

XSLT

Given one or several XML documents, it may be useful

- to search for information in the documents,
- to output what is found in a format suitable for another application or reader.

so one needs XSLT (**eXtensible Stylesheet Language Transformations**).

An XSLT processor

- reads an XML document,
- an XSLT file,

and applies the transformations to the XML and the result is printed out.

XSLT/Saxon

I recommend the open source, Java-based, XSLT processor available at

<http://sourceforge.net/projects/saxon/files/Saxon-HE/9.3/saxonhe9-3-0-4j.zip/download>

Its name is Saxon. You only need the archive `saxon9he.jar`.

The command-line syntax for using Saxon is

```
java -jar saxonhe9.jar -o output doc.xml trans.xsl
```

If no *output* file is specified, the output is the terminal.

XSLT

An XSLT file is actually an XML document, in the sense that XML is both the infrastructure and its contents, which is interpreted by some other application.

For example, if we use an element `<book>` in an XML document, XML itself does not imply that this element models a book: some application using the document perhaps will precisely do that.

One can think of XML as syntactic rules (see page 61), but with no semantics attached to the constructs.

An XSLT document is thus XML with a special interpretation (which makes it XSLT).

XSLT/Empty transformation

For reason explained later, XSLT documents require the usage of a namespace, usually `xsl`, defined at <http://www.w3.org/1999/XSL/Transform>

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
</xsl:transform>
```

Note the first line of this transformation, which says that this is an XML document. The second line declares the namespace for XSLT and makes use of an XSLT tag name, `transform`, which means that the interpretation of this XML is done according to XSLT. The version of XSLT is declared to be 2.0.

XSLT/Empty transformation (cont)

Assume the following XML document to be transformed:

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

```
<cookbook>
```

```
  <title>XSLT Cookbook</title>
```

```
  <author>Salvatore Mangano</author>
```

```
  <chapter>XPath</chapter>
```

```
  <chapter>Selecting and Traversing</chapter>
```

```
  <chapter>XML to Text</chapter>
```

```
  <chapter>XML to XML</chapter>
```

```
  <chapter>XML to HTML</chapter>
```

```
</cookbook>
```

XSLT/Empty transformation (cont)

The result of applying the empty transformation page 94 to this document yields

XSLT Cookbook

Salvatore Mangano

XPath

Selecting and Traversing

XML to Text

XML to XML

XML to HTML

XSLT/Empty transformation (cont)

Note that if the element `<xsl:output method="text"/>` were missing, the output would be considered XML and `<?xml ...?>` would be outputted by default.

Then it prints the contents of the text nodes of the input XML document *in the same order*.

More precisely, the order corresponds to a prefix traversal: this is the implicit traversal supported by XSLT processors, also called **document order**. The rationale is that since the aim is often to rewrite a document into another, this traversal corresponds to the order in which the input is read.

The attributes are not copied by default.

XSLT/Matching

More precisely, the concepts underlying XSLT transformations are

- an implicit prefix traversal of the XML tree,
- each element is matched against a **template**,
- as a result, some output may be produced.

A template allows to identify an element by specifying a part of it, like its name, some of its attribute names etc.

When a template identifies an input element, one says it **matches** the element.

If no template matches the current node, the children nodes are visited and tried for matching. Text nodes implicitly match and their contents are printed (this is the default behaviour).

XSLT/Matching (cont)

Consider the following XSLT transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="chapte">A chapter.</xsl:template>
</xsl:transform>
```

Note the XSLT predefined element template, which defines a template, and its predefined attribute match, whose value is the element name one wishes to match in the input XML document.

The content of the template is output only if a chapte (**wrong spelling**) element is matched (i.e., found).

XSLT/Matching (cont)

The result of applying the previous transformation to the document page 95 is

XSLT Cookbook

Salvatore Mangano

XPath

Selecting and Traversing

XML to Text

XML to XML

XML to HTML

because the template matched no node in the input tree, but **text nodes always implicitly match**.

XSLT/Matching (cont)

Let us try to match also the root element and try the next transformation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="cookbook">Chapters:</xsl:template>
</xsl:transform>
```

The result is now:

Chapters:

Why?

XSLT/Applying several templates

The reason is that when a template matches the current element, this element is transformed and the prefix traversal goes on *without visiting the children of the current element*.

Therefore, after the element `cookbook` is matched, the XSLT processor ignores everything else since it is the root element (hence, no element `chapter` is matched).

In order to try to match the children elements of a matched element, one must tell so the processor by using the special empty element

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

The meaning of all the XSLT elements is implicitly relative to the last matched element, called the **context node**.

XSLT/Applying several templates (cont)

The following transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="cookbook">
    <xsl:apply-templates/>
  </xsl:template>
</xsl:transform>
```

instructs the XSLT processor to match the root element `cookbook` and then try to apply any available template to the child elements, i.e., chapter elements.

XSLT/Applying several templates (cont)

Since there is no template matching chapter elements, their child text nodes will be printed.

The result is now:

```
XSLT Cookbook
Salvatore Mangano
XPath
Selecting and Traversing
XML to Text
XML to XML
XML to HTML
```

XSLT/Applying several templates (cont)

What if we do not want to print the name of the book and the author's name?

The first solution consists in adding two templates matching the elements to be ignored and do nothing (empty template elements):

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

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

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

XSLT/Applying several templates (cont)

When several templates do the same thing (or do nothing), it is possible to write down only one template element with a match attribute containing all the element names to be matched.

For example, instead of

```
<xsl:template match="title"/>  
<xsl:template match="author"/>
```

one can write

```
<xsl:template match="title|author"/>
```

The symbol “|” means “or” and this kind of match attribute is called a **disjunctive pattern**.

XSLT/Selecting children to be matched

The second solution consists in specifying that only some child elements of a given context node should be matched against the available templates.

This **selection** of the required children is done by means of the select attribute of the apply-templates element.

The content of the attribute is the name of the children, relatively to the context node.

Therefore, the select attribute evaluates to a **sequence of nodes** (the children) to the matched in document order.

XSLT/Selecting children to be matched (cont)

Consider the following XSLT transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="cookbook">
    <xsl:apply-templates select="chapter"/>
  </xsl:template>
</xsl:transform>
```

XSLT/Selecting children to be matched (cont)

The result is now

XPathSelecting and TraversingXML to TextXML to XMLXML to HTML

Note that only the chapter titles have been printed, not the book title, nor the author's name. This is due to the selection of chapter elements only.

Note also that the text nodes of the selected chapters have been printed without spacing. This is due to having a sequence of nodes to apply templates to.

Let us improve the printing.

XSLT/Selecting children to be matched (cont)

Since we want to handle explicitly how the children of chapter are printed, we need to have a dedicated template matching chapter:

```
<xsl:template match="chapter">  
...  
</xsl:template>
```

Now, when the context node is chapter, we want to print its text node.

XSLT/Selecting children to be matched (cont)

The XSLT element that creates a text node is

`value-of`

This element, like `apply-templates` has a `select` attribute.

Since the text nodes which are children of `chapter` (there is only one by `chapter`, actually) are to be selected, we write

```
<xsl:template match="chapter">
  <xsl:value-of select="text()"/>
</xsl:template>
```

The expression `text()` denotes **all** the text nodes which are children of the context node `chapter`, i.e., the current chapter. The effect of `value-of` here is thus to duplicate the selected text node.

XSLT/Selecting children to be matched (cont)

Unfortunately, this changes nothing, because `apply-templates` serialises the results of the matchings as a sequence of strings with no inter-spaces.

The solution is thus to force a **new line** after getting the value of each text node.

This can be done by means of the element

```
<xsl:text>&#10;</xsl:text>
```

The numbered entity `
` corresponds to the new line character in the ASCII.

The XSLT element `text` can be used to write text verbatim.

It is useful when the complex rules of XML about space characters have to be bent.

XSLT/From XML to XHTML

One application of XML, server-side, is to generate XHTML or, in general, HTML.

Following a client's request for a web page, the web server retrieves its content as XML (stored in a database, for instance) and runs an XSLT transformation, depending perhaps on the client's user-agent (e.g., the kind of browser) or display device (e.g., a mobile device with a small screen or limited number of colours) to produce (perhaps dedicated) HTML, which is then sent back to the client.

This way, the content of the page (XML specifies semi-structured data, without a specific interpretation) is stored independently of any presentation style (HTML specifies both contents and style).

XSLT/From XML to XHTML (cont)

The first step when designing an XSLT stylesheet (i.e., transformation or transform) to produce XHTML is to write an instance of expected output.

Once the shape of the general output is clearly in mind, think backwards to the input and, by doing so, write the transformation.

Technically, the difficulty in the XSLT comes from managing the namespaces: one for the XSLT elements and one for the XHTML elements.

XSLT/From XML to XHTML (cont)

First, the start of the XSLT stylesheet must be

```
<xsl:output
  method="xhtml"
  doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
  doctype-system=
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
  indent="yes"/>
```

XSLT/From XML to XHTML/Example

Let us try to transform the XML page 95 into some XHTML displayed as

XSLT Cookbook

by Salvatore Mangano

1. XPath
2. Selecting and Traversing
3. XML to Text
4. XML to XML
5. XML to HTML

Also, the title of the browser window should be “XSLT Cookbook”.

XSLT/From XML to XHTML/Example (cont)

First, we must decide that the XHTML should look as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8"/>
    <title>XSLT Cookbook</title>
  </head>
  <body>
    <h2>XSLT Cookbook</h2>
    <p><em>by Salvatore Mangano</em></p>
```

XSLT/From XML to XHTML/Example (cont)

```
<h3>Table of contents</h3>
<ol>
  <li>XPath</li>
  <li>Selecting and Traversing</li>
  <li>XML to Text</li>
  <li>XML to XML</li>
  <li>XML to HTML</li>
</ol>
</body>
</html>
```

XSLT/From XML to XHTML/Example (cont)

Therefore, it becomes clear that the infrastructure of the XHTML, made of the XML processing instruction, XHTML elements `html`, `head` and `body`, should be output first.

Then, the title and author.

The last part being the ordered list of chapters.

XSLT/From XML to XHTML/Example (cont)

From this analysis, we deduce the following structure for the XSLT stylesheet.

- A template to match the root element, `cookbook`, is needed to output the XHTML elements `html`, `head`, `body` **and** the `title`, `author` **and** an *empty* XHTML element `ol`.
- Another template is needed to match the chapter elements and fill the empty XHTML element `ol` with the list items corresponding to the chapters.

XSLT/From XML to XHTML/Example (cont)

```
<xsl:template match="cookbook">
  <html xmlns="http://www.w3.org/1999/xhtml"
        xml:lang="en" lang="en">
    <head>
      <title><xsl:value-of select="title/text()"/></title>
    </head>
    <body>
      <h2><xsl:value-of select="title/text()"/></h2>
      <p>
        <em>by <xsl:value-of select="author/text()"/></em>
      </p>
      <h3>Table of contents</h3>
      <ol>
        <xsl:apply-templates select="chapter"/>
      </ol>
    </body>
  </html>
</xsl:template>
```

XSLT/From XML to XHTML/Example (cont)

Notice the select values “title/text()” and “author/text()”.

To understand the meaning of such expressions, one must decompose them step by step. Let us consider the first one as an example.

First, `title` as an expression means “all the child elements of the context node.” So, in general, this is a sequence of nodes, but here the sequence contains only one node.

Then `.../text()` means “all the text nodes which are children of **every** element in the previous sequence.” In this case, the `title` elements have only one text node as a child.

XSLT/Selecting children to be matched (cont)

In this case, it is possible to write

```
<xsl:value-of select="title"/>
```

instead of

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

The former means “**all** the text nodes descendant of the title element, which are the children of the context node.” But since there is only one text node below title, it means the same as the selection title/text().

XSLT/From XML to XHTML/Example (cont)

Let us resume our stylesheet. Another template is needed for the chapters:

```
<xsl:template match="chapter">
  <li xmlns="http://www.w3.org/1999/xhtml">
    <xsl:value-of select="text()"/>
  </li>
</xsl:template>
```

Note here the obligation to specify the XHTML namespace as default in the XHTML `li` element. Otherwise it will be output without namespace, instead of having the XHTML namespace implicitly. In other words, if one writes `` instead of

`<li xmlns="http://www.w3.org/1999/xhtml">`, the output will be `<li xmlns="">`, because the XHTML namespace is the default for the embedding `html` element.

XSLT/From XML to XHTML/Example (cont)

The last step consists in checking the conformity to XHTML by means of the W3C validator available at

<http://validator.w3.org/>

XSLT/Variables

Sometimes it is handy to define **variables** to hold some intermediary result. Consider

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
    <xsl:variable name="v" select="0"/>
    <xsl:variable name="w" select="1 + $v"/>
    <xsl:value-of select="$w"/>
  </xsl:template>
</xsl:transform>
```

XSLT/Variables (cont)

The match expression "/" means: "the document root".

The document root is not the root element.

The latter is the unique element at the root of the XML tree, e.g.,
cookbook is the root element of the document page 95.

The former is the implicit node that contains all the elements in the
XSLT file. It is not written.

Therefore, the root element is always a child of the document root and
"/" matches any XML document.

Also, "/*" matches any root element.

XSLT/Variables (cont)

The variable `v` is set to the selected value `0`. Note that it is possible to select basic types, like integers, not just nodes.

Also, as variable `w` demonstrates it, it is possible to select the contents of another variable by prefixing its name with the symbol `$`, and then operate on it as an arithmetic expression, e.g., `"1 + $v"`.

Important: Variables in XSLT are immutable.

In other words, their value cannot be changed. That explains why there is no assignment on variables, like `a = a + 1`; in C.

XSLT/Conditionals

It is possible to choose a sequence of nodes rather than another one, based on a given criterion.

The pure XSLT way to achieve this is by means of the `if`, `choose`, `when` and `otherwise` elements. Consider first the input

```
<?xml version='1.0' encoding='iso-8859-1'?>
<numbers>
  <num>18</num>
  <num>-1.3</num>
  <num>3</num>
  <num>5</num>
  <num>23</num>
</numbers>
```

XSLT/Conditionals (cont)

Let us output all the text nodes in order, **separated by a comma**. But the last string must not be followed by a comma, i.e., we expect

18, -1.3, 3, 5, 23

The solution is

```
<xsl:template match="num">
  <xsl:value-of select="text()"/>
  <xsl:if test="position() ne last()">
    <xsl:value-of select="', '"/>
  </xsl:if>
</xsl:template>
```


XSLT/Conditionals (cont)

Notice the XSLT element `if` and its `test` attribute. The value of this attribute is evaluated to either `true()` or `false()` — or a dynamic error happens. If the value is true, then the children of `if` are computed and outputted (here, a comma).

The comma must be written in a **string**, i.e., enclosed in single quotes: `' , '`.

The conditional expression is `position() ne last()`. The comparison operator is `ne`, which means “not equal”. The built-in function `position` returns the integer index of the context node in the sequence where it comes from (first node has index 1). Here, the context node corresponds to a `num` element in the sequence of all the children of the element `root numbers`.

XSLT/Conditionals (cont)

If one wants to see the explicit sequence of nodes `num`, suffices to write

```
<xsl:template match="numbers">  
  <xsl:apply-templates select="num"/>  
</xsl:template>
```

The other built-in function is `last`, which returns the last index of the context node in the original sequence it belongs to.

Thus, a comma is outputted if, and only if, the current number is not the last.

XSLT/Conditionals (cont)

Imagine now that we want

18, -1.3, 3, 5, 23.

That is to say, we want to terminate the comma-separated list by a period. We would need a `else` element, but it does not exist and a more general construct is needed: we can no longer use `if`.

XSLT/Conditionals (cont)

The solution is now:

```
<xsl:template match="num">
  <xsl:value-of select="text()"/>
  <xsl:choose>
    <xsl:when test="position() ne last()">
      <xsl:value-of select="', '/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="'. '"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

XSLT/Template parameters

Sometimes it is useful that templates be provided parameters.

Imagine that we have a document in XML describing part of the table of contents of a book, with chapter and section elements. Each chapter and section contains a first child title.

We want to output an unordered XHTML list of the chapter and section titles with a twist: the title should be annotated with the depth of the parent chapter or section. Chapters have depth 1, then child sections have depth 2 etc.

XSLT/Template parameters (cont)

An excerpt of the input:

```
<chapter>
  <title>Instances and schemas</title>
  <section>
    <title>Using the instance attributes</title>
  </section>
  <section>
    <title>Schema processing</title>
    <section>
      <title>Validation</title>
    </section>
    <section>
      <title>Augmenting the instance</title>
    </section>
  </section>
</chapter>
```

XSLT/Template parameters (cont)

The corresponding expected result:

- [1] Instances and schemas
 - [2] Using the instance attributes
 - [2] Schema processing
 - [3] Validation
 - [3] Augmenting the instance

XSLT/Template parameters (cont)

We need to pass down the tree the depth, output the titles at this depth (which are all siblings) and apply further templates on the children:

```
<xsl:template match="section|chapter">
  <xsl:param name="depth"/>
  <li xmlns="http://www.w3.org/1999/xhtml">
    <xsl:value-of select="concat('[', $depth, '] ', title)"/>
    <xsl:if test="not(empty(section))">
      <ul><xsl:apply-templates select="section">
        <xsl:with-param name="depth" select="$depth + 1"/>
      </xsl:apply-templates></ul>
    </xsl:if>
  </li>
</xsl:template>
```


XSLT/Template parameters (cont)

Note the new XSLT element `param` and its attribute `name`. It defines a parameter to the template.

The function `concat` takes an arbitrary number of arguments, evaluate them, transform them into strings and join the strings together, in order.

Note that the element `apply-templates` is not empty anymore: it contains a new XSLT element named `with-param`, which carries attributes `name` and `select`. This defines an argument, i.e., a sequence of nodes or basic types passed to the children.

The value of the `name` attribute must be the same as the one defined by the `name` attribute in the called template (with element `param`).

XSLT/Selecting nodes

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <entrees title="Starters">
    <dish id="1" price="8.95">Crab Cakes</dish>
    <dish id="2" price="9.95">Jumbo Prawns</dish>
    <dish id="3" price="10.95">Smoked Salmon</dish>
  </entrees>
  <main title="Main course">
    <dish id="4" price="19.95">Grilled Salmon</dish>
    <dish id="5" price="17.95">Seafood Pasta</dish>
    <dish id="6" price="16.95">Linguini al Pesto</dish>
  </main>
  <desserts title="Sweet End">
    <dish id="7" price="6.95">Dame Blanche</dish>
    <dish id="8" price="5.95">Chocolat Mousse</dish>
  </desserts>
</menu>
```

XSLT/Selecting nodes (cont)

XSLT allows the selection of any node in a sequence by putting the index of the node, following document order, between square brackets or using the `position()` function.

For example, to select the third dish in entrees:

`"/menu/entrees/dish[3]"` or `/menu/entrees/dish[position() eq 3]`.

Remember that the first node has always index 1.

This kind of notation is a special case of **predicate**, which we shall study more in detail later.

Exercise. Propose a transformation that prints Today's menu followed by the third *entree*, the first main dish and the last dessert, on different lines and preceded by a dash.

XSLT/Selecting attributes

Let us say that the dessert of day is the second on the menu page 140 and that we want to know its price. This information is stored as an attribute node, not as a text node, so we need a special construct @ as in the transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="menu">
    Dessert of the Day:
    <xsl:value-of select="desserts/dish[2]" />
    Price: <xsl:value-of select="desserts/dish[2]/@price" />
  </xsl:template>
</xsl:transform>
```

XSLT/Selecting attributes (cont)

The result is

Dessert of the Day:

Chocolat Mousse

Price: 5.95

Note that attributes of a given element must have different names, so selecting "@foo" results either in one or no attribute.

XSLT/Using wildcards

Wildcards can be used to select nodes whose names do not matter, e.g.,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="menu">
    Dessert of the Day:
    <xsl:value-of select="*[3]/dish[2]" />
    Price: <xsl:value-of select="*[3]/dish[2]/@[2]" />
  </xsl:template>
</xsl:transform>
```

gives the same result as the transform page 142.

XSLT/Location paths

A **location path** contains **test nodes** and, optionally, **predicates**. For example,

```
/menu/*[3]/dish[2]/@*[2]
```

is a location path; menu, *, dish and @* are node tests and [3] and [2] are predicates.

XSLT/*n*-th node

Let us try to select the sixth dish in the document using the transform

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="menu">
    -<xsl:value-of select="*/dish[6]"/> </xsl:template>
</xsl:transform>
```

The output contains no dishes! Why? Because value-of select no element... Indeed, the location path `*/dish[6]` does **not** yield the sixth dish node in the menu but, instead, the first dish in the menu which is the sixth child of any parent.

XSLT/*n*-th node (cont)

In other words, the expression `*/dish[6]` actually means `*/(dish[6])`, i.e. the sixth element in all parent contexts which are children of the menu element, thus there is no matched node. What we want is `(*/dish)[6]`, as in the transform

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="menu">
    -<xsl:value-of select="(*/dish)[6]"/>
  </xsl:template>
</xsl:transform>
```

XSLT/Inserting elements (cont)

Consider two documents with the same content but different structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car model="Matiz" manufacturer="Daewoo"/>
  <car model="Sonata" manufacturer="Hyundai"/>
</cars>
```

and

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car>
    <model>Matiz</model><manufacturer>Daewoo</manufacturer>
  </car>
  <car>
    <model>Sonata</model><manufacturer>Hyundai</manufacturer>
  </car>
</cars>
```

XSLT/Inserting elements (cont)

Using the same technique, we can make a transform that converts the first car list, i.e. using attributes, into the second one, i.e. using elements. Consider

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>

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

XSLT/Inserting elements (cont)

```
<xsl:template match="car">
  <car>
    <xsl:apply-templates select="attribute()"/>
  </car>
</xsl:template>
```

```
<xsl:template match="@model">
  <model>
    <xsl:value-of select="."/>
  </model>
</xsl:template>
```

XSLT/Inserting elements (cont)

```
<xsl:template match="@manufacturer">
  <manufacturer>
    <xsl:value-of select="." />
  </manufacturer>
</xsl:template>
</xsl:transform>
```

XSLT/Inserting elements (cont)

Notice the XSLT element value-of when matching an attribute:

```
<xsl:template match="@model">  
  ... <xsl:value-of select="."/> ...  
</xsl:template>
```

Selecting "." is the only way to get the attribute value, in particular `text()` does not work because attributes are different from elements.

For instance, the empty transformation does not print the attribute values because they are considered different from text nodes.

XSLT/Inserting elements (cont)

Applying it to the first car list page 148 yields

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car>
    <model>Matiz</model>
    <manufacturer>Daewoo</manufacturer>
  </car>
  <car>
    <model>Sonata</model>
    <manufacturer>Hyundai</manufacturer>
  </car>
</cars>
```

which is exactly the second car list (same page).

XSLT/Inserting elements (cont)

The name of the inserted elements was until now written in the transform, but it is possible to insert elements whose names are made at run-time. Consider

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="cars">
    <models><xsl:apply-templates/></models>
  </xsl:template>
  <xsl:template match="car">
    <xsl:element name="{@manufacturer}">
      <xsl:value-of select="@model"/>
    </xsl:element> </xsl:template>
</xsl:transform>
```


XSLT/Inserting elements (cont)

Notice that element `<xsl:element name="td">Matiz</xsl:element>` is equivalent to `<td>Matiz</td>`. Also, remember the braces around `@manufacturer` to select the attribute **value**: `{@manufacturer}`.

Then, the result of applying it to the first car list page 148 yields

```
<?xml version="1.0" encoding="UTF-8"?>
<models>
  <Daewoo>Matiz</Daewoo>
  <Hyundai>Sonata</Hyundai>
</models>
```

XSLT/Inserting elements (cont)

In the transform page 149 we had a template for each attribute (model and manufacturer).

It would be better to have only one template for all the attributes, since we process them in the same way, i.e., insert in the output an element with the same name as the attribute and with a text node whose contents is the same as the attribute value.

We need a function named `name` which returns the name of the context node when called as `name()`.

XSLT/Inserting elements (cont)

Consider

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="cars">
    <cars><xsl:apply-templates/></cars></xsl:template>
  <xsl:template match="car">
    <car><xsl:apply-templates select="attribute()"/></car>
  </xsl:template>
  <xsl:template match="attribute()">
    <xsl:element name="{name()}">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
</xsl:transform>
```

XSLT/Inserting elements (cont)

Applying it to the first car list page 148 yields the second car list:

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car>
    <model>Matiz</model>
    <manufacturer>Daewoo</manufacturer>
  </car>
  <car>
    <model>Sonata</model>
    <manufacturer>Hyundai</manufacturer>
  </car>
</cars>
```

XSLT/Inserting elements (cont)

We can even generalize the transform and make it work on any kind of document, by converting every attribute into an element of the same name. Consider

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="element(">
    <xsl:element name="{name()}">
      <xsl:apply-templates select="element()|attribute()"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="attribute(">
    <xsl:element name="{name()}">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
</xsl:transform>
```

XSLT/Inserting elements (cont)

Apply it to the first car list page 148 yields the second car list again:

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car>
    <model>Matiz</model>
    <manufacturer>Daewoo</manufacturer>
  </car>
  <car>
    <model>Sonata</model>
    <manufacturer>Hyundai</manufacturer>
  </car>
</cars>
```

XSLT/Inserting attributes

Inserting attributes is akin to inserting elements. Consider the straightforward

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="cars">
    <table border="1" width="500">
      <xsl:apply-templates/>
    </table>
  </xsl:template>
  <xsl:template match="car">
    <tr bgcolor="#dddddd">
      <xsl:apply-templates select="attribute()"/>
    </tr>
  </xsl:template>
  <xsl:template match="attribute()">
    <td> <xsl:value-of select="."/> </td>
  </xsl:template>
</xsl:transform>
```

XSLT/Inserting attributes (cont)

Instead of writing

```
<xsl:template match="car">
  <tr bgcolor="#dddddd">
    <xsl:apply-templates select="attribute()"/>
  </tr>
</xsl:template>
```

one can write

```
<xsl:template match="car">
  <tr>
    <xsl:attribute name="bgcolor">#dddddd</xsl:attribute>
    <xsl:apply-templates select="attribute()"/>
  </tr>
</xsl:template>
```


XSLT/Inserting attributes (cont)

Converting the second car list to the first one is thus achieved by the transform

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="cars">
    <cars><xsl:apply-templates/></cars> </xsl:template>
  <xsl:template match="car">
    <car><xsl:apply-templates/></car> </xsl:template>
  <xsl:template match="element()">
    <xsl:attribute name="{name()}">
      <xsl:value-of select="text()" />
    </xsl:attribute>
  </xsl:template>
</xsl:transform>
```

XSLT/Inserting attributes (cont)

Note that the last template matches "element()", which means, in general, "any element node."

The XSLT processor always select the first template matching the current node (element or attribute) most precisely, so templates must be ordered carefully.

Therefore, here, the template matching element() will match any element node whose name is neither car nor cars, because of the two first templates.

XSLT/Copying elements

Sometimes one wants to copy an element from the source document to the output.

One way is the so-called **shallow copy**, i.e. the *context node and its text nodes* are copied but neither the children element nodes nor the attribute nodes:

```
<xsl:copy> ... </xsl:copy>
```

or

```
<xsl:copy/>
```

In the latter, the context node is copied without modification, whilst, in the former, attributes and children elements can be added.

XSLT/Copying elements (cont)

Consider the simple template

```
<xsl:template match="car">  
  <xsl:copy/>  
</xsl:template>
```

When matching

```
<car>  
  <model>Matiz</model>  
  <manufacturer>Daewoo</manufacturer>  
</car>
```

produces

```
<car/>
```

XSLT/Copying elements (cont)

Consider the template

```
<xsl:template match="car">
  <xsl:copy>
    <xsl:attribute name="model">Matiz</xsl:attribute>
    <xsl:attribute name="manufacturer">Daewoo</xsl:attribute>
  </xsl:copy>
</xsl:template>
```

which, when matching the same element, now produces

```
<car model="Matiz" manufacturer="Daewoo">
```

XSLT/Copying elements (cont)

There is a way to copy the node and the whole subtree below it by using the XSLT element `copy-of`. Consider

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="cars">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:transform>
```

When applied to either the car lists page 148 yields the same document.

XSLT/Copying elements (cont)

If the selection in the copy-of element is a node set, *all the nodes and their subtrees will be copied verbatim to the output*. For example, the transform

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="cars">
    <xsl:copy>
      <xsl:copy-of select="car[1]|car[2]"/>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```

will also entirely copy the car list to the output.

XSLT/Copying elements (cont)

It is often useful to copy verbatim the source document *except some parts*. To achieve this, we cannot use copy-of. So, first, we need to define an identity transform and then modify it to take into account some exceptions. First try

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="element()">
    <xsl:copy>
      <xsl:apply-templates select="node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```


XSLT/Copying elements (cont)

Notice the function `node()`, allows the selection of the child element **and** child text nodes. (`element()` would select only the child element nodes, `text()` only the child text nodes, and `.` the context node.)

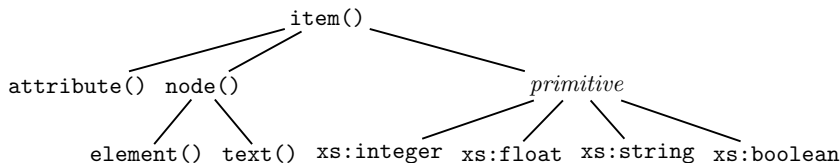
When applied to the second car list page 148, we get the same document, but when applied to the first car list (same page) it yields

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
  <car/>
  <car/>
</cars>
```

because `node()` does not match/select attribute nodes.

XSLT/Copying elements (cont)

The relationship between the different tests or types can be summarised in a tree:



This means, for example, that a node is either an element or a text node. The items are the most general data type in XSLT.

XSLT/Copying elements (cont)

Indeed, the previous transform is equivalent to the transform

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="element()">
    <xsl:copy>
      <xsl:apply-templates select="element()|text()"/>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```

XSLT/Copying elements (cont)

The way to extend our identity transform to include attributes is to match any attribute and, after matching an element node, to select any of its attributes. The identity transform is therefore

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="node()|attribute()">
    <xsl:copy>
      <xsl:apply-templates select="node()|attribute()"/>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```

XSLT/Copying elements (cont)

Now it is easy to override the identity template by adding new templates which are more precise, like matching a specific element name:

```
<xsl:template match="broken">
  <fixed>
    <xsl:apply-templates select="node()|attribute()"/>
  </fixed>
</xsl:template>
```

Now, all the elements will be copied verbatim, as well as the attributes, *except* elements named `broken`, which will be changed into `fixed`.

XSLT/Copying elements/Importing stylesheets

The typical idiom for copying with modification is to import the identity stylesheet and then add new templates which override the corresponding templates in the imported stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="identity.xsl"/>
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="broken">
    <fixed>
      <xsl:apply-templates select="node()|attribute()"/>
    </fixed>
  </xsl:template>
</xsl:transform>
```

XSLT/Copying elements (cont)

There is an alternative identity transform, using copy-of for the attributes only, thus allowing to be extended to handle special cases for element nodes:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="node()">
    <xsl:copy>
      <xsl:copy-of select="attribute()"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```

XSLT/Functions

It is possible to define functions in XSLT.

As an example, consider the need to debug a transform. The usual technique consists in using an XSLT element message, as in

```
<xsl:message>This is a debug message.</xsl:message>
```

One may print the value of attributes or text nodes:

```
<xsl:message>  
  Price: <xsl:value-of select="@price"/>  
</xsl:message>
```

What if the programmer wants to know the names of all the elements in a sequence? She needs a function!

XSLT/Functions/Namespace

The first thing to do, when defining or using functions is to declare the namespace to which they belong. For the purpose of this simple example, let us imagine that the namespace is named `my`, at a dummy URL:

```
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:my="file://functions.uri">
```

A name must be found. For instance, `my:names`. Note and remember that **the function name contains the namespace**. The definition is made in the XSLT element function:

```
<xsl:function name="my:names">
...
</xsl:function>
```

XSLT/Functions/Parameters

The function here takes one argument, the sequence of nodes, so we need to declare a parameter. This is the purpose of the XSLT element `param`:

```
<xsl:function name="my:names">
  <xsl:param name="nodes"/>
  ...
</xsl:function>
```

As usual in other programming languages, the parameter must be named.

If a function requires more parameters, other `param` elements are added.
The order is significant.

XSLT/Functions/Recursivity

We need to write a recursive function, because it needs to access all the nodes in a sequence. The idea is to check first whether the sequence is empty or not. If empty, do nothing. Otherwise, print the name of the first node and call again the function on the remaining ones (maybe none, it does not matter at that moment):

```
<xsl:function name="my:names">
  <xsl:param name="nodes"/>
  <xsl:if test="not(empty($nodes))">
    <xsl:value-of select="(name($nodes[1]),
                          my:names($nodes[position()>1]))"/>
  </xsl:if>
</xsl:function>
```

XSLT/Sequences revisited

Remark the notation in the select attribute (`... , ...`). It means “Create a sequence by appending the first sequence to the second sequence.” This operation is called **concatenation**.

Here `my:names($nodes[position()>1])` is a sequence of names, possibly empty. If empty, it disappear from the concatenation, e.g., `(1,(),2)` eq `(1,2)`.

Here, also, `name($nodes[1])` is a sequence of one element, because there is no difference, in XSLT, between a sequence of one and the item itself.

The concatenation is flat, so `((1,2,3),(4,5,(6,7),8,9,10))` is the same as `(1,2,3,4,5,6,7,8,9,10)`. The built-in function `empty` tests if a sequence is empty, e.g., `empty($seq)`.

XSLT/Named templates

It is possible to call a template by name instead of it being applied when matching an element. Consider the transform

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="car">
    <xsl:call-template name="the_car"/>
  </xsl:template>
  <xsl:template name="the_car">
    Manufacturer: <xsl:value-of select="@manufacturer" />
    Model:       <xsl:value-of select="@model" />
  </xsl:template>
</xsl:transform>
```

XSLT/Named templates (cont)

Named template and matching templates never interfere with each other. For example, the name can be the same as an element name, as in

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="car">
    <xsl:call-template name="car"/>
  </xsl:template>
  <xsl:template name="car">
    Manufacturer: <xsl:value-of select="@manufacturer" />
    Model:        <xsl:value-of select="@model" />
  </xsl:template>
</xsl:transform>
```

XSLT/Modes

It is sometimes useful to traverse the same document several times and match the same elements in a different way each time. This can be done using templates with modes.

One needs to define a template element with a `match` attribute and a `mode` attribute. The value of the latter can be any string with the condition that two templates with the same `match` attribute value have a different `mode` attribute value.

XSLT/Modes (cont)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
    <xsl:text>INDEX</xsl:text>
    <xsl:apply-templates mode="index"/>
    <xsl:text>INFO</xsl:text>
    <xsl:apply-templates mode="info"/>
  </xsl:template>
```


XSLT/Modes (cont)

```
<xsl:template match="car" mode="index">
  <xsl:value-of select="@model" />
</xsl:template>
<xsl:template match="car" mode="info">
  <xsl:text>Model: </xsl:text>
  <xsl:value-of select="@model" />
  <xsl:text>, Manufacturer: </xsl:text>
  <xsl:value-of select="@manufacturer"/>
</xsl:template>
</xsl:transform>
```

XSLT/Modes (cont)

The result of applying it to the first car list page 148 is

INDEX

Matiz

Sonata

INFO

Model: Matiz, Manufacturer: Daewoo

Model: Sonata, Manufacturer: Hyundai

Notice the two different runs on the input, distinguished by two different outputs in sequence.