Procedural XSL

Though the underlying structure of XSLT is data-driven, there are some prodeural programming options.

The `xsl:for-each` Element

```
<xsl:for-each select="customers/customer">
    <tr><td><xsl:apply-templates select="name"/></td><td>
        <xsl:for-each select="order"><xsl:apply-templates/>,
</xsl:for-each>
    </td></tr>
</xsl:for-each>
```

This replicates something we could already do with a data-driven model.

Sorting

There's an element `xsl:sort` which can be used to sort collections of nodes.

The format is as follows:

```
<xsl:sort
    select = string-expression
    lang = {nmtoken}
    data-type = {"text"|"number"|qname-but-not-ncname}
    order = {ascending|descending}
    […]
/>
```

I think it has to be the first thing in the parent `for-each`.

You can include this as a child of an `xsl:for-each` element to sort the result, e.g.:

```
[…]
<xsl:for-each select="customers/customer">
    <xsl:sort select="name"/>
    […]
</xsl:for-each>
```

The sort statement here will only be executed once, not once each time the loop is executed. It sorts the `customer` elements in the default manner – text-based; ascending; upper case before lower case.

Note that the sort statement does not restructure the parse tree. When the computer is executing the `for-each` instruction, the system builds the list of nodes specified in the XPath expression, and the `sort` instruction sorts that list.

Multiple-Key Sorting

You can sort by (e.g.) surname and then by first name by putting in multiple `xsl:sort` elements. The first one will be the primary sort order and the second the secondary:

```
[…]
<xsl:sort select="name/surname"/> <xsl:sort
select="name/firstname"/>
[…]
```

This will sort by surname (for the appropriately structured XML) and then by first name within that for entries with the same surname.

`xsl:variable`

Even though this language support variables, it's not the kind of variable we see

in Python or Java (or procedural programming languages in general).

The format is:

```
<xsl:variable name=qname select=expression/>
```

or

```
<xsl:variable name=qname>
    <-- Content: template -->
</xsl:variable>
```

or

```
<xsl:variable name=qname/>
```

XSL variables cannot be changed once they've been created. They can only have the value they are first given.

- The first format above creates a variable, gives it a name, and gives it the value `expression`.
- The second format creates a variable, gives it a name, and gives it the string value that results from executing whatever's between the open and close tags.
- The third format creates a variable, gives it a name, and gives it the value of the empty string.

Variables are needed when xml documents are created from databases with a lot of cross-referencing between database tables.

Example Usage

```
[…]
<xsl:for-each select="./stock/secondHandBooks/book">
    <xsl:variable name="thisISBN" select="./@isbn"/>
    <tr>
        <td><xsl:value-of select="price"/></td>
        <td><xsl:value-of select="$thisISBN"/></td>
        <td><xsl:value-of
select="/stock/publications/book[@isbn=$thisISBN]/title"/>
</td>
    </tr>
</xsl:for-each>
[…]
```

This XSLT outputs a table row for each `book` element in `secondHandBooks`, and uses the ISBN attribute to cross-reference between `book` elements in `secondHandBooks` and `book` elements in `publications`.

Side Note

Using the `.` operator in an XPath expression in a position other than the start (e.g. `"/stock/."` rather than `"./book"`) will cause the `.` to have a meaning other than what we're used to, and it won't do what we want. We'll see later what meaning it has.

Variable Scope

XSL variables can only be accessed inside the element they are created in. Outside of that element, they will not be accessible.

In the example code above, if we try to access `thisISBN` outside of the `<xsl:for-each>` block, we will get an error.

Alternatively, if we define that variable deeper inside the `<xsl:for-each>` statement (e.g. inside a `<td>`), we won't be able to access it outside (e.g.) that `<td>`.

Conditional Processing

Conditional processing is already provided by the `select` and `match` attributes, but there are two constructs for explicitly specifying conditional processing:

- The `<xsl:if>` element.
- The `<xsl:choose>` element.

`xsl:if` Syntax

```
<xsl:if test="boolean expression">
    <!--Content: template-->
</xsl:if>
```

`xsl:choose` Syntax

```
<xsl:choose>
    <!--Content: (xsl:when+ xsl:otherwise?) -->
</xsl:choose>

<xsl:when test="boolean expression">
    <!--Content: template -->
</xsl:when>

<xsl:otherwise>
    <!--Content: template-->
</xsl:otherwise>
```

`<xsl:call-template>` Instruction

Note that this doesn't change the current node, unlike `<xsl:apply-`

`templates>`. Here `.` refers to the same node before executing the template referred to in `<xsl:call-template>` as it does during execution.

Parameters

You can also pass parameters to named templates.

```
<xsl:call-template name="show_item">
    <xsl:with-param name="thisNum" select="./@num"/>
</xsl:call-template>

<xsl:template name="show_item">
    <xsl:param name="num"/>
    <xsl:value-of select="$num"/>
</xsl:template>
```

The `<xls:with-param>` element allows you to specify the actual parameters (i.e. the values passed to the function), and the `<xsl:param>` element allows you to specify the formal parameters (the parameters referred to in the function definition).

Parameters are specified positionally – the value given in the first `<xsl:with-param>` element will be given to the first `<xsl:param>` parameter.