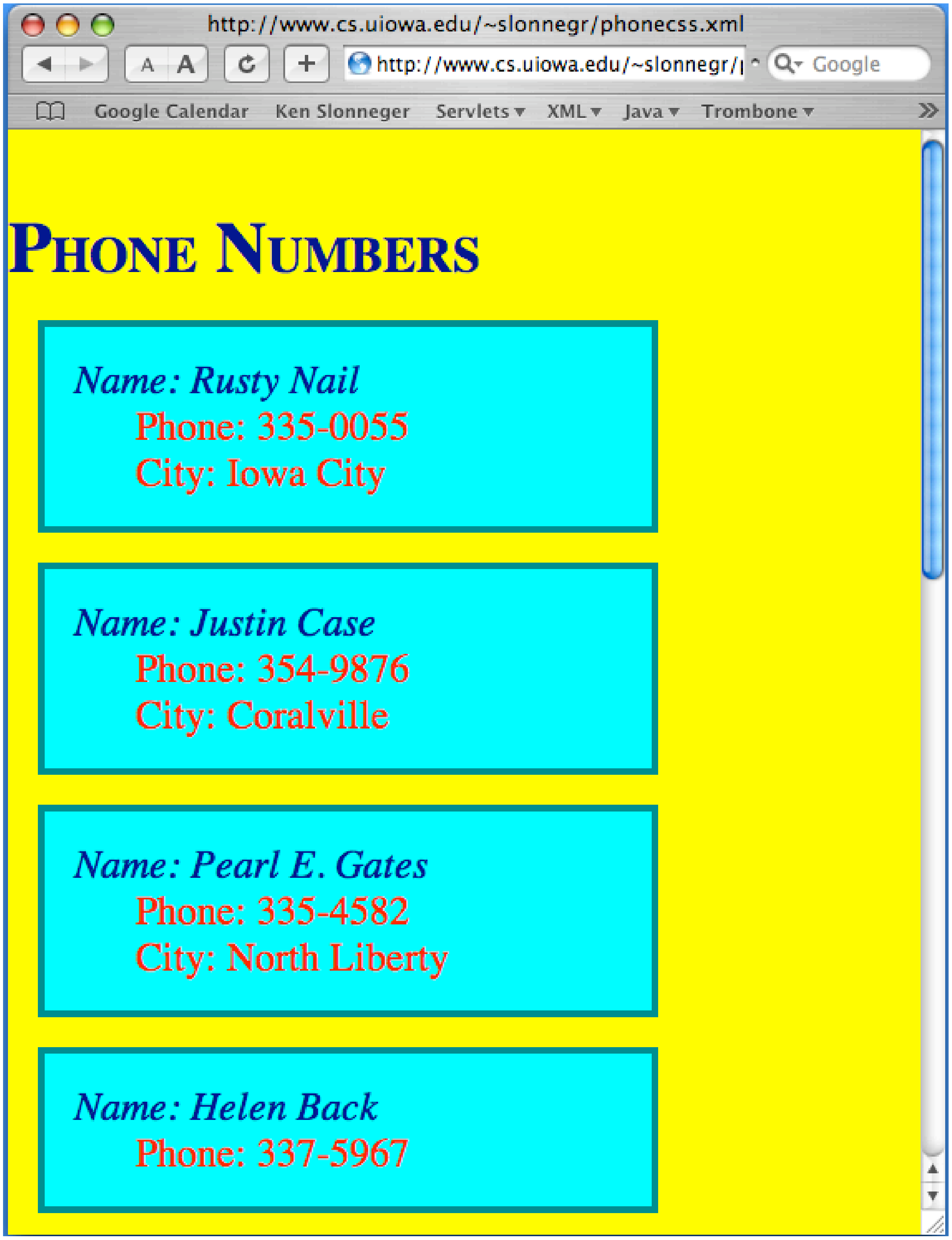
XML Document: phonecss.xml

The XML document needs to indicate the source and type of the CSS stylesheet in a processing instruction.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM "phoneA.dtd">
<?xml-stylesheet href="phone.css" type="text/css"?>

<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
   <entry>
     <name>
       <first>Rusty</first>
       <last>Nail</last>
     </name>
     <phone>335-0055</phone>
     <city>Iowa City</city>
   </entry>
       :
```

Resulting Web Page


http://www.cs.uiowa.edu/~slonnegr/phonecss.xml
http://www.cs.uiowa.edu/~slonnegr/
Google
Google Calendar
Ken Slonneger
Servlets
XML
Java
Trombone

PHONE NUMBERS

Name: Rusty Nail
Phone: 335-0055
City: Iowa City

Name: Justin Case
Phone: 354-9876
City: Coralville

Name: Pearl E. Gates
Phone: 335-4582
City: North Liberty

Name: Helen Back
Phone: 337-5967

Transforming XML into HTML

The second possibility is to transform an XML document into HTML or XHTML using XSLT.

This operation will be similar to the steps we take to translate an XML document into another XML document.

Inside of the templates we produce a combination of literal text, in the form of start and end tags, and values extracted from the XML document using *xsl:value-of* and *xsl:apply-templates*.

HTML Translation Template

```
<xsl:template match="/">
  <html>
    <head>
      <title>title of page goes here</title>
    </head>
    <body>
      mixture of
        • literal html tags
        • xsl:value-of to select values to markup
        • xsl:apply-templates elements that select
              subtrees to process
    </body>
  </html>
```

Example

In this stylesheet we build an HTML document (note the *xsl:output* element) that displays the *entry* elements of *phoneA.xml* in a table with headers.

Observe where the literal HTML tags are placed so that they appear once or many times depending on their roles in the HTML document.

In the following stylesheet we use *xsl:if* to determine whether the *gender* attributes and *city* elements are present.

If either is missing, the spot in the table will be filled with the word "unknown".

File: html.xsl

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

  <xsl:template match="/">
    <html>
      <head>
        <title>
          <xsl:value-of select="phoneNumbers/title"/>
        </title>
      </head>
      <body>
        <hr></hr>
        <table border="2" cellpadding="5">
          <tr bgcolor="#ffaacc">
            <th>Name</th>
            <th>Gender</th>
            <th>Phone</th>
            <th>City</th>
          </tr>
          <xsl:apply-templates
                  select="phoneNumbers/entries/entry"/>
        </table>
        <hr/>
      </body>
    </html>
  </xsl:template>
```

```
      <xsl:template match="entry">
        <tr>
          <td><xsl:apply-templates select="name"/></td>
          <td>
            <xsl:if test="name/@gender">
              <xsl:value-of select="name/@gender"/>
            </xsl:if>
            <xsl:if test="not(name/@gender)">
              <xsl:value-of select="'unknown'"/>
            </xsl:if>              <!-- note apostrophes -->
          </td>
          <td><xsl:value-of select="phone"/></td>
          <td>
            <xsl:if test="city">
              <xsl:value-of select="city"/>
            </xsl:if>
            <xsl:if test="not(city)">unknown</xsl:if>
          </td>
        </tr>
      </xsl:template>

      <xsl:template match="name">
        <xsl:value-of select="first"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="last"/>
      </xsl:template>

    </xsl:stylesheet>
```

Note the need for apostrophes when we want to return the string "unknown" when the attribute is missing.

Without apostrophes, <xsl:value-of select="unknown"/> is looking for an element *unknown* in the current context.

Applying html.xsl to phoneA.xml

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html;
                                        charset=UTF-8">
<title>Phone Numbers</title>
</head>
<body>
<hr>
<table border="2" cellpadding="5">
<tr bgcolor="#ffaacc">
<th>Name</th><th>Gender</th><th>Phone</th><th>City</th>
</tr>
<tr>
<td>Rusty Nail</td><td>unknown</td>
                        <td>335-0055</td><td>Iowa City</td>
</tr>
<tr>
<td>Justin Case</td><td>male</td>
                        <td>354-9876</td><td>Coralville</td>
</tr>
<tr>
<td>Pearl Gates</td><td>female</td>
                   <td>335-4582</td><td>North Liberty</td>
</tr>
<tr>
<td>Helen Back</td><td>female</td>
                        <td>337-5967</td><td>unknown</td>
</tr>
</table>
<hr>
</body>
</html>
```

Notes on HTML Translations

Browsers are notoriously cavalier in enforcing rules for HTML in web pages.

But when we put HTML tags inside the content of an *xsl:template* element, the well-formedness rules of XML must be followed.

- Every start tag needs a matching end tag, and these tags must agree in their use of uppercase and lowercase.
- Elements must nest properly.
- Attribute values must occur inside quote symbols.

The transformation of an XML document to HTML may occur at several different points of time.

Offline Transformation

Here the XML document is translated into an HTML document once before it is installed in the web page directory.

Server-side Transformation

The server waits until the page is requested and then does the translation at that time before sending the HTML document to the client. This approach keeps the information on the web page up-to-date.

Client-side Transformation

The XML document is sent to the client whose browser then requests the associated XSL document. The translation is performed on the client machine. This approach ensures that the information is kept current and reduces the load on the server, but it requires that the browser can perform an XSL transformation.

Linking XML Document and XSL Document

A stylesheet is linked to an XML document by placing a processing instruction into the XML file before its root element, identifying the location and type of the file that contains the XSL code.

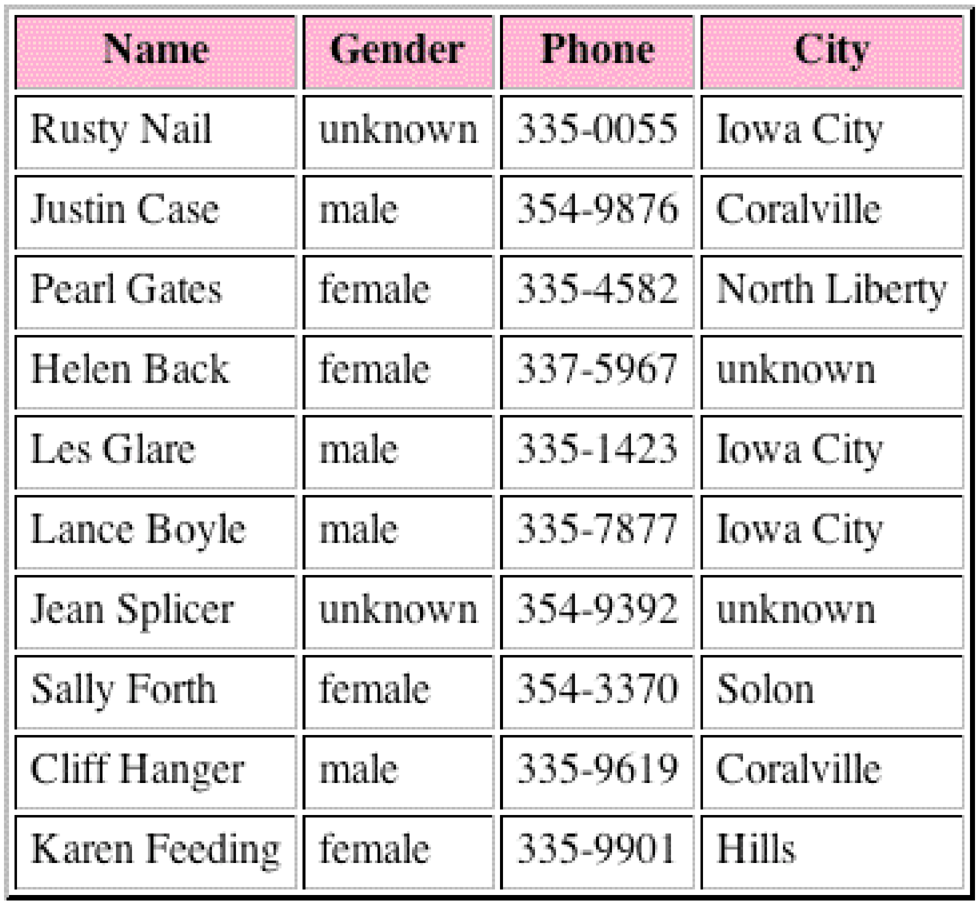
Example: phoneH.xml

Create a version of *phoneA.xml* that contains the processing instruction:

```
<?xml-stylesheet href="html.xsl" type="text/xsl"?>
```

Now place both files, *phoneH.xml* and *html.xsl*, into a web directory.

Table Shown in a Browser



Name	Gender	Phone	City
Rusty Nail	unknown	335-0055	Iowa City
Justin Case	male	354-9876	Coralville
Pearl Gates	female	335-4582	North Liberty
Helen Back	female	337-5967	unknown
Les Glare	male	335-1423	Iowa City
Lance Boyle	male	335-7877	Iowa City
Jean Splicer	unknown	354-9392	unknown
Sally Forth	female	354-3370	Solon
Cliff Hanger	male	335-9619	Coralville
Karen Feeding	female	335-9901	Hills

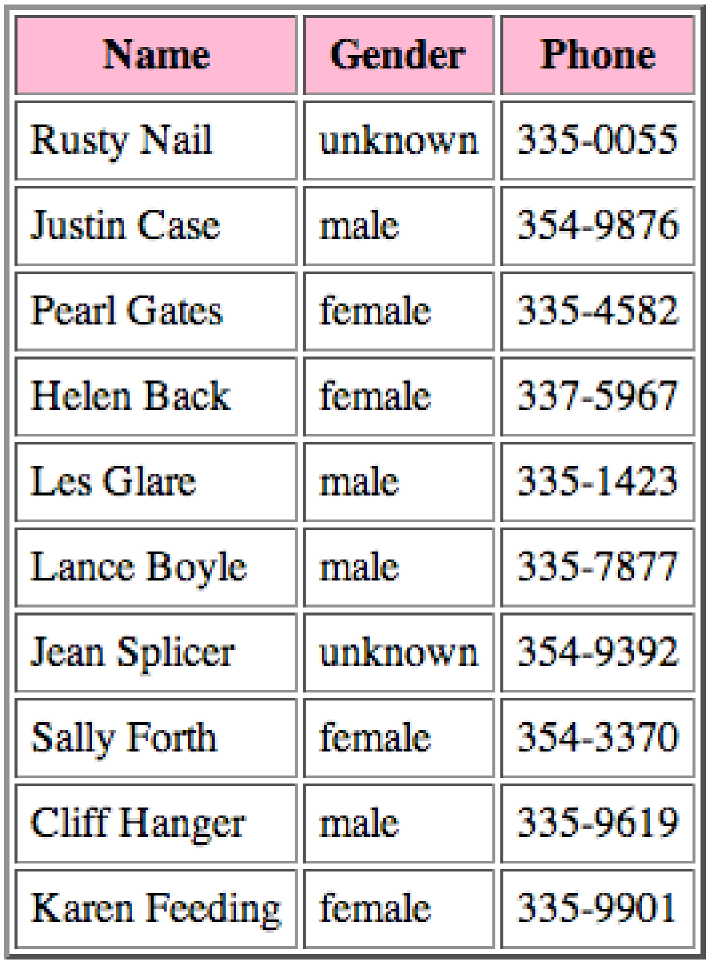
Hiding Information

One advantage of XSLT over CSS is that we can rearrange and even delete information in the XML document when we build and HTML document for a Web browser.

For example, suppose that we do not want to share the names of the cities where the individuals live that is stored in the phone XML document.

All we have to do is omit the fourth *th* element and the fourth *td* element in the stylesheet *html.xsl*.

Table Shown in a Browser



Name	Gender	Phone
Rusty Nail	unknown	335-0055
Justin Case	male	354-9876
Pearl Gates	female	335-4582
Helen Back	female	337-5967
Les Glare	male	335-1423
Lance Boyle	male	335-7877
Jean Splicer	unknown	354-9392
Sally Forth	female	354-3370
Cliff Hanger	male	335-9619
Karen Feeding	female	335-9901

A Question

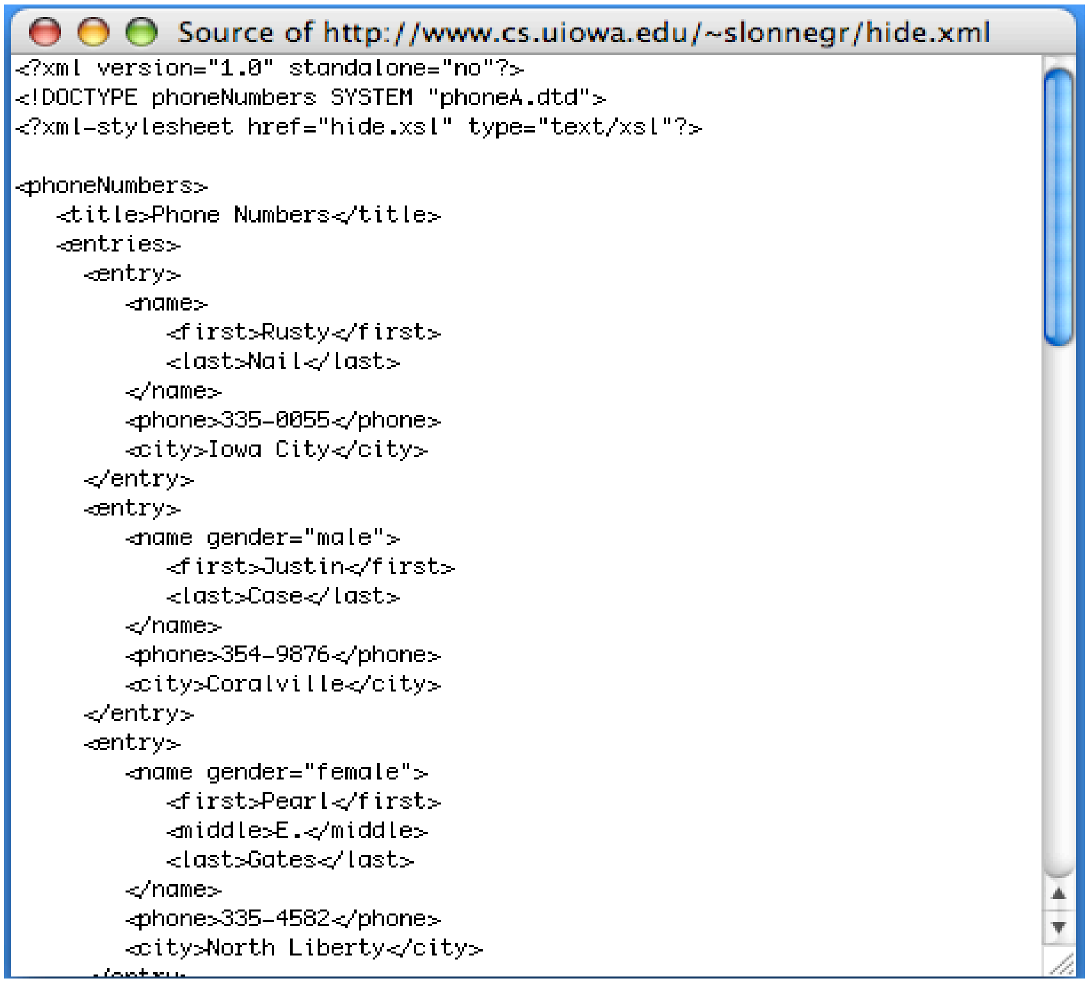
What is wrong with this example?

It is true that the displayed table does not show the names of the cities in the XML document, but does that mean they are really hidden?

Remember that browsers have an option that allows us to see the source of the web page.

In this case, the source is the XML document, which has the city names stored as data.

Viewing the Source


Source of http://www.cs.uiowa.edu/~slonnegr/hide.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM "phoneA.dtd">
<?xml-stylesheet href="hide.xsl" type="text/xsl"?>

<phoneNumbers>
   <title>Phone Numbers</title>
   <entries>
     <entry>
        <name>
           <first>Rusty</first>
           <last>Nail</last>
        </name>
        <phone>335-0055</phone>
        <city>Iowa City</city>
     </entry>
     <entry>
        <name gender="male">
           <first>Justin</first>
           <last>Case</last>
        </name>
        <phone>354-9876</phone>
        <city>Coralville</city>
     </entry>
     <entry>
        <name gender="female">
           <first>Pearl</first>
           <middle>E.</middle>
           <last>Gates</last>
        </name>
        <phone>335-4582</phone>
        <city>North Liberty</city>
```

Browsers that Perform XSLT Correctly

- Mozilla and SeaMonkey on Linux
- Netscape 7.2 on Macintosh
- Foxfire on Macintosh
- Safari on Macintosh
- Internet Explorer on Windows

To find the latest information on browsers and their XML capabilities, try the web site below.

www.xmlsoftware.com/browsers.html

Variation on html.xsl

In this next stylesheet we want the rows of the table to cycle between three colors to enhance the clarity of the information.

As the *entry* elements are processed, we want the table-row tag to indicate three different colors, one after another.

The table-row start tags should cycle between these three possibilities:

```
<tr bgcolor="#cccc99">
<tr bgcolor="#aaaa99">
<tr bgcolor="#aacc99">
```

Tools Required to Solve Problem

- We use the *xsl:choose* element to select one of the three colors depending on the position of the current *entry* element in the node set of all of the *entry* elements formed by

  ```
  <xsl:apply-template select="phoneNumbers/entries/entry">
  ```

- The XPath function *position()* returns the ordinal position, starting at 1, of the current node in its enclosing node set.

- The binary operation *mod* returns the remainder (as with % in Java) of its left operand divided by its right operand.

- The decision making is handled by *xsl:when* element of the form

  ```
  <xsl:when test="position() mod 3 = 0">
  ```

  using the three possible remainders: 0, 1, and 2.

- The next problem is to place the color chosen, say #aacc99, into the attribute value for the attribute *bgcolor* of the *tr* element. The solution is to define an XSLT variable whose value is the chosen color, and then evaluate that variable for the attribute value.

XSLT Variables

A variable is an identifier that can be bound to any XPath expression, but that binding cannot be changed.

So a "variable" in XSLT is not variable.

An XSLT variable is defined using the *xsl:variable* element.

The name of a variable is given by an attribute *name*, and the value of the variable is specified by an attribute *select* or by the content of the *xsl:variable* element.

Note the apostrophes around the string in the first example.

```
<xsl:variable name="home" select="'Iowa City'"/>

<xsl:variable name="space">
      <xsl:text> </xsl:text>
</xsl:variable>
```

To access or reference a variable, place a dollar sign in front of the identifier.

```
<xsl:value-of select="$home"/>
<xsl:value-of select="$space"/>
```

If the reference is in the value of an attribute that is being defined, use an attribute value template.

```
<entry city="{$home}">
```

Scope of Variables

Variables defined at the top-level of the stylesheet are visible everywhere in the stylesheet. These are *global variables*.

Variables defined in an *xsl:template* element are visible from the point of definition to the end of the element. These are *local variables*. They can hide global variables but not other local variables with the same name.

Returning to the original problem, we color the rows of the table by replacing the following three lines with code that defines and accesses a variable that is bound to the color we want.

Old version

```
<xsl:template match="entry">
  <tr>
    <td><xsl:apply-templates select="name"/></td>
```

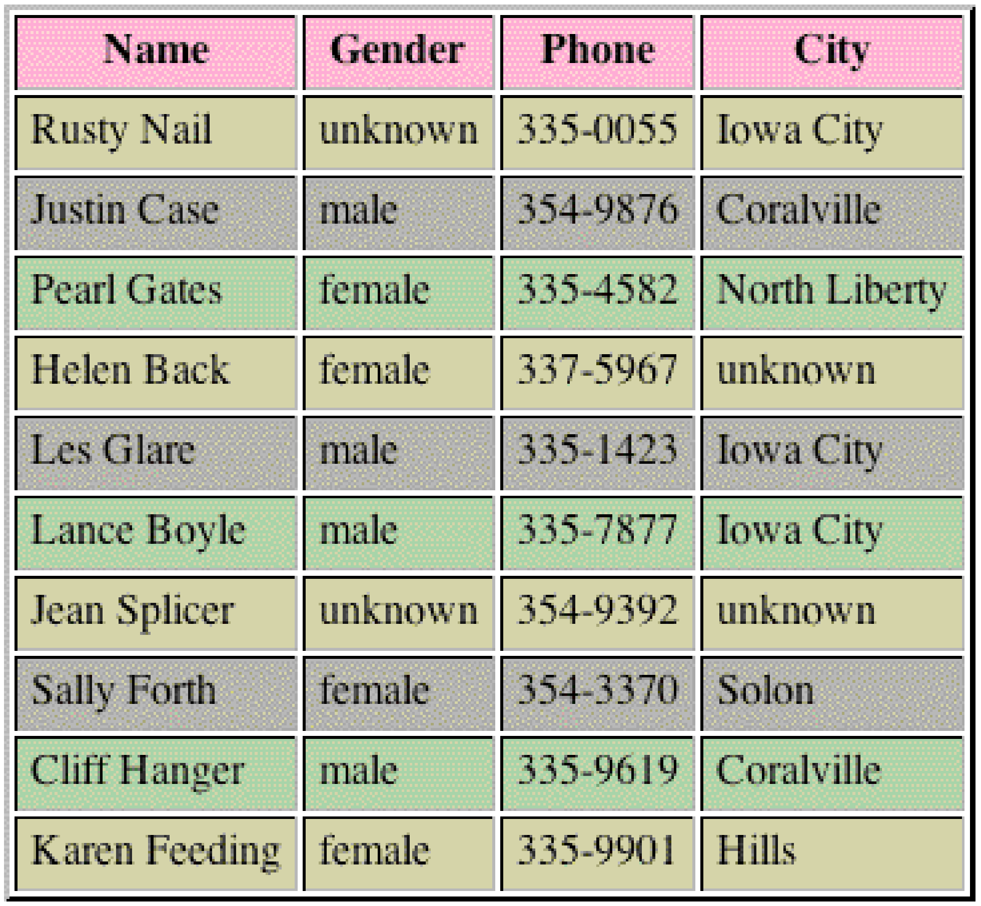
New version

```
<xsl:template match="entry">
  <xsl:variable name="color">
    <xsl:choose>
      <xsl:when test="position() mod 3 = 0">
          #aacc99
      </xsl:when>
```

```
            <xsl:when test="position() mod 3 = 1">
                #cccc99
            </xsl:when>
            <xsl:when test="position() mod 3 = 2">
                #aaaa99
            </xsl:when>
        </xsl:choose>
    </xsl:variable>
    <tr bgcolor="{$color}">
        <td><xsl:apply-templates select="name"/></td>
```

Observe that we calculate the value of the variable *color* using an *xsl:choose* element as the content of the *xsl:variable* element.

Table Shown in a Browser



Name	Gender	Phone	City
Rusty Nail	unknown	335-0055	Iowa City
Justin Case	male	354-9876	Coralville
Pearl Gates	female	335-4582	North Liberty
Helen Back	female	337-5967	unknown
Les Glare	male	335-1423	Iowa City
Lance Boyle	male	335-7877	Iowa City
Jean Splicer	unknown	354-9392	unknown
Sally Forth	female	354-3370	Solon
Cliff Hanger	male	335-9619	Coralville
Karen Feeding	female	335-9901	Hills

XPath Continued

XPath is an expression language.

It has no commands (statements) and no declarations.

Note: XPath 2.0 allows the typing of variables (and parameters).

Uses of XPath expressions

- selecting nodes from source tree

  select attribute in *xsl:apply-templates* and *xsl:for-each*

- specifying matching source nodes conditions

  match attribute in *xsl:template*

- evaluating the value of some expression

  select attribute in *xsl:value-of*, *test* attribute in *xsl:if* and *xsl:when*.

Two basic kinds of expressions

Pattern expressions:	specify a node-set, supporting either matching or evaluations
Non-pattern expressions:	specify any data type, supporting only evaluation

Components of XPath expressions

- A series of location steps to specify a node or a set of nodes in an XSLT tree or a basic value.
- A collection of functions that manipulate node sets, strings, numbers, and boolean values to produce values for various situations.

Location Definitions

A node, a set of nodes, or a value can be specified in the XSLT tree by a sequence of *location steps* separated by / symbols.

The path can be defined from the root of the tree, denoted by starting with / (an absolute path) or can be specified relative to some node that we are currently visiting.

Location paths are read from left to right.

Location Steps

- Primary unit in an XPath.
- A series of location steps defines node, a set of nodes, or a value.
- Syntax: axis :: nodeTest [predicate] [predicate] …

The Context

Each location path is evaluated with respect to a *current context*.

The context consists of six components.

1. The current node in some node set.
2. The size (≥1) of that node set.
3. The position (≥1) of the current node in its node set.
4. The set of variables (and parameters) that are currently in scope.
5. The set of functions available to XPath expressions.
6. The set of namespace declarations currently in scope.

Axis: 13 Axes

The axis specification defines the direction from the current node in which the location step identifies a node set.

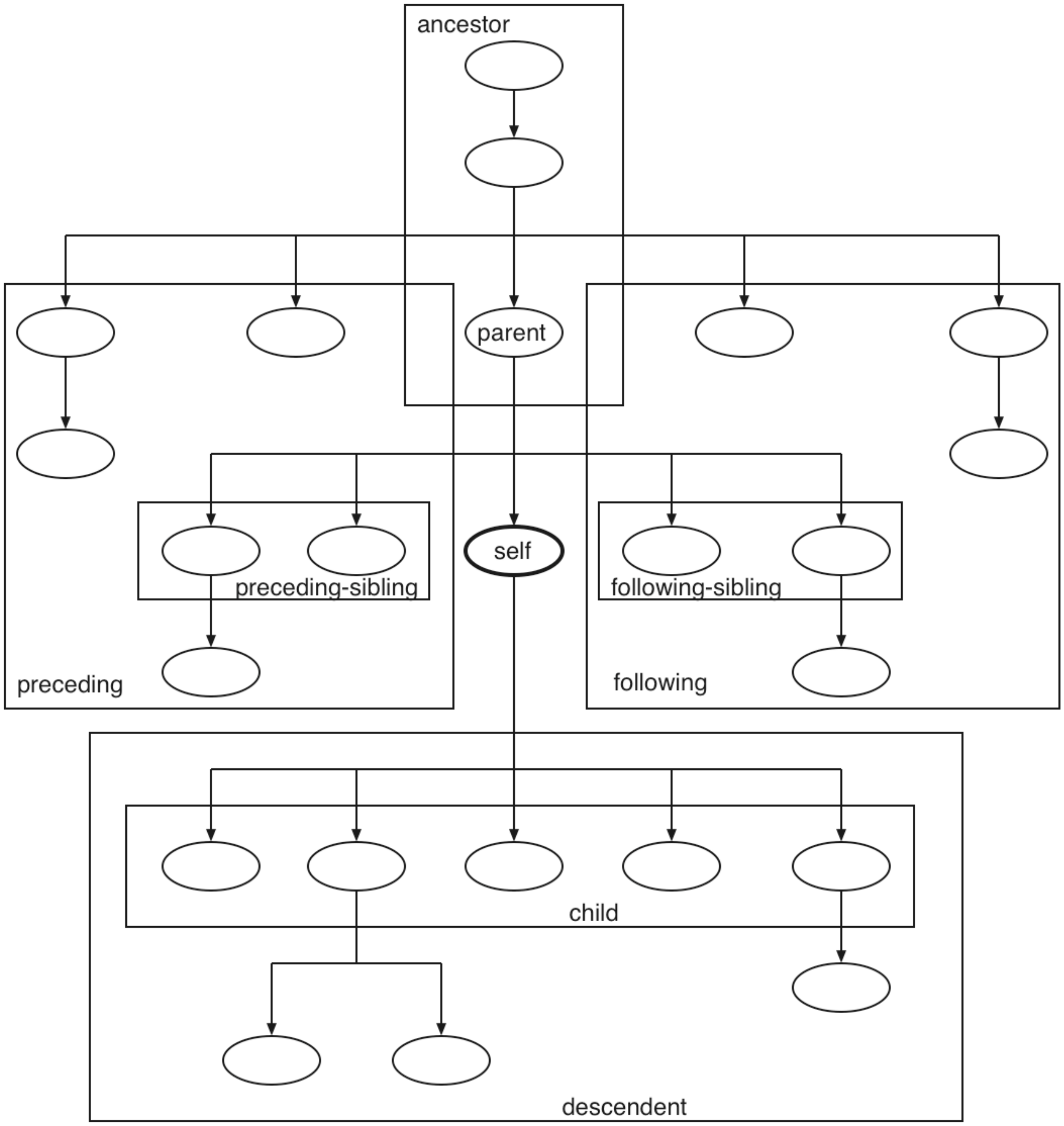
When more than one node is possible in the node set, the nodes are defined in either forward document order or reverse document order.

The table below describes the thirteen axes.

Axis	Order	Relation to Context Node
self	only one	Context node
parent	only one	Parent of context node
child	forward	Children of context node
attribute	not ordered	All attributes of context node
ancestor	reverse	Parent of context node plus all of the parent's ancestors
ancestor-or-self	reverse	Ancestors plus context node
descendent	forward	Children of context node plus their descendants
descendant-or-self	forward	Descendants plus context node
following-sibling	forward	Siblings of the context node appearing after it
preceding-sibling	reverse	Siblings of the context node appearing before it
following	forward	All nodes appearing after the context node in source
preceding	reverse	All nodes appearing before the context node in source
namespace	not applicable	All namespace nodes of the context node

Axis Selection

The context node is shown as a bold oval.



Abbreviations

default	child ::
@	attribute ::
.	self :: node()
..	parent :: node()
//	/descendant-or-self :: node()/

Kinds of Nodes

Nodes in an XSLT tree come in seven varieties.
Some have names and some have values.

Kind	**Name**	**Value**
root	none	concatenation of all descendant text nodes
element nodes	element tag	concatenation of all descendant text nodes
attribute node (attached to an element)	attribute name	attribute value
text node	none	text in node
comment node	none	contents of comment
processing instruction node	PI name	PI value
namespace node	prefix	namespace URI

Node Test

The node test is a specification of nodes of some sort using the name of the node, a function that specifies a kind of node, or an abbreviation.

/	the root
name	nodes with the given name
node()	any nodes at all
*	any element nodes
prefix : *	any elements from a namespace
prefix : *name*	nodes with namespace and name
@*	all attribute nodes for an element
text()	text nodes
comment()	comment nodes
processing-instruction()	processing instruction nodes
processing-instructions(*'target'*)	

Predicate Tests

A predicate is filter that removes unwanted nodes by means of a boolean expression.

Only those nodes for which the predicate is true are retained.

Node Set Tests

An empty node set is evaluated as *false*.

Some functions are used to test a node in a node set.

number *position*()	Returns the index (≥1) of the context node in the context node set
number *last*()	Returns size of the context node set
string *name*(node-set?)	Returns Qname of the context node

Examples

Suppose the context node is the *entries* element in *phoneA.xml*.

entry[position()=2]	Returns the second entry element
entry[2]	An abbreviation of the previous XPath
entry[last()]	Returns the last entry element
entry[position()<last()]	Returns the node set of all *entry* elements except the last one.
entry[position()!=1]	Returns the node set of all *entry* elements except the first one.
descendant::*[name()='first']	Returns all descendant elements whose Qname is *first*
entry[name/@gender='male']	Returns all *entry* elements whose *name* child has a *gender* attribute equal to 'male'

Boolean Operators

XPath has infix operators *and* and *or* that serve as boolean operations.

entry[position()=1 or position()=last()]	Returns the first and last *entry* elements.
entry[position()=2 and name/@gender='male']	Returns the second *entry* element if its *gender* is 'male', or an empty set

Equivalent expression:

entry[position()=2][name/@gender='male']

Union Operator

The union operator (|) can be used to combine node tests or predicates.

The following XPath expressions produce the same node set.

```
entry[2] | entry[4]

entry[position()=2 or position()=4]

entry[ 2 | 4 ]
```

The union operator has different interpretations depending on whether it is selecting nodes or matching nodes.

match="entry[1] | entry[last()]"	Matches the first or the last *entry* elements
select="entry[1] | entry[last()]"	Selects the first and the last *entry* elements

Boolean Functions

boolean *boolean*(expr)	Converts an XPath expression to a boolean value
boolean *true*()	Returns the value *true*
boolean *false*()	Returns the value *false*
boolean *not*(boolean)	Returns logical *not* of expression

Number Operations

Binary (infix): +, –, *, div, mod

Unary: – (Put a space before each unary minus to avoid confusion.)

Relational: <, <=, >, >=

The operands for these operations are coerced to numbers if possible. Otherwise, the value NaN (not a number) is produced.

Number literals can be written as integers or real numbers with a decimal point. No scientific notation (*e* for exponent) is allowed.

Equality relations, = and !=, can be used for numbers as well as the other types (boolean, string, and node set) in XPath.

The equality rules for node sets are somewhat bizarre.

Number Functions

number *number*(expr?)	Converts *expr* to a number
number *floor*(number)	Returns largest integer ≤ parameter
number *ceiling*(number)	Returns smallest integer ≥ parameter
number *round*(number)	Rounds to nearest integer
number *sum*(node-set)	Converts text in each node to a number and returns sum of them
number *count*(node-set)	Returns the number of nodes in set

Strings

String literals can be delimited using quote symbols or apostrophes.

String Functions

string *string*(expr?)	Converts *expr* to a string
string *concat*(string, string, string*)	Returns the concatenation of all the string parameters
string *substring*(string, number, number?)	Returns the substring starting in position given by second parameter; third parameter gives the length
string *substring-after*(string, string)	Returns the substring of the first parameter that occurs after the first occurrence of second parameter
string *substring-before*(string, string)	Returns the substring of the first parameter that occurs before the first occurrence of second parameter
boolean *contains*(string, string)	Returns *true* if the first parameter contains the second parameter
boolean *starts-with*(string, string)	Returns *true* if the first parameter starts with the second parameter
number *string-length*(string?)	Returns the length of the string

string *translate*(string, string, string)

Returns a copy of the first parameter with each character in the second parameter replaced by the corresponding character in the third parameter

string *normalize-space*(string?)

Returns the string with its space normalized (interior white space becomes one space and trim)

Conversions and Coercions

To From	**boolean**	**number**	**string**
boolean		false ⇒ 0 true ⇒ 1	false ⇒ 'false' true ⇒ 'true'
number	0 ⇒ false NaN ⇒ false other ⇒ true		convert to decimal format
string	empty ⇒ false other ⇒ true	parse as a decimal number	
node set	empty ⇒ false other ⇒ true	converted via string	string value of first node in document order

Some Examples

To illustrate the use of XPath location steps and functions, we solve several problems that involve extracting data and making calculations on a small "database" of grades stored in an XML document.

The information used in the examples will be found in the following XML document produced by a grading program written in Java.

File: roster.xml

```
<?xml version="1.0"?>
<!DOCTYPE roster SYSTEM "roster.dtd">
<roster>
  <rosterName>roster</rosterName>
 <maximums>
   <quizzes>
     <quiz>20</quiz>
     <quiz>30</quiz>
   </quizzes>
   <projects>
     <project>50</project>
     <project>60</project>
   </projects>
   <exams>
     <exam>100</exam>
     <exam>100</exam>
     <exam>100</exam>
   </exams>
 </maximums>
  <students>
   <student id="94811">
     <name>Rusty Nail</name>
     <quizzes>
       <quiz>16</quiz>
       <quiz>12</quiz>
  </quizzes>
  <projects>
    <project>44</project>
    <project>52</project>
  </projects>
  <exams>
    <exam>77</exam>
    <exam>68</exam>
    <exam>49 </exam>
  </exams>
</student>

<student id="2562">
  <name>Guy Wire</name>
  <quizzes>
    <quiz>15</quiz>
    <quiz>23</quiz>
  </quizzes>
  <projects>
    <project>33</project>
    <project>47</project>
  </projects>
  <exams>
    <exam>78</exam>
    <exam>86</exam>
    <exam>88</exam>
  </exams>
</student>

<student id="132987">
  <name>Norman Conquest</name>
  <quizzes>
    <quiz>18</quiz>
    <quiz>26</quiz>
  </quizzes>
  <projects>
    <project>46</project>
    <project>50</project>
  </projects>
  <exams>
    <exam>99</exam>
    <exam>79</exam>
```

```
      <exam>78</exam>
    </exams>
  </student>

  <student id="49194">
    <name>Eileen Dover</name>
    <quizzes>
      <quiz>20</quiz>
      <quiz>19</quiz>
    </quizzes>
    <projects>
      <project>39</project>
      <project>46</project>
   </projects>
    <exams>
      <exam>90</exam>
      <exam>79</exam>
      <exam>89</exam>
    </exams>
  </student>

  <student id="137745">
    <name>Barb Wire</name>
    <quizzes>
      <quiz>20</quiz>
      <quiz>25</quiz>
    </quizzes>
    <projects>
      <project>48</project>
      <project>60</project>
    </projects>
    <exams>
      <exam>38</exam>
      <exam>48</exam>
      <exam>66</exam>
    </exams>
  </student>
 </students>
</roster>
```

File: roster.dtd

```
<!ELEMENT roster (rosterName, maximums, students)>
<!ELEMENT rosterName (#PCDATA)>
<!ELEMENT maximums (quizzes, projects, exams)>
<!ELEMENT students (student*)>
<!ELEMENT student (name, quizzes, projects, exams, total?)>
<!ATTLIST student id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT quizzes (quiz*)>
<!ELEMENT quiz (#PCDATA)>
<!ELEMENT projects (project*)>
<!ELEMENT project (#PCDATA)>
<!ELEMENT exams (exam*)>
<!ELEMENT exam (#PCDATA)>
<!ELEMENT total (#PCDATA)>
```

Problem 1

Find the average for the first exam in the grade book.

This requires finding the sum of the scores and how many there are.

In the stylesheet we use variables for these values to reduce the complexity of the XPath expressions.

File: mean.xsl

```
<?xml version="1.0"?> <!-- Find the sum of all first exams, -->
<!-- mean.xsl -->         <!-- the number of first exams, and -->
                          <!-- the average on the first exam. -->

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

  <xsl:template match="/">
    <xsl:variable name="sum1"
      select="sum(roster/students/student/exams/exam[1])"/>
    <xsl:variable name="count1"
      select="count(roster/students/student/exams/exam[1])"/>
    Sum for exam 1 = <xsl:value-of select="$sum1"/>
    Number of exams = <xsl:value-of select="$count1"/>
    Average on exam 1 =
              <xsl:value-of select="$sum1 div $count1"/>
  </xsl:template>
</xsl:stylesheet>
```

Applying mean.xsl to roster.xml

```
Sum for exam 1 = 382
Number of exams = 5
Average on exam 1 = 76.4
```

Remember to use *div* for division and not /.

Problem 2

In this example we want to find the average scores on the quizzes for each of the students.

The stylesheet uses the element *xsl:for-each* to iterate through the *student* elements.

For each student we find the sum of the quiz scores and how many there are using variables again.

To enhance the output, we delimit the student information with a row of equal signs that is defined as a global variable.

File: means.xsl

```
<?xml version="1.0"?>   <!-- Find sum, count, and average -->
<!-- means.xsl -->          <!-- for quizzes for all students.     -->
<xsl:stylesheet  version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:variable name="delimiter">
    <xsl:text>
========================================
    </xsl:text>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:value-of select="$delimiter"/>
     <xsl:for-each select="roster/students/student">
      <xsl:variable name="quizzes"
                          select="sum(quizzes/quiz)"/>
      <xsl:variable name="count"
                          select="count(quizzes/quiz)"/>
```

```
         <xsl:value-of select="name"/>
          Sum: <xsl:value-of select="$quizzes"/>
          Count: <xsl:value-of select="$count"/>
          Mean: <xsl:value-of select="$quizzes div $count"/>
         <xsl:value-of select="$delimiter"/>
      </xsl:for-each>
    </xsl:template>
  </xsl:stylesheet>
```

Applying means.xsl to roster.xml

```
=============================================
    Rusty Nail
      Sum: 28
      Count: 2
      Mean: 14
=============================================
    Guy Wire
      Sum: 38
      Count: 2
      Mean: 19
=============================================
    Norman Conquest
      Sum: 44
      Count: 2
      Mean: 22
=============================================
    Eileen Dover
      Sum: 39
      Count: 2
      Mean: 19.5
=============================================
    Barb Wire
      Sum: 45
      Count: 2
      Mean: 22.5
=============================================
```

Problem 3

Now we want to find out whether any exam scores fall below 50 in the entire grade book.

If such scores exist, we want to know how many.

We use an XPath expression with the function *count* to compute this number in one step.

Finally, we use *xsl:if* elements to control the output.

File: fifty.xsl

```
<?xml version="1.0"?>   <!-- Find out if there are exam -->
<!-- fifty.xsl -->              <!-- scores less than 50. -->
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="text"/>
   <xsl:template match="/">
     <xsl:variable name="num" select=
       "count(roster/students/student/exams/exam[.&lt;50])"/>
     <xsl:if test="$num > 0">
       Number of scores below 50:
                     <xsl:value-of select="$num"/>
     </xsl:if>
     <xsl:if test="$num = 0">
        No student scored below 50 on an exam.
     </xsl:if>
   </xsl:template>
</xsl:stylesheet>
```

Applying fifty.xsl to roster.xml

```
            Number of scores below 50: 3
```

Problem 4

We want to find the names of all students who scored higher on exam 1 than on exam 3.

In the stylesheet we use *xsl:apply-templates* to visit each of the *student* elements in the source tree.

A template that matches these *student* elements selects the scores for the two exams so they can be compared using *xsl:if*.

File: higher.xsl

```
<?xml version="1.0"?>  <!-- Find the names of all students -->
<!-- higher.xsl -->         <!-- who scored higher on exam 1 -->
                           <!-- than on exam 3. -->
<xsl:stylesheet  version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
     Students who scored higher on exam 1 than on exam 3:
      <xsl:apply-templates select="roster/students/student"/>
  </xsl:template>

  <xsl:template match="student">
     <xsl:variable name="exam1" select="exams/exam[1]"/>
     <xsl:variable name="exam3" select="exams/exam[3]"/>
     <xsl:if test="$exam1 > $exam3">
         Student: <xsl:value-of select="name"/>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Applying higher.xsl to roster.xml

```
Students who scored higher on exam 1 than on exam 3:
    Student: Rusty Nail
    Student: Norman Conquest
    Student: Eileen Dover
```

Problem 5

Find all students who scored 80 or higher on exam 3.

In the first solution to this problem, we use an XPath expression in the *xsl:apply-templates* element to choose those *exam* elements that satisfy the criteria.

Then a template matches those *exam* elements and finds the name of the student from that point using another XPath expression.

File: ge80.xsl

```
<?xml version="1.0"?>   <!-- Find all students who scored -->
<!-- ge80.xsl -->         <!-- 80 or higher on exam 3.  -->
<xsl:stylesheet  version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    Students who scored 80 or higher on exam 3:
    <xsl:apply-templates  select=
          "roster/students/student/exams/exam[3][.>=80]"/>
  </xsl:template>

  <xsl:template match="exam">
    Student <xsl:value-of select="ancestor::student/name"/>
    scored <xsl:value-of select="."/> on exam 3.
  </xsl:template>
</xsl:stylesheet>
```

Applying ge80.xsl to roster.xml

```
        Students who scored 80 or higher on exam 3:

        Student Guy Wire
        scored 88 on exam 3.

        Student Eileen Dover
        scored 89 on exam 3.
```

In a second solution to this problem, we use an *xsl:for-each* element to iterate through the set of *exam* elements that satisfy the criteria given in the problem.

File: gef.xsl

```
<?xml version="1.0"?>   <!-- Find all students who scored -->
<!-- gef.xsl -->        <!-- 80 or higher on exam 3.  -->
<xsl:stylesheet  version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    Students who scored 80 or higher on exam 3:
    <xsl:for-each
     select="roster/students/student/exams/exam[3][.>=80]">
       Student <xsl:value-of  select="../../name"/>
       scored <xsl:value-of  select="."/> on exam 3.
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The output from using this stylesheet on *roster.xml* is identical to that of the former solution.

Problem 6

Find all students who scored 80 or higher on any exams.

The first solution uses nested *xsl:for-each* elements to choose the *student* elements and then the *exam* elements that satisfy the condition.

File: geall.xsl

```
<?xml version="1.0"?> <!-- Find all students who scored -->
<!-- geall.xsl -->           <!-- 80 or higher on any exam.    -->
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    Students who scored 80 or higher on any exam:
     <xsl:for-each select="roster/students/student">
       <xsl:for-each select="exams/exam">
      <xsl:if test=".>=80">
        Student <xsl:value-of  select="../../name"/>
        scored <xsl:value-of  select="."/>
        on exam <xsl:value-of select="position()"/>.
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Applying geall.xsl to roster.xml

```
Students who scored 80 or higher on any exam:
Student Guy Wire
scored 86
on exam 2.
Student Guy Wire
scored 88
on exam 3.
```

```
Student Norman Conquest
scored 99
on exam 1.
Student Eileen Dover
scored 90
on exam 1.
Student Eileen Dover
scored 89
on exam 3.
```

In the second solution to Problem 6, the nested *xsl:for-each* elements are replaced by *xsl:template* elements that are invoked by *xsl:apply-templates* elements.

File: geat.xsl

```
<?xml version="1.0"?> <!-- Find all students who scored -->
<!-- geat.xsl -->           <!-- 80 or higher on any exam.    -->
<xsl:stylesheet  version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    Students who scored 80 or higher on any exam:
     <xsl:apply-templates select="roster/students/student"/>
  </xsl:template>

  <xsl:template match="student">
    <xsl:apply-templates select="exams/exam"/>
  </xsl:template>

  <xsl:template match="exam">
     <xsl:if test=".>=80">
        Student <xsl:value-of  select="../../name"/>
        scored <xsl:value-of  select="."/>
        on exam <xsl:value-of select="position()"/>.
      </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

One way to take more control of the output format is to use the string concatenation function to assemble a line of text.

In the previous example, add one new template and replace the template that matches *exam* elements.

Excerpts from File: geconcat.xsl

```
<xsl:variable name="newline">
  <xsl:text>
  </xsl:text>
</xsl:variable>

<xsl:template match="exam">
 <xsl:if test=".&gt;=80">
    <xsl:variable name="result"
                  select="concat('Student ', ../../name,
                                 ' scored ', ., ' on exam ',
                                 position(), '.')"/>
    <xsl:value-of select="$result"/>
    <xsl:value-of select="$newline"/>
 </xsl:if>
</xsl:template>
```

Applying geconcat.xsl to roster.xml

```
Students who scored 80 or higher on any exam:

Student Guy Wire scored 86 on exam 1.
Student Guy Wire scored 88 on exam 2.
Student Norman Conquest scored 99 on exam 1.
Student Eileen Dover scored 90 on exam 1.
Student Eileen Dover scored 89 on exam 3.
```

Sorting

Elements of a node set can be sorted according various criteria using the XSLT element *xsl:sort*.

This element must be a child of one of the two XSLT elements that define node sets, *xsl:for-each* or *xsl:apply-templates*.

The *xsl:sort* element has several attributes that can be used to control properties of the sort to be performed.

Attribute	Possible Values	Default
select	XPath expression	"."
order	"ascending" | "descending"	"ascending"
data-type	"text" | "number"	"text"
case-order	"lower-first" | "upper-first"	"lower-first"

The XPath expression for the *select* attribute specifies the values on which the sort will work.

The next problem will illustrate various features of sorting.

Problem 7

Show the totals on the projects for each of the students in the grade book. List students by name in the first solution, and later by ID number.

File: npro.xsl

```
<?xml version="1.0"?>  <!-- Show total on projects for each -->
<!-- npro.xsl -->            <!-- student.List students by name. -->
<xsl:stylesheet  version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:output method="text"/>

  <xsl:variable name="delimiter">
    <xsl:text>
========================================</xsl:text>
  </xsl:variable>

  <xsl:template match="/">
    <xsl:value-of select="$delimiter"/>
     <xsl:for-each select="roster/students/student">
       <!-- Note this spot in the template -->
        Student name: <xsl:value-of select="name"/>
       Project total: <xsl:value-of
                           select="sum(projects/project)"/>
       <xsl:value-of select="$delimiter"/>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Applying npro.xsl to roster.xml

```
========================================
        Student name: Rusty Nail
        Project total: 96
========================================
        Student name: Guy Wire
        Project total: 80
========================================
        Student name: Norman Conquest
        Project total: 96
========================================
        Student name: Eileen Dover
        Project total: 85
========================================
        Student name: Barb Wire
        Project total: 108
========================================
```

The literal text "Student name: " provides a return character between the delimiters and the next part of the result document.
So the delimiter variable has no return character at its end.

To sort the list of names in the result document, simply replace the comment (Note this spot ...) in the stylesheet above with the following element a new file.

```
            <xsl:sort select="name"/>
```

Applying snpro.xsl to roster.xml

```
    ========================================
            Student name: Barb Wire
            Project total: 108
    ========================================
            Student name: Eileen Dover
            Project total: 85
    ========================================
            Student name: Guy Wire
            Project total: 80
    ========================================
            Student name: Norman Conquest
            Project total: 96
    ========================================
            Student name: Rusty Nail
            Project total: 96
    ========================================
```

The sorting is by first name since the *name* elements are sorted as they were entered.

To sort by last name, replace the comment by this element.

```
    <xsl:sort select="substring-after(name, ' ')"/>
```

Applying slnpro.xsl to roster.xml

```
==========================================
        Student name: Norman Conquest
        Project total: 96
==========================================
        Student name: Eileen Dover
        Project total: 85
==========================================
        Student name: Rusty Nail
        Project total: 96
==========================================
        Student name: Guy Wire
        Project total: 80
==========================================
        Student name: Barb Wire
        Project total: 108
==========================================
```

Although the sorting is now by last name, the two students with last name Wire probably should be sorted by first name.

XSLT allows multiple sorting conditions, which are entered in the order of their priority.

Now replace the comment by two *xsl:sort* elements.

```
        <xsl:sort select="substring-after(name,' ')"/>
        <xsl:sort select="substring-before(name,' ')"/>
```

Applying slfnpro.xsl to roster.xml

```
=========================================
          Student name: Norman Conquest
          Project total: 96
=========================================
          Student name: Eileen Dover
          Project total: 85
=========================================
          Student name: Rusty Nail
          Project total: 96
=========================================
          Student name: Barb Wire
          Project total: 108
=========================================
          Student name: Guy Wire
          Project total: 80
=========================================
```

Next alter the orginal stylesheet so that ID numbers are printed and supply an *xsl:sort* element to arrange the output sorted by these ID numbers.

Here is the only template that needs changing.

```
<xsl:template match="/">
    <xsl:value-of select="$delimiter"/>
     <xsl:for-each select="roster/students/student">
        <xsl:sort select="@id"/>
        Student ID: <xsl:value-of select="@id"/>
        Project total: <xsl:value-of
                          select="sum(projects/project)"/>
        <xsl:value-of select="$delimiter"/>
    </xsl:for-each>
  </xsl:template>
```

Applying sipro.xsl to roster.xml

```
========================================
        Student ID: 132987
        Project total: 96
========================================
        Student ID: 137745
        Project total: 108
========================================
        Student ID: 2562
        Project total: 80
========================================
        Student ID: 49194
        Project total: 85
========================================
        Student ID: 94811
        Project total: 96
========================================
```

Observe that the ID numbers are not in numerical order.

The order produced by *xsl:sort* is an alphabetic ordering by default.

We need to alter the *order* attribute to get numerical ordering of the ID numbers.

```
        <xsl:sort select="@id" data-type="number"/>
```

Applying nsipro.xsl to roster.xml

```
==========================================
        Student ID: 2562
        Project total: 80
==========================================
        Student ID: 49194
        Project total: 85
==========================================
        Student ID: 94811
        Project total: 96
==========================================
        Student ID: 132987
        Project total: 96
==========================================
        Student ID: 137745
        Project total: 108
==========================================
```

Named Templates

The *xsl:template* element may have a *name* attribute instead of the *match* attribute.

This kind of template must be called explicitly using the element *xsl:call-template*, which requires a *name* attribute to specify which template to invoke.

Unlike *xsl:apply-templates*, the *xsl:call-template* element does not change the current node or the current node set when it instantiates a new template.

Named templates correspond to method definitions in other programming languages.

Actually, the XSLT methods are functions since the content of the named template defines a "return value" produced by the template.

The *xsl:call-template* element then acts as a method call.

Example: A Delimiter Line

Method Definition

```
      <xsl:template name="delimiter">
         <xsl:text>
    ==================================</xsl:text>
       </xsl:template>
```

Method Call

```
         <xsl:call-template name="delimiter"/>
```

As we saw before, this behavior can be handled by defining the delimiter line as the value of a variable just as easily.

The full power of methods derives from the use of formal and actual parameters so that separate calls of the method can execute in different settings.

Formal Parameters

Formal parameters for a named template are indicated by a sequence of *xsl:param* elements as the first items in the content of the *xsl:template* element.

An *xsl:param* element allows two attributes.

Attribute	Purpose of Value	Default Value
name	Name of parameter	Required
select	Default value	""

Actual Parameters

Actual parameters are specified in an *xsl:call-template* element using the *xsl:with-param* element.

An *xsl:with-param* element allows two attributes.

Attribute	Purpose of Value	Default Value
name	Name of parameter	Required
select	Actual value	""

Example: Find Average of Numbers in a Node Set

Method Definition

```
<xsl:template name="mean">
  <xsl:param name="nodeset"/>
  <xsl:value-of select=
                "sum($nodeset) div count($nodeset)"/>
</xsl:template>
```

Method Call (current context = /roster/students/student)

```
<xsl:call-template name="mean">
   <xsl:with-param name="nodeset"
                        select="exams/exam"/>
</xsl:call-template>
```

Problem 8

Find the average scores on the exams for each of the students in the XML document *roster.xml*.

The solution uses the two methods (named templates) given in the examples above.

File: average.xsl

```
<?xml version="1.0"?>      <!-- Find the averages on exams -->
<!-- average.xsl -->         <!-- for each of the students. -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
     <xsl:apply-templates select="roster/students/student"/>
  </xsl:template>

  <xsl:template match="student">
    Student: <xsl:value-of select="name"/>
    Exam average: <xsl:call-template name="mean">
       <xsl:with-param name="nset"
                          select="exams/exam"/>
     </xsl:call-template>
     <xsl:call-template name="delimiter"/>
  </xsl:template>

  <xsl:template name="delimiter">
    <xsl:text>
==================================</xsl:text>
  </xsl:template>

  <xsl:template name="mean">
    <xsl:param name="nset"/>
    <xsl:value-of select="sum($nset) div count($nset)"/>
  </xsl:template>
</xsl:stylesheet>
```

Applying average.xsl to roster.xml

```
      Student: Rusty Nail
      Exam average: 64.66666666666667
   =================================
      Student: Guy Wire
      Exam average: 84
   =================================
      Student: Norman Conquest
      Exam average: 85.33333333333333
   =================================
      Student: Eileen Dover
      Exam average: 86
   =================================
      Student: Barb Wire
      Exam average: 50.666666666666664
   =================================
```

Repetition in Methods

The power of methods or any algorithms comes from the ability to repeat computations in a loop.

Since XSLT variables cannot be altered, the loops of an imperative programming language cannot be implemented because we cannot change the state of variables that might control a while-type loop.

As with functional programming languages, repetition is achieved using recursion.

The changing state that controls the recursion will be embodied in the parameters to the recursive methods.

Example: Simulate a for Loop

In this stylesheet we create a recursive method using a named template to simulate the behavior of the for loop shown below.

for (**int** k=start; k<=end; k=k+inc)

To simplify the problem, we assume that the *inc* value is always positive.

The named template will have three parameters to hold the *start* value, the *end* value, and the *inc* value.

Successive calls of the method will see the *start* value climb until it surpasses the *end* value.

File: for.xsl

```
<?xml version="1.0"?>  <!-- for (int k=start; k<=end; k=k+inc) -->
<xsl:stylesheet  version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- for.xsl -->
  <xsl:output method="text"/>

  <xsl:template name="for">             <!-- method definition -->
    <xsl:param name="start"/>
    <xsl:param name="end"/>
    <xsl:param name="inc" select="1"/>
    <xsl:if test="$start &lt;= $end">
      Perform task for k = <xsl:value-of select="$start"/>
      <xsl:call-template name="for">
        <xsl:with-param name="start" select="$start + $inc"/>
        <xsl:with-param name="end" select="$end"/>
        <xsl:with-param name="inc" select="$inc"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:call-template name="for">           <!-- method call -->
      <xsl:with-param name="start" select="1"/>
      <xsl:with-param name="end" select="18"/>
      <xsl:with-param name="inc" select="2"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

Scoping Issues

Observe that a variable or parameter defined in an outer scope is visible inside the *select* attribute of an *xsl:with-param* element.

```
<xsl:with-param name="inc" select="$inc"/>
```

The first occurrence of *inc* is the name of the formal parameter that is defined in the named template.

The second occurrence of *inc* refers to the value of the parameter defined at the top of the named template.

Applying the Stylesheet

Note that the only XPath location defined in this stylesheet is the reference to the root (/).

We have no need for any information from the XML document that will be used in the transformation.

Therefore we use an XML document with no content at all.

File: empty.xml

```
<?xml version="1.0"?>
<!-- empty.xml -->
<empty/>
```

Applying for.xsl to empty.xml

```
Perform task for k = 1
Perform task for k = 3
Perform task for k = 5
Perform task for k = 7
Perform task for k = 9
Perform task for k = 11
Perform task for k = 13
Perform task for k = 15
Perform task for k = 17
```

Global Parameters

Parameters may be specified at the top level of an XSLT stylesheet.

These parameters need to be supplied by the application software that implements the XSLT processor.

In the next example we define a method that takes three parameters representing a month, a day, and a year and returns the day of the week for that day.

We will supply these three values from outside of the stylesheet using global paramters.

Top-level elements will define the three global parameter and indicate default values in case the values are not supplied when the XSLT processor is invoked.

```
<xsl:param name="pm" select="9"/>
<xsl:param name="pd" select="11"/>
<xsl:param name="py" select="2001"/>
```

Specifying Global Parameters

We have considered three different XSLT processors: *xsltproc*, Saxon, and the processor called from Java.

Each has its own way of specifying these global parameters.

xsltproc

This command-line instruction needs to be on only one line.

```
% xsltproc --param pm 12 --param pd 7
                    --param py 1941 dow.xsl empty.xml
Sunday
```

The xsltproc command allows an option "--stringparam pd 7" as well. With this option, the actual parameter may be entered inside of quote symbols.

Saxon

Enter this command on a single line.

```
% java net.sf.saxon.Transform empty.xml
                         dow.xsl pm=7 pd=4 py=1776
Warning: Running an XSLT 1.0 stylesheet with an XSLT
2.0 processor
Thursday
```

Java

With Java we use an instance method for the Transformer object to set the parameters.

```
Transformer tran = factory.newTransformer(xslStream);
tran.setParameter("pm", "5");
tran.setParameter("pd", "11");
tran.setParameter("py", "2005");
```

Problem: Day of the Week

Now we turn to writing a method to solve the problem.

We are given three values, *m* for the month, *d* for the day, and *y* for the year.

Our solution will follow the steps of the Schillo Algorithm shown below.

a) If *m>2*, let *m* be *m-2*; otherwise, let *m* be *m+10* and *y* be *y-1*.

b) Find the century *cent* of the year by dividing *y* by 100.

c) Find the position *annum* (0≤annum≤99) of the year in the century.

d) Let *base* be the quantity *(13•m-1)/5 + annum/4 + cent/4*.

e) Let *offset* be the remainder on dividing *(base + annum + d - 2•cent)* by 7. If that remainder is negative, add 7 to it.

f) Now *offset* indicates the day of the week with 0 for Sunday, 1 for Monday, and so on to 6 for Saturday.
Return the appropriate String.

Observe that this algorithm is defined in the imperative style.

Variables are modified as the algorithm proceeds.

In XSLT we need to use new variables to record the changes described in the algorithm.

The stylesheet below has three important units at the top level:

- definition of the global parameters,
- call of the function *dow* with the three parameters
- definition of the function *dow*.

File: dow.xsl

```
<?xml version="1.0"?> <!-- Calculate the day of the week -->
<!-- dow.xsl -->            <!-- from the numbers m, d, y -->
<xsl:stylesheet  version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:param name="pm" select="9"/>      <!-- global  -->
  <xsl:param name="pd" select="11"/>     <!-- parameters -->
  <xsl:param name="py" select="2001"/>

  <xsl:template match="/">              <!-- method call -->
    <xsl:call-template name="dow">
      <xsl:with-param name="month" select="$pm"/>
      <xsl:with-param name="day" select="$pd"/>
      <xsl:with-param name="year" select="$py"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="dow">            <!-- method definition -->
    <xsl:param name="month"/>
    <xsl:param name="day"/>
    <xsl:param name="year"/>

    <xsl:variable name="m">
      <xsl:if test="$month &gt; 2">
        <xsl:value-of select="$month - 2"/>
      </xsl:if>
      <xsl:if test="$month &lt;= 2">
        <xsl:value-of select="$month + 10"/>
      </xsl:if>
    </xsl:variable>
```

```
<xsl:variable name="y">
  <xsl:if test="$month &gt; 2">
    <xsl:value-of select="$year"/>
  </xsl:if>
  <xsl:if test="$month &lt;= 2">
    <xsl:value-of select="$year - 1"/>
  </xsl:if>
</xsl:variable>

<xsl:variable name="ct" select="floor($y div 100)"/>

<xsl:variable name="an" select="$y mod 100"/>

<xsl:variable name="base" select=
"floor((13 * $m – 1) div 5) + floor($an div 4) + floor($ct div 4)"/>

<xsl:variable name="rem" select=
                    "($base + $an + $day – 2 * $ct) mod 7"/>

<xsl:variable name="offset">
  <xsl:choose>
    <xsl:when test="$rem &lt; 0">
      <xsl:value-of select="$rem + 7"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$rem"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>

<xsl:choose>
  <xsl:when test="$offset = 0">Sunday</xsl:when>
  <xsl:when test="$offset = 1">Monday</xsl:when>
  <xsl:when test="$offset = 2">Tuesday</xsl:when>
  <xsl:when test="$offset = 3">Wednesday</xsl:when>
  <xsl:when test="$offset = 4">Thursday</xsl:when>
```

```
      <xsl:when test="$offset = 5">Friday</xsl:when>
      <xsl:when test="$offset = 6">Saturday</xsl:when>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Global Parameters in Java

We can alter the Java program MyTransform.java so that it will recognize global parameters as command-line arguments and pass them along to the XSLT processor.

File: XsltParams.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class XsltParams
{
    public static void main(String [] args)
                          throws TransformerException,
                                 FileNotFoundException
    {
        if (args.length < 3)
        {
            System.out.print ("Usage: java XsltParams ");
            System.out.print("old.xml style.xsl new.xml ");
            System.out.println("param1 value1 ...");
            return;
        }

        TransformerFactory factory =
                     TransformerFactory.newInstance();

        StreamSource oldStream =
                     new StreamSource(new File(args[0]));

        StreamSource xslStream =
                     new StreamSource(new File(args[1]));
            StreamResult newStream =
                        new StreamResult(new File(args[2]));

            Transformer transformer =
                        factory.newTransformer(xslStream);

            for (int k=3; k+1<args.length; k=k+2)
                transformer.setParameter(args[k], args[k+1]);

            transformer.transform(oldStream, newStream);
        }
    }
```

Executing the Program

Enter this command on one line.

```
    % java XsltParams empty.xml dow.xsl output
                                        pm 11 pd 11 py 1918
```

Content of *output*

```
Monday
```

Fibonacci Numbers

The Fibonacci sequence has a simple inductive definition that we express as an XSLT method in a stylesheet.

fib(0) = 1

fib(1) = 1

fib(n) = fib(n-1) + fib(n-2) for n > 1   for n > 0

The stylesheet below defines the Fibonacci function and calls it with n=14 from the template that matches the root of the XML document.

File: fib.xsl

```
<?xml version="1.0"?>     <!-- Find a term in the Fibonacci -->
<!-- fib.xsl -->                  <!-- sequence  -->
<xsl:stylesheet  version="1.0"
             xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:call-template name="fibo">          <!-- method call -->
      <xsl:with-param name="n" select="14"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="fibo">          <!-- method definition -->
    <xsl:param name="n"/>
    <xsl:choose>
      <xsl:when test="$n=0 or $n=1">1</xsl:when>
      <xsl:when test="$n>1">
          <xsl:variable name="left">
              <xsl:call-template name="fibo">
                  <xsl:with-param name="n" select="$n - 1"/>
              </xsl:call-template>
          </xsl:variable>
          <xsl:variable name="right">
              <xsl:call-template name="fibo">
                  <xsl:with-param name="n" select="$n - 2"/>
              </xsl:call-template>
          </xsl:variable>
          <xsl:value-of select="$left+$right"/>
      </xsl:when>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Applying fib.xsl to empty.xml

```
610
```

Tail Recursion

The Fibonacci definition just presented is notoriously inefficient.

As the parameter *n* increases in value, the amount of computation required by this algorithm increases exponentially.

The problem arises from the "tree recursion" created by the two recursive calls, *fib(n-1)* and *fib(n-2)*, and the fact that some computation is required after the calls return.

An algorithm is *tail recursive* if there is only a single recursive call and that call is the last operation performed in the algorithm.

Tail recursive algorithms are equivalent to iterative algorithms with loops, and some language processors translate them into iterative algorithms automatically.

A tail recursive algorithm to compute the Fibonacci sequence requires three parameters, one for the counter and two to hold consecutive values in the sequence.

A Tail Recursive Definition

tfib(0, low, high) = low

tfib(n, low, high) = tfib(n-1, high, low+high)   if $n > 0$

Trace the definition to convince yourself that it is equivalent to the previous inductive definition:

for $n \geq 0$, fib(n) = tfib(n,1,1)

Next we define this algorithm as an XSLT function using default values for *low* and *high* in the first call of the function.

File: tfib.xsl

```
<?xml version="1.0"?>   <!-- Find a Fibonacci number -->
<!-- tfib.xsl -->               <!-- using tail recursion.  -->
<xsl:stylesheet  version="1.0"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
     <xsl:call-template name="tfib">   <!-- method call -->
       <xsl:with-param name="n" select="14"/>
     </xsl:call-template>
  </xsl:template>

  <xsl:template name="tfib">            <!-- method definition -->
   <xsl:param name="n"/>
   <xsl:param name="low"  select="1"/>
   <xsl:param name="high" select="1"/>
    <xsl:choose>
     <xsl:when test="$n &lt;= 0">
       <xsl:value-of select="$low"/>
     </xsl:when>
     <xsl:otherwise>
       <xsl:call-template name="tfib">
         <xsl:with-param name="n" select="$n - 1"/>
         <xsl:with-param name="low"   select="$high"/>
         <xsl:with-param name="high"  select="$high + $low"/>
       </xsl:call-template>
     </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

The result is the same as the previous stylesheet, *fib.xsl*.

XLST as a Programming Language

XLST is a fully Turing complete programming language, although there are many algorithms for which it is not well suited.

Some Language Properties (XSLT 1.0)

- Dynamically typed: Each identifier is bound to its type at runtime based on the kind of value bound to the identifier as the stylesheet is processed.
- Weakly typed: Type errors may not be detected in a stylesheet as it is processed.
- Parameters passed by value: Actual parameters are evaluated at the point of method call, and their values are bound to the formal parameters, which act as local variables.
- Static scoping is followed: Nonlocal identifiers are resolved based on their point of definition in the stylesheet and not on the calling order of methods.
- Parameters use keyword correspondence: The binding of actual parameters to formal parameters is defined using the names of the formal parameters and not the position of the parameters.
- Methods have no side effects: Methods cannot alter the values of variable identifiers.
- Declarative: The value of identifiers cannot be modified.
- Referentially transparent: Every subexpression can be replaced by any other that is equal to it in value.
- Determinacy: The value of an expression is independent of the order in which its subexpressions are evaluated.

Some Other Useful XSLT Elements

xsl:copy

This element copies the context node in the source document to the result document. This is a shallow copy; it does not copy the children, descendants, or attributes of the context node, only the context node itself.

xsl:copy-of

This element can be used to copy data to and from a temporary tree, and it can be used to copy a subtree unchanged from the input document to the output. When copying an element node, a deep copy is performed with all of its descendents also copied.

xsl:import

This element can be used to import the contents of one stylesheet into another. The definitions in the importing stylesheet have a higher import precedence than those in the imported stylesheet.

xsl:include

This top-level element can be used to include the contents of one stylesheet within another. The declarations in the included stylesheet have the same import precedence as those in the including stylesheet.

xsl:number

This element can be used to allocate a sequential number to the current node, and it can be used to format a number for output.

XPath 2.0

XPath has been extended in version 2.0 to include a new basic data structure and a large number of additional functions.

Sequences

Node sets have been generalized to sequences, which are lists, in the sense of Java List, of nodes and primitive values, including string.

Example

(15, "abc", <tag/>, 76.3, 15) is a list containing five items.

They occur in positions 1, 2, 3, 4, and 5 of the sequence.

Operations and Functions on Sequences

We illustrate the main operations on sequences by writing an XSLT stylesheet *sequences.xsl* that invokes them.

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

   <xsl:output method="text"/>

   <xsl:variable name="delimiter">
      <xsl:text>
==========================================
      </xsl:text>
   </xsl:variable>
   <xsl:variable name="newline">
      <xsl:text>
      </xsl:text>
   </xsl:variable>

   <xsl:template match="/">
      <xsl:value-of select="$delimiter"/>
      <xsl:text>1 to 8</xsl:text>
      <xsl:value-of select="$newline"/>
      <xsl:value-of select="1 to 8"/>

      <xsl:value-of select="$delimiter"/>
      <xsl:text>(1 to 4, 8, 13, 22)</xsl:text>
      <xsl:value-of select="$newline"/>
      <xsl:value-of select="(1 to 4, 8, 13, 22)"/>

      <xsl:value-of select="$delimiter"/>
      <xsl:text>for $k in 1 to 5 return $k*$k</xsl:text>
      <xsl:value-of select="$newline"/>
      <xsl:value-of select="for $k in 1 to 5 return $k*$k"/>

      <xsl:value-of select="$delimiter"/>
      <xsl:text>for $k in 1 to 4 return 1 to $k</xsl:text>
      <xsl:value-of select="$newline"/>
      <xsl:value-of select="for $k in 1 to 4 return 1 to $k"/>
<!--
   The stylesheet continues in this same manner with
   more function expressions using sequences.
   See the output for the rest.
-->
   </xsl:template>
</xsl:stylesheet>
```

Output

Apply *sequences.xsl* to the XML document *phoneA.xml*.

% **java net.sf.saxon.Transform empty.xml sequences.xsl**

```
==========================================
   1 to 8
   1 2 3 4 5 6 7 8
==========================================
   (1 to 4, 8, 13, 22)
   1 2 3 4 8 13 22
==========================================
   for $k in 1 to 5 return $k*$k
   1 4 9 16 25
==========================================
   for $k in 1 to 4 return 1 to $k
   1 1 2 1 2 3 1 2 3 4
==========================================
   for $m in 1 to 4, $n in 1 to 4 return ($m, $n)
   1 1 1 2 1 3 1 4 2 1 2 2 2 3 2 4 3 1 3 2 3 3 3 4 4 1 4 2 4 3 4 4
==========================================
   index-of((2,4,6,4,2), 4)
  2 4
==========================================
   insert-before(('a','b','c','d'), 2, 'X')
  a X b c d
==========================================
   distinct-values(for $m in 1 to 4, $n in 1 to 4 return ($m, $n))
   1 2 3 4
```

```
========================================
   reverse(1 to 8)
  8 7 6 5 4 3 2 1
========================================
   remove(11 to 18, 5)
  11 12 13 14 16 17 18
========================================
   subsequence(3 to 10, 4)
  6 7 8 9 10
========================================
   subsequence(3 to 10, 4, 2)
  6 7
========================================
   empty(//item)
  true
========================================
   empty(//entry)
  false
========================================
   exists(//item)
  false
========================================
   exists(//entry)
  true
========================================
```

We will investigate XPath sequences in more detail when we look at XQuery.

Additional String Functions

XPath 2.0 also includes a collection of new functions that deal with strings and sequences of strings.

We illustrate these functions the same way as the sequence operations.

Here are a few of the templates that will occur a the XSLT stylesheet called *strings.xsl*.

```
<xsl:value-of select="$delimiter"/>
<xsl:text>ends-with('file.xml', '.xml')</xsl:text>
<xsl:value-of select="$newline"/>
<xsl:value-of select="ends-with('file.xml', '.xml')"/>

<xsl:value-of select="$delimiter"/>
<xsl:text>lower-case('Hello')")</xsl:text>
<xsl:value-of select="$newline"/>
<xsl:value-of select="lower-case('Hello')"/>
```

Output

Apply *strings.xsl* to an "empty" XML document.

% **java net.sf.saxon.Transform empty.xml strings.xsl**

```
=============================================
    codepoints-to-string(65 to 68)
   ABCD
=============================================
    string-to-codepoints('Herky')
   72 101 114 107 121
=============================================
   ends-with('file.xml', '.xml')
   true
```

```
===========================================
   lower-case('Hello Sailor')")
   hello sailor
===========================================
   upper-case('Herky')")
   HERKY
===========================================
   replace('banana', 'a', 'oo')
    boonoonoo
===========================================
   string-join(('a','bb','cc'), ', ')
   a, bb, ccc
===========================================
   substring('abcdefg', 3)
   cdefg
===========================================
   substring('abcdefg', 3, 2)
   cd
===========================================
   tokenize('Go home, Jack!', '\W+')
   Go home Jack
===========================================
   tokenize('abc[NL]def[XY]gh[]i', '\[.*?\]')
   abc def gh i
===========================================
   compare('ant', 'bug')
   -1
===========================================
   compare('lynx', 'gnat')
   1
===========================================
   compare('pelican', 'pelican')
   0
```

```
=========================================
   current-date()
  2006-10-26-05:00
=========================================
   current-time()
  14:37:56.123-05:00
=========================================
```

XPath 2.0 also includes a few more numeric functions such as *abs*, *avg*, *max*, and *min*, a large number of functions for handling date and time values, and a few more miscellaneous functions.

See Michael Kay's book *XPath 2.0: Programmers's Reference* for more details.

XSLT 2.0

XSLT 2.0 incorporates all of the extensions in XPath 2.0 and contains some enhancements of its own.

The new version of XSLT incorporates about a dozen additional elements and a few functions related to these new elements.

We will concentrate this presentation on one element and a couple of associated functions.

xsl:for-each-group Element

This instruction selects a set of items, arranges them into groups based on common values or other criteria, and then processes each group in turn.

Attributes

select defines the sequence of items to be grouped.

group-by defines the grouping key; items with common values for the grouping key are placed in the same group.

Related Functions

current-group returns the set of items in the group currently being processed by an *xsl:for-each-group* element.

current-group-key returns the value of the grouping key that defines the group currently being processed by an *xsl:for-each-group* element.

Examples

XML Document: staff.xml

```
<?xml version="1.0"?>     <!-- staff.xml -->
<staff>
    <employee name="Cinda Lott"
                            department="Sales"/>
    <employee name="Ginger Vitus"
                            department="Personnel"/>
    <employee name="Virginia Ham"
                            department="Transport"/>
    <employee name="Sally Forth"
                            department="Personnel"/>
    <employee name="Robin Banks"
                            department="Sales"/>
</staff>
```

Stylesheet: group-by-dept.xsl

```
<?xml version="1.0"?>     <!-- group-by-dept.xsl -->
<xsl:stylesheet version="2.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" indent="yes"/>
```

```
    <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="staff"/>
      </body>
    </html>
    </xsl:template>

    <xsl:template match="staff">
      <xsl:for-each-group select="employee"
                                   group-by="@department">
        <h2><xsl:value-of select="current-grouping-key()"/>
        <xsl:text> department</xsl:text></h2>
        <xsl:for-each select="current-group()">
          <p><xsl:value-of select="@name"/></p>
        </xsl:for-each>
      </xsl:for-each-group>
    </xsl:template>
  </xsl:stylesheet>
```

Output

Apply *group-by-dept.xsl* to *staff.xml*.

% **java net.sf.saxon.Transform staff.xml group-by-dept.xsl**

```
  <html>
    <body>
        <h2>Sales department</h2>
        <p>Cinda Lott</p>
        <p>Robin Banks</p>
        <h2>Personnel department</h2>
        <p>Ginger Vitus</p>
        <p>Sally Forth</p>
        <h2>Transport department</h2>
        <p>Virginia Ham</p>
    </body>
  </html>
```

XML Document: othello.xml

```
<?xml version="1.0"?>      <!-- othello.xml -->
<PLAY>
<TITLE>The Tragedy of Othello, the Moor of Venice</TITLE>

<FM>
<P>Text placed in the public domain by Moby Lexical Tools,
1992.</P>
<P>SGML markup by Jon Bosak, 1992-1994.</P>
<P>XML version by Jon Bosak, 1996-1998.</P>
<P>This work may be freely copied and distributed worldwide.</P>
</FM>

<PERSONAE>
<TITLE>Dramatis Personae</TITLE>

<PERSONA>DUKE OF VENICE</PERSONA>
<PERSONA>BRABANTIO, a senator.</PERSONA>
<PERSONA>Other Senators.</PERSONA>
<PERSONA>GRATIANO, brother to Brabantio.</PERSONA>
<PERSONA>LODOVICO, kinsman to Brabantio.</PERSONA>
<PERSONA>OTHELLO, a noble Moor in the service of the
Venetian state.</PERSONA>
<PERSONA>CASSIO, his lieutenant.</PERSONA>
<PERSONA>IAGO, his ancient.</PERSONA>
<PERSONA>RODERIGO, a Venetian gentleman.</PERSONA>
<PERSONA>MONTANO, Othello's predecessor in the
government of Cyprus.</PERSONA>
<PERSONA>Clown, servant to Othello. </PERSONA>
<PERSONA>DESDEMONA, daughter to Brabantio and wife to
Othello.</PERSONA>
<PERSONA>EMILIA, wife to Iago.</PERSONA>
<PERSONA>BIANCA, mistress to Cassio.</PERSONA>
```

```
<PERSONA>Sailor, Messenger, Herald, Officers, Gentlemen,
Musicians, and Attendants.</PERSONA>
</PERSONAE>

<SCNDESCR>SCENE  Venice: a Sea-port in
Cyprus.</SCNDESCR>

<PLAYSUBT>OTHELLO</PLAYSUBT>

<ACT><TITLE>ACT I</TITLE>

<SCENE><TITLE>SCENE I.  Venice. A street.</TITLE>
<STAGEDIR>Enter RODERIGO and IAGO</STAGEDIR>

<SPEECH>
<SPEAKER>RODERIGO</SPEAKER>
<LINE>Tush! never tell me; I take it much unkindly</LINE>
<LINE>That thou, Iago, who hast had my purse</LINE>
<LINE>As if the strings were thine, shouldst know of
this.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>IAGO</SPEAKER>
<LINE>'Sblood, but you will not hear me:</LINE>
<LINE>If ever I did dream of such a matter, Abhor me.</LINE>
</SPEECH>

  :

<STAGEDIR>Exeunt</STAGEDIR>
</SCENE>
</ACT>
</PLAY>
```

Stylesheet: wordcount.xsl

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

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

  <xsl:template match="/">
    <wordcount>
      <xsl:for-each-group group-by="." select=
  "for $w in tokenize(string(.), '\W+') return lower-case($w)">
        <xsl:sort select="count(current-group())"
                                order="descending"/>
      <word word="{current-grouping-key()}"
                  frequency="{count(current-group())}"/>
      </xsl:for-each-group>
    </wordcount>
  </xsl:template>
</xsl:stylesheet>
```

Output

Apply *wordcount.xsl* to *othello.xml*.

% **java net.sf.saxon.Transform othello.xml wordcount.xsl**

```
<?xml version="1.0" encoding="UTF-8"?>
<wordcount>
  <word word="i" frequency="899"/>
  <word word="and" frequency="796"/>
  <word word="the" frequency="764"/>
  <word word="to" frequency="632"/>
  <word word="you" frequency="494"/>
```

```
<word word="of" frequency="476"/>
<word word="a" frequency="453"/>
<word word="my" frequency="427"/>
<word word="that" frequency="396"/>
<word word="iago" frequency="361"/>
<word word="in" frequency="343"/>
<word word="othello" frequency="336"/>
<word word="it" frequency="319"/>
<word word="not" frequency="319"/>
<word word="is" frequency="309"/>
<word word="me" frequency="281"/>
<word word="cassio" frequency="254"/>
<word word="he" frequency="247"/>
<word word="for" frequency="240"/>
<word word="desdemona" frequency="230"/>
<word word="be" frequency="227"/>
<word word="s" frequency="225"/>
<word word="this" frequency="223"/>
<word word="but" frequency="222"/>
<word word="with" frequency="222"/>
<word word="do" frequency="221"/>
<word word="her" frequency="216"/>
<word word="have" frequency="207"/>
<word word="your" frequency="207"/>
<word word="what" frequency="197"/>
<word word="him" frequency="187"/>
<word word="she" frequency="173"/>
<word word="as" frequency="171"/>
<word word="his" frequency="167"/>
<word word="o" frequency="163"/>
<word word="so" frequency="162"/>
<word word="d" frequency="153"/>
<word word="will" frequency="152"/>
<word word="thou" frequency="145"/>
```

```
<word word="if" frequency="140"/>
<word word="emilia" frequency="138"/>
<word word="on" frequency="120"/>
<word word="by" frequency="113"/>
<word word="are" frequency="109"/>
<word word="now" frequency="106"/>
<word word="roderigo" frequency="105"/>
    :        :                    :
```

File Sizes

othello.xml contains 248742 characters in 8776 lines of text.

The output file contains 144597 characters in 3674 lines, which means the source XML document has 3671 different word in its content.

More Information

See Michael Kay’s book *XSLT 2.0: Programmers's Reference* for more details about the extensions to XSLT in version 2.0.