Rendering XML Documents

Introduction

No XML tag has any pre-defined rendering semantics. There are at least 4 options:

- Most browsers are starting to introduce very simple default rendering semantics for arbitrary XML tags.
- Most browsers accept CSS specifications for rendering XML tags.
- The most powerful approaches involve using XSL (Extensible Stylesheet Language), a powerful language which enables arbitrary ways of rendering XML documents:
    - Most browsers accept XSL stylesheets.
    - Server-side software driven by XSL stylesheets can transform XML documents into HTML documents before serving them to browsers which cannot understand XML+XSL.

We will look at all four approaches. Most of our time will be spent on XSL, and on client-side XSL rather than server-side.

Default Rendering Semantics

The XML is displayed onscreen (tags and all), with some colour-coding of tags, and the ability to collapse tags that have element (or presumably mixed?) content via a minus next to the tag.

It's useful to examine the XML sometimes.

Using CSS with XML Documents

```
<?xsl:stylesheet type="text/css" href="stylesheet.css"?>
```

This line is placed after the DOCTYPE and before the root node.

It's very straightforward, but may not be useful at all.

XML Without Rendering

We're used to HTML, which is always rendered by a browser. XML has many uses that do not involve rendering, e.g. podcasts, data exchange.

Rendering XML Documents with XSL

The most powerful approaches to rendering XML documents involve using XSL.

XSL enables arbitrary ways of rendering XML documents. XSL enables an XML document to be translated into any arbitrary format, including PDF or HTML[1].

We will consider browser-processed XSL stylesheets first, and then server-side use of XSL.

Browser-processed XSL

Consider the XML specification below:

```
<?xml version="1.0"?>
<!DOCTYPE people SYSTEM "personnel2.dtd">
<?xml-stylesheet type="text/xsl" href="personnel2.xsl"?>
…
```

It refers to an XSL stylesheet. When the broswer reads the file and reaches the third line, it sends a second request to retrieve the stylesheet.

Overview of XSL

Every XML document can be viewed as a tree. There are two main stages to rendering an XML document using XSL – XSL transform and XSL formatting objects.

XSL Transform

XSL Transform takes an input tree and converts it to a new tree.

XSL Formatting Objects

XSL formatting formats the resulting document. We won't be looking at that in this course.

XSLFO is mostly not used, because XSL is usually used to convert XML into more XML, or convert it into HTML (which is then presumably styled with CSS).

Browser-side Usage of XSL

We use XSLT to convert XML into XHTML, which is then displayed on a browser screen.

Trees

The types of nodes in the tree include element nodes, attribute nodes, and text nodes. (There are about 13 types of node overall.)

A stylesheet in XSLT consists of a set of **template rules**.

Template Rules

A template rule has two parts:

- A pattern which is matched against nodes in the source tree
- A template which can be instantiated to form part of the result

If it matches the pattern, apply the template.

XSLT Stylesheets

An XSLT stylesheet is itself an XML document.

The root element is of type `xsl:stylesheet` but `xsl:transform` may be used as a synonym `xsl:stylesheet`.

Here's an example:

```
<?xml version="1.0"?>
<xsl:stylesheet … >
…
</xsl:stylesheet … >
```

Two attributes are required for the root element:

- `version` – the correct value at the moment is "1.0"
- `xmlns:xsl`[2] – the correct value at the moment is "http://www.w3.org/1999/XSL/Transform"

Elements Withing the Root Element

The root element may contain many different types of elements. The most commonly used element is the `xsl:template` element.

XSL Template Elements

Each `xsl:template` element has a `match` element which is used to specify the type of node that is matched.

The Match Attribute

The value of a `match` attribute is an expression from a language called XPath. We'll look at XPath later.

For now, know that an XPath expression simply specifies a certain class of node in an XML file.

One simple expression would be to use a tag name:

```
<xsl:template match="person">
…
</xsl:template>
```

However, certain meta-characters may also be used in XPath expressions.

XPath Meta-Characters

Meta-characters that may be used include: `*`, `|`, `/` `//`.

Here are some examples:

- The pattern `/` matches the root node
- The pattern `*` matches any element
- The pattern `a|b` matches any `a` element and any `b` element
- The pattern `poem/verse` matches any `verse` element with a `poem` parent
- The pattern `poem//line` matches any `line` element with a `poem` element as an ancestor

Fragment from an XSLT Stylesheet

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

Content of XSL Template Elements

The content specifies what to generate in the output.

It may:

- Specify canned text.
- Contain XSLT instruction elements.
    - These specify certain kinds of processing that should be performed on the source node in order to compute the appropriate result text.
    - The output will be whatever results from the execution of these instructions.
- Contain a mixture of canned text and XSLT instruction elements.

Canned Text in an XSL Stylesheet

Here's the XML:

```
<?xml version="1.0"?>
<!DOCTYPE people SYSTEM "bertie.dtd">
<?xml-stylesheet type="text/xsl" href="bertie.xsl"?>
<people>
    <person>
        <female>Celia Larkin</female>
    </person>
    <person>
        <male>Bertie Ahern</male>
    </person>
</people>
```

And here's the stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <html><body><h1>People</h1>
            <p>This document contains information about some
people.</p>
        </body></html>
    </xsl:template>
</xsl:stylesheet>
```

The `template` matches the root node, and generates the canned text within.

Canned HTML text must be well-formed XML.

XSLT Instruction Elements

A template can contain canned text. It can also contain XSLT intruction elements.

When the template is instantiated, each instruction is executed, and the text fragment that it creates is placed in the result tree.

Instructions can process [check].

An XSLT Instruction Element

The instruction `<xsl:apply-templates>` does the following:

- Processes the content of the current element.
- Processes any descendents of the current element.
- It applies the template rules that match those descendent elements.

Example

Same XML as before.

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="people"> <p>I found some people.
</p> </xsl:template>
    <xsl:template match="/"> <xsl:apply-templates/>
</xsl:template>
</xsl:stylesheet>
```

These templates can be written in any order, and the output will not be affected[3]. This is different from what we are used to.

The structure of the incoming XML document decides which templates will be executed first. XSLT is an example of a "data-driven language".

Parsed Tree

When a browser reads an XML document, it will parse it to check if it's well-formed (among other things), and will generate a parsed tree. For our running XML example:

- One node will be created for the root node of the document.

Children of that node will be created as follows:

- One child node will be created for the xml version statement.
- One child node will be created for the doctype statement.
- One child node will be created for the stylesheet statement.
- One child node will be created for the people element.

Children of the people element will be created as follows:

- One child node will be created for the first person element.
- One child node will be created for the second person element.

Children of those person elements will be created as follows:

- One child node will be created for the female element that is a child of the first person node.
- One child node will be created for the male element that is a child of the second person node.

Each of those nodes will have another kind of node as a child node which will contain the plaintext within the element.

In this parsed tree, there are three kinds of node:

- The root node.
- Element nodes.
- Text nodes.

Order of Operations

Through this tree, XSL will follow "top-down left-to-right traversal".

Starting at the top, you move down. The next node you visit is the leftmost child of the root node. In our example, they will be visited in the following order:

- The Root node.
- The XML version node.
- The Doctype node.
- The Stylesheet node.
- The People element node.

At this point, the first person node will be chosen, as the leftmost, and then its children will be visited, before gradually coming back up, giving the following order:

- The first Person element node.
- The Female element node.
- The text node child of the Female element node.
- The second Person element node.
- The Male element node.
- The text node child of the Male element node.

In our XSL file, at each node in the tree, the browser checks the stylesheet to see if there are any instructions for this node.

The browser will check the root node first, and find that while the first template doesn't apply, the second does, so the second will be executed. The instruction `<xsl:apply-templates/>` tells the browser to look at all the children of this root element in turn and see if any templates apply to it. If they do, the template will be executed.

Once the people element has been processed, its children will be ignored, because there's no `<xsl:apply-templates/>` instruction in the template which matches the people element.

Another Example

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <body>
            <xsl:apply-templates/>
        </body>
    </xsl:template>
    <xsl:template match="male">
        <p>I found a man.</p>
    </xsl:template>
    <xsl:template match="female">
        <p>I found a woman.</p>
    </xsl:template>
</xsl:stylesheet>
```

One More

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <body>
            <xsl:apply-templates/>
        </body>
    </xsl:template>
    <xsl:template match="person">
        <p>I found a person.</p>
    </xsl:template>
    <xsl:template match="male">
        <p>I found a man.</p>
    </xsl:template>
    <xsl:template match="female">
        <p>I found a woman.</p>
    </xsl:template>
</xsl:stylesheet>
```

In this example (applied to our recurring XML file), the `<male>` and `<female>` elements are not processed, because the instructions for processing `<person>` elements do not include instructions to process their descendants.

The templates that match `<male>` and `<female>` elements are never used.

I Lied, Here's Another

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <body>
            <xsl:apply-templates/>
        </body>
    </xsl:template>
    <xsl:template match="person">
        <p>I found a person. It was <xsl:apply-templates/>.
</p>
    </xsl:template>
    <xsl:template match="male">a man</xsl:template>
    <xsl:template match="female">a woman</xsl:template>
</xsl:stylesheet>
```

In this example, when the browser finds a person element, it outputs `<p>I found a person. It was` and then applies templates to the descendants, as instructed. In this way, we can tell the browser when to move on to the descendants for more info. When it reaches the child, it either outputs `a man` or `a woman` and then returns to the person template, which tells it to output `. </p>`.

Note

You need to understand these last few examples in order to know what's going on from here on.

The `select` Attribute

The `select` attribute for the `<xsl:apply-templates/>` instruction is used to limit the application of templates. If you change the last example to this:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <body>
            <xsl:apply-templates/>
        </body>
    </xsl:template>
    <xsl:template match="person">
        <p>I found a person. <xsl:apply-templates
select="male"/></p>
    </xsl:template>
    <xsl:template match="male">It was a man.</xsl:template>
    <xsl:template match="female">It was a woman.
</xsl:template>
</xsl:stylesheet>
```

Then the browser will only execute one of the last pair of templates when it finds a `male` element. If it finds a `female` element, it won't look for any templates, so none will be executed.

This attribute can take any valid XPath value. We'll see the full range of XPath values later.

The `xsl:value-of` Instruction

This can be used to generate result text by extracting data from the source tree. Its required `select` attribute specifies the data to be extracted. In the values of the `select` attribute, the dot character means "the context node". For now we can think of this as the current node, though sometimes there is a difference. We'll see this later.

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <body>
            <xsl:apply-templates/>
        </body>
    </xsl:template>
    <xsl:template match="female">
        <p>I found a woman. Her name is <xsl:value-of
select="."/>.</p>
    </xsl:template>
    <xsl:template match="male">
        <p>I found a man. His name is <xsl:value-of
select="."/>.</p>
    </xsl:template>
</xsl:stylesheet>
```

In these templates, the `xsl:value-of` instruction says "output the value of the current node." By default, the value of an element node is the concatentation of any text nodes which are children of the element node. So these instructions output the names contained in the `<male>` and the `<female>` nodes.

Extracting Attribute Values

You can also extract attribute values with this. From XPath, we can use the `@` character to prefix an attribute name:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <body>
            <xsl:apply-templates/>
        </body>
    </xsl:template>
    <xsl:template match="female">
        <p>I found a woman. Her name is <xsl:value-of
select="."/> and her
        age is <xsl:value-of select="@age"/>.</p>
    </xsl:template>
    <xsl:template match="male">
        <p>I found a man. His name is <xsl:value-of
select="."/> and his age
        is <xsl:value-of select="@age"/>.</p>
    </xsl:template>
</xsl:stylesheet>
```

This stylesheet would process a slightly different XML file, where our `<male>` and `<female>` elements have an `age` attribute.

Each attribute has a corresponding node in the parse tree, which is the child of the node for the element it's part of.

Back to the Table Example

```
<?xml version="1.0"?>
<xsl:transform
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:template match="/">
        <html>
            <body>
                <table rules="all">
                    <thead>
                        <tr><th>Name</th><th>Sex</th></tr>
                    </thead>
                    <tbody>
                    <xsl:apply-template/>
                    </tbody>
                </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="person">
        <tr><xsl:apply-templates/></tr>
    </xsl:template>
    <xsl:template match="male">
        <td><xsl:value-of select="."/></td><td>male</td>
    </xsl:template>
    <xsl:template match="female">
        <td><xsl:value-of select="."/></td><td>female</td>
    </xsl:template>
</xsl:transform>
```

Top-Down Program Development

It's a way of writing a program. You think about the overall task, break it down

into subtasks (like a tree), and continue breaking down subtasks until the subtasks are simple.

Embedding CSS

Suppose we want to embed some CSS – we can embed it in the html, or we can generate a reference to an external stylesheet. Any CSS must be escaped so that it doesn't cause your XML to be ill-formed.

Example:

```
…
<style> <![CDATA[table {background-color:red}]]> </style>
```

Bug in Microsoft Browsers

Microsoft browsers (except possibly Edge) cache the stylesheets and don't update them when you refresh, if the XML documents haven't changed. You need to restart the browser to clear the cache.

XPath Intermission

We've seen some XPath expressions already. Here's some more.

XPath is a W3C standard. It's a syntax for navigating around the parse tree for an XML document. It provides a library of more than 100 functions for building these path expressions.

It's used in XSLT, but it's also used in other technologies like XQuery and XPointer.

XPath Nodes

XPath views an XML document as a tree of nodes. There are seven kinds of node:

- Element nodes
- Attribute nodes
- Text nodes
- Namespace nodes

- Processing-instruction nodes
- Comment nodes
- Document (root) nodes

In an XML document, the root element is not the root node of the XPath tree view of the document.

The root node is a virtual node which contains the root element. The root node corresponds to the whole document. The node which contains the root element is an element node.

Here's an example parse tree:

- There's a root node with two children.
- One child is the xml version statement. The other is an element node corresponding to the catalogue element.
- The catalogue element node then has children and descendant element nodes which follow the structure of the XML.

So the root node and the node corresponding to the root element (which is an element node) are separate.

XPath Path

An XPath path can be either absolute or relative. An absolute path starts with the single character `/`. A relative path starts from the current *context node*.

The root node is the default context node, but other nodes can become the current context node at various times.

`name()` Function

The instruction below says "output the value of the expression '`name()`'". `name()` is a function that returns the name of the current node. It's designed to be used with element nodes to return the tag name.

```
<xsl:value-of select="name()"/>
```

Locating Children of the Root Element

`*` is a meta-character with similar meaning to its meaning in Bash. `/catalogue/*` will match any node which is a child of a `catalogue` node which is a child of the root node. There are some subtleties to this that we'll cover later.

[missed a bit]

Default Processing in XSLT

- Root node: Apply templates to children
- Element nodes: Apply templates to children
- Text nodes: Copy text to the result tree
- Comment nodes: Do nothing
- Processing Instruction nodes: Do nothing
- Attribute nodes: Copy the value of the attribute to the result tree
- Namespace nodes: Do nothing

Overriding Default Processing

The presence of any templates in the xslt which

[…]

XPath Axes

An XPath node has 13 axes:

- `ancestor` – ancestors of the current node
- `ancestor-or-self` – ancestors including the current node
- `attribute` – attributes of the current node (abbreviated `@`)
- `child` – children of the current node (the default axis)
    - never contains namespace nodes
- `descendant` – descendants of the current node (thus never contains attribute or namespace nodes)

- `descendant-or-self` – descendants including the current node (abbreviated `//`)
- `following` – nodes which occur after the current node in the TDLR traversal
- `following-sibling` - nodes which occur after the current node in the TDLR traversal and have the same parent
- `namespace` – The namespace nodes of the current node. An element has a namespace node:
    - for every attribute on the element whose name starts with `xmlns:`
    - for every attribute on an ancestor element whose name starts `xmlns:` unless the element itself or a nearer ancestor redeclares the prefix
    - for an xmlns attribute, if the element or some ancestor has an xmlns attribute, and the value of the xmlns attribute for the nearest such element is non-empty
- `preceeding` / `preceeding-sibling` – obvious as with `preceeding`, `preceeding-sibling`
- `parent` – the parent of the current node (abbreviated `..`)
- `self` – the current node (abbreviated `.`)

Axis (and Node Test) Examples

```
<xsl:value-of select="count(./catalogue/descendant::*)"/>
```

This statement will count element descendant nodes of the catalogue node. It only counts element descendants because the node test `*` matches only nodes of the current principal type – in this case element nodes.

```
<xsl:value-of
select="count(./catalogue/descendant::node())"/>
```

This statement will count all descendant nodes of the catalogue node. `node()` is

a call to a function which returns true for any node. The attribute nodes are still not counted because they are not regarded as children or descendants.

```
<xsl:value-of
select="count(./catalogue/descendant::text())"/>
```

The node test `text()` is true only for text nodes.

```
<xsl:value-of
select="count(./catalogue/descendant::comment())"/>
```

The node test `comment()` is true only for comments.

```
<xsl:value-of
select="count(./catalogue/descendant::price)"/>
```

The node test `price` is true only for nodes corresponding to price elements.

```
<xsl:value-select="count(./preceding-
sibling::*)+count(./following-sibling::*)"/>
```

You can add together preceding-sibling and following sibling counts to give the number of siblings.

---

1. One cool example would be to have an XSL file that converts an XML file that describes an object into machine code that instructs a 3D printer how to print it.↩
2. You can of course replace the `xsl` prefix with any other prefix as long as the URL is correct.↩
3. I've gone with the reversed order rather than the one we find intuitive.↩