http://webdam.inria.fr

Web Data Management

XSLT

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

`http://webdam.inria.fr/Jorge/`

Contents

XML is used in a large number of applications, which are either data-centric (semi-structured databases), or document-centric (Web publishing). In either case, there is a need for transforming documents from one XML formalism to another, or from one XML formalism to another kind of formalism (mostly HTML or text-based formats). This can of course be achieved in a traditional programming language, with any library for parsing XML documents (e.g., DOM, SAX) but this tends to be tedious. A declarative approach, more readable, more concise, and more adapted to a public of non-programmers would be welcome. This is the role of XSLT.

XSL (or *eXtensible Stylesheet Language*) is an initiative of the World Wide Web Consortium, originally planned as a language for the formatting of XML documents. It has been then split into a presentation format for printed text (XSL-FO, or XSL-Formatting Objects), and a transformation language for XML (XSLT, or XSL-Transformations), which is now used in a much broader context. XSLT is a declarative, side-effect–free, language for transforming XML documents into text, HTML, or XML. It heavily relies on the XPath expression language for selecting nodes in the XML tree. The focus of this chapter is on XSLT 1.0, a 1999 W3C Recommandation which makes use of XPath 1.0. XSLT 2.0, based on XPath 2.0, is briefly discussed in Section 3.3.

We first present the most important aspects of XSLT, and in particular its execution model, in Section 1. Understanding this section is a prerequisite to grasp the specific mechanism of XSLT evaluation, the rest being closer to traditional (functional) programming. We then discuss more advanced features in Section 2, without attempting to completely cover the language. Section 3 provides reference material and evokes some limitations of XSLT 1.0 and the possibilities to overcome them with XSLT 2.0.

1 A First Look at XSLT

An *XSLT program* (or *XSLT stylesheet*, or just *stylesheet*) is built out of rules called *templates*. When executed, the stylesheet takes an XML document as input, and combines the application of its templates to produce another XML document as output. The use of the terms *stylesheet* and *template* instead of the more commonly used (in programming languages) *program* and *rule* come from the origin (and the still important use) of XSLT as a formatting language for XML documents.

In order to master the creation of XSLT stylesheets, an absolute initial requirement is to clearly understand the role of templates, and the "execution model" of XLST which determines how templates are applied to the input document to produce an output. We devote the largest part of this section to these two aspects. Once they are clarified, the rest of the language is much easier to understand, partly because many of its other features are close to traditional programming languages.

1.1 What is a Template?

The specification of a template consists of two parts:

1. an XPath expression, called *pattern*, that indicates the kind of input nodes this template applies to;
2. a (well-formed!) XML fragment, called *body*, that represents the part of the output document which is created when the template is *instantiated*.

Following the long-standing tradition of introducing programming languages, we shall illustrate our first XSLT template with a "Hello world" example:

```
<xsl:template match="/">
  <hello>world</hello>
</xsl:template>
```

An incident comment is that the syntax of XSLT is based on XML: an XSLT program is itself an XML document. This should not affect our understanding of the meaning of templates (and other XSLT aspects). On the present example, this meaning is

> On meeting the document root in the *input* document, insert the XML fragment `<hello>world</hello>` in the *output* document.

Syntactically, the template is specified as an `xsl:template` element. The pattern is the value of the `match` attribute, and the body is the content of `<xsl:template>`.

The pattern is, here, the XPath expression `/` that denotes the document root of the input document. The role of the XPath expression of the `match` attribute is quite specific: it describes the nodes which can be the target of a template instantiation. Those expressions must comply to some restrictions and thus constitute a sub-class of of XPath called *patterns*. Informally, these restrictions are motivated by the following concerns:

- *a pattern always denotes a node set* (thus, `<`**xsl:template** `match='1'>` is incorrect).

- *it must be easy to decide whether a node is denoted or not by a pattern;* for instance although the value of `match` in <**xsl:template** `match='preceding::*[12]'`> is a meaningful XPath expression, it is quite difficult to evaluate and is therefore forbidden as a pattern.

These motivations lead to a rather drastic restriction of XPath expressiveness. A *pattern* is a valid XPath expression which uses only the `child` and `@` axes, and the abbreviation `//`. Predicates are allowed.

We now turn our attention to the *body* of a template. Our "Hello world" example showed that it may consists of well-formed XML content, including text and tags. (Note that, if the body were not a well-formed XML fragment, the XSLT program itself would not be a well-formed XML document, and its parsing would fail!) We may further distinguish *literal* tags and XSLT tags. Consider the slightly more complicated example below:

```
<xsl:template  match="book">
   <h2>Description</h2>

  The book title is:
     <xsl:value-of select="title" />
</xsl:template>
```

This template specifies a rule that is instantiated when a <book> node is met is the input document. The body (content of the <**xsl:template**> element) becomes part of the output document. However, on this second example, we must distinguish the literal tag <h2> from the XSLT tag <**xsl:value-of**>. The former is copied to the output document as it. The latter belongs to the XSLT namespace, and is therefore interpreted as an instruction. In this specific case, this interpretation is as follows: evaluate the node set result of the (relative) XPath expression `title`, and substitute this node set to the <**xsl:value-of**> instruction. So, if the instantiation of the template initially produces the following fragment in the output document:

```
   <h2>Description</h2>

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

the complete execution of the XSLT program result in the following final fragment (assuming an input related to the present book):

```
   <h2>Description</h2>

  The book title is:
     "Web Data Management and Distribution"
```

This shows that an XSLT program is not limited to the combination of static XML fragments but can also insert in the result values extracted from the input document (the language would not be very useful otherwise). Generally speaking, the body of a template may contain elements that belong to the XSLT namespace, interpreted as instructions that essentially

describe navigation and extraction operations applied to the input document.

1.2 The Execution Model of XSLT

The execution model of XSLT specifies how templates are instantiated. The basic mechanism is as follows: given a node (called "context node") in the input document, one selects with an XPath expression (relative to the context node) a node set. For each node in this node set, the XSLT processor chooses and instantiates a template. We are going to detail each of these concepts, but first note that the process is indirect: one does not choose the template that must be instantiated, but rather designates some node, and lets the processor finds the appropriate template. This indirect approach is probably what makes XSLT programming difficult to understand for the novice. We shall spend some time to carefully decompose the execution process, taking as example the XML document of Figure 1.

```
<book>
...
 <authors>
  <name>Serge</name>
  <name>Ioana</name>
 </authors>
</book>
```

Figure 1: Example document

The main XSLT instruction related to the execution model is <**xsl:apply−templates**>. It expects a `select` attribute that contains an XPath expression which, if relative, is interpreted with respect to the context node. This attribute is optional: the default value is `child::node()`, i.e., select all the children of the context node. Here is an example:

```
<xsl:apply-templates select="authors/name" />
```

This instruction can be put anywhere in the body of a template. A practical example of use of <**xsl:apply−templates**> is given in Figure 2

```
<xsl:template match="book">
  <ul><xsl:apply-templates
          select="authors/name" /></ul>
</xsl:template>

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

Figure 2: Two example templates, one instantiating the other

Assume now that the context node is the <book> element of the example document of Figure 1. The first template of Figure 2 is instantiated, because its pattern book matches the context node (so far, we adopt an intuitive meaning to the concept of "pattern matching" in XSLT: we give a formal definition at the end of this section). The body of this first template becomes part of the output, and the processor further considers the XSLT instructions contained in the body. One of these instructions is <**xsl:apply−templates** select='authors/name'>. Its interpretation is as follows:

1. the XPath expression authors/name is evaluated relatively to the context node; one obtains (see Figure 1) a node set with two author's <name>;
2. the processor searches for template(s) whose pattern matches the <name> nodes: the second template of Figure 2 is instantiated.

That is all there is to it. Once the <**xsl:value−of**> instruction of this second template has been evaluated in turn, one obtains the final result shown in Figure 3.

```
<ul>
  <li>Serge</li>
  <li>Ioana</li>
</ul>
```

Figure 3: Result of applying the templates from Figure 2 to the document of Figure 1

You are invited to carefully review this example, as it covers the core concepts of XSLT template execution. In particular, an essential feature is that (relative) XPath expressions contained in a template body are always interpreted witht respect to the context node, i.e., the node for which the template has been instantiated. Initially, the context node is the *document root* of the XML input document. The matching template principle is applied to this node: the processor looks for a template with pattern /. This template may in turn change the context node through calls to apply-templates, and initiate a traversal of the input document.

It remains, as promised, to precisely define the semantics of the pattern matching process.

Definition 1.1 Given an XML tree T, a pattern P *matches* a node N if there exists a node C (the *context node*) in T such that $N \in P(T, C)$.

In words: a node N matches a pattern P if N belongs to the result of P interpreted with respect to some node of the document. Here are a few examples of patterns to illustrate the idea:

- <**xsl:template** match='B'> pattern B matches any B element: it suffices to let C be the parent of B;
- <**xsl:template** match='A/B'> applies to any B element, child of an A element: take the parent of A as C, and the matching condition holds;
- <**xsl:template** match='@att1'> applies to any att1 attribute, whatever its parent element (take this parent as C);

- <**xsl:template** `match='A//@att1'`> applies to any `att1` attribute, if its parent element is a descendant of an `A` element (take the parent of `A` as *C*).

It should be noted that the first two templates in the list above are candidates for being instantiated for a `B` context node. In general, there might be more than one possible candidate template, or none. In case of many candidates, a precedence rule applies, whose driving idea is that the more specific pattern takes priority. On the present example, pattern `A/B` is more restrictive (hence, more specific) than pattern `B`, and the second template will be instantiated. In the case where no candidate template can be found, the processor applies some default "built-in" template, to be described next. A well-designed XSLT program should however favor clarity and simplicity in the organization of its templates, so that in most case one and only one template clearly applies to a node of the input document.

1.3 XSLT Stylesheets

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

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

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    <hello>world</hello>
  </xsl:template>

</xsl:stylesheet>
```

Figure 4: Hello World XSLT Stylesheet

An original characteristic of XSLT is that an XSLT stylesheet is itself an XML document, whose root element is named `stylesheet` (or, equivalently, `transform`) and belongs to the XSLT namespace `http://www.w3.org/1999/XSL/Transform`. All other XSLT element names belong to this namespace, which is usually referred to by the namespace prefix `xsl:`. A first complete (yet minimalistic) XSLT stylesheet is shown in Figure 4, with a single "Hello world" template. It illustrates the general structure of a stylesheet:

- a top-level <**xsl:stylesheet**> element, with a `version="1.0"` attribute (and, of course, the definition of the `xsl` namespace prefix);
- some *declarations* (all elements except <**xsl:template**> ones), in this case just an <**xsl:output**>, which specifies to generate an XML document, in the UTF-8 encoding (generating XML is the default: the corresponding option could be omitted);
- the list of *template rules*.

An XSLT stylesheet may be invoked either *programmatically*, through one of the various XSLT libraries, through a *command line* interface, or in a Web publishing context, by including a styling processing instruction in the XML document. This processing instructions specifies the stylesheet that must be applied to the current document, and must be put before the element root, as shown below:

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

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

<doc>
  <content />
</doc>
```

When an XML document containing such a processing instruction is loaded in a Web browser, the internal XSLT processor of the browser carries out the transformation and presents the output document in its window. In the case of Web publishing, XSLT stylesheets are designed to produce (X)HTML documents as result of the transformation, and the result can be shown right away to the user by the browser.

1.4 A Complete Example

We conclude this introductory part with a complete example of an XSLT stylesheet, along with detailed comments on its execution. The input document (Figure 5) contains a (very partial) description of the current book. We aim at transforming this content in XHTML.

```
<?xml version="1.0"
    encoding="utf-8"?>
<book>
  <title>Web [...]</title>
  <authors>
    <name>Serge</name>
    <name>Ioana</name>
  </authors>
  <content>
    <chapter id="1">
      XML data model
    </chapter>
    <chapter id="2">
     XPath
    </chapter>
  </content>
</book>
```

Figure 5: Input document to be transformed in XTML

Here is the full XSLT stylesheet. It consists of three templates. The first one (“Main template”) applies to the document root of the input document (note the pattern /) and

produces the skeleton of a valid XHTML document, with the <html>, <head> and <body> elements. The content of the body is simply an <**xsl:apply−templates**> XSLT instruction. The goal is to replace, at run time, this instruction by the result of its evaluation in order to obtain a content of the XHTML result that reflects the transformed content of the input document.

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

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

  <xsl:output method="xml" encoding="utf-8" />

<!-- Main template -->
<xsl:template match="/">
 <html>
   <head>
     <title>
      <xsl:value-of select="/book/title"/>
     </title>
    </head>
    <body bgcolor="white">

     <xsl:apply-templates select="book"/>

    </body>
  </html>
</xsl:template>

<!-- Book template -->
<xsl:template  match="book">
  The book title is:
     <xsl:value-of select="title" />

   <h2>Authors list</h2>
    <ul>
      <xsl:apply-templates select="authors/name" />
    </ul>
</xsl:template>

<!-- Author template -->
<xsl:template match="authors/name">
 <li><xsl:value-of select="."/></li>
</xsl:template>

</xsl:stylesheet>
```

We now examine how the XSLT processor executes this stylesheet over the document of Figure 5. The initial context node is the document root, so the processor seeks for a template whose pattern matches this node. The main template, with its pattern /, is the obvious unique candidate. It is therefore instantiated, on the output document is initialized with the

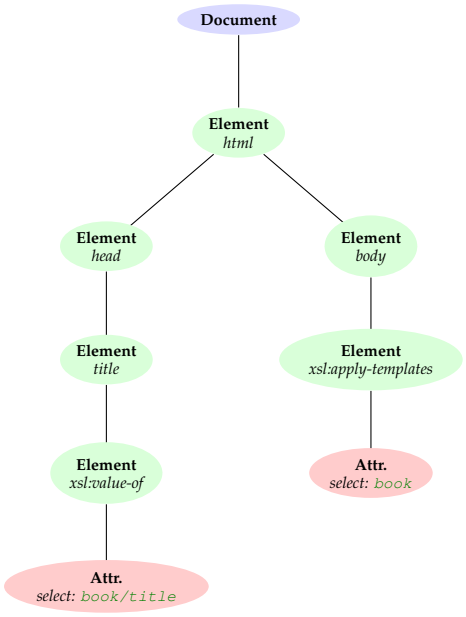

Figure 6: The tree representation of the output document, after instantiation of the first template

template's body. The tree view of this initial content is displayed in Figure 6.

The processor needs to consider the XSLT instructions that remain in the output document. Let us focus on the <**xsl:apply−templates**> one. Figure 6 shows that the context node for this instruction is the node that triggered the instantiation of the root: `/`. Any XPath expression is interpreted relative to this context node. In particular, the expression `book` yields a singleton node set with the root element of the input document, `/book`. The processor looks for a candidate template, find the second one in the stylesheet, and instantiates it. The body of this templates replaces the <**xsl:apply−templates**> instruction, and we obtain an extended output document shown in Figure 7. Note that each XSLT instruction remaining in this intermediate result is labelled with the context node that triggered its instantiation, i.e., `<book>`.

The next step is an evaluation of the XPath expression `author/name`, which yields a node set with two nodes (refer to the document of Figure 5). For each of these nodes, the processor picks the proper template. In this case (but this is not mandatory), the same template (the third one in the stylesheet) is instantiated twice, and one obtains the intermediate result of Figure 8. It is important to note here that, although these instantiations seem quite similar, they actually differ by their context node, which are repectively the first and second node `<name>` of the input document.

We let the reader interpret the last steps of the XSLT evaluation, that leads to the final result of Figure 9. Note that this is a valid XHTML content, where every XSLT instruction has been executed and replaced by its result.

2 Advanced Features

The mechanism presented at length in what precedes is sufficient for transforming simple documents. It is complemented by many data manipulation operators, programming in-

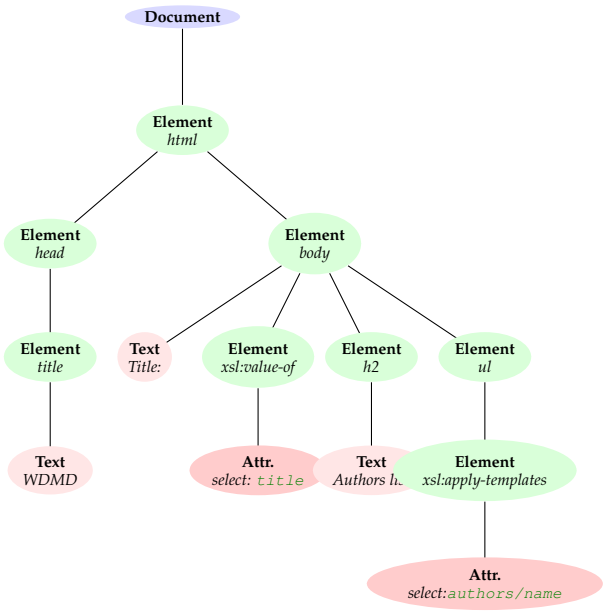

Figure 7: After instantiation of the second template

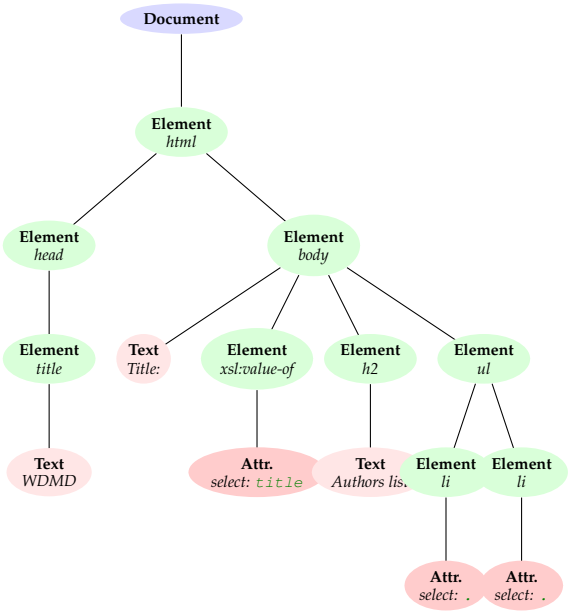

Figure 8: The third template is instantiated twice, but with distinct context nodes

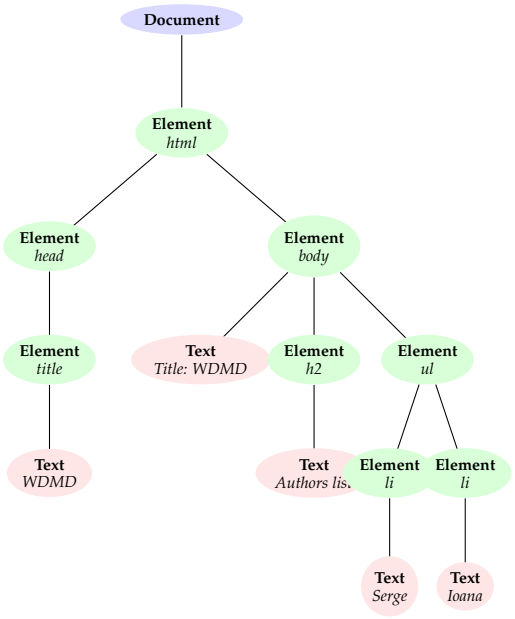

Figure 9: Final result

structions and built-in or external functions that make XSLT a quite powerful language. We briefly introduce the more important features, but this presentation is by no means complete. The reader seeking for a full presentation of XSLT is invited to consult the last section of this chapter for references.

2.1 Declarations

The following XSLT elements are *declarations*, that is, they appear as children of the top-level <**xsl:stylesheet**> element, and are detailed further on:

xsl:import Import the templates of an XSLT program, with low priorities

xsl:include Same as before, but no priority level

xsl:output Gives the output format (default: `xml`)

xsl:param Defines or imports a parameter

xsl:variable Defines a variable.

Other declarations include `xsl:strip-space` and `xsl:preserve-space`, for, respectively, removal or preservation of blank text nodes. They are discussed in Section 2.3.

Let us begin with <**xsl:output**>, an elaborated example of which is given in Figure 10. This XSLT element is used to control the final serialization of the generated document. It expects the following attributes (all are optional, as the <**xsl:output**> element itself):

- `method` is either `xml` (default), `html` or `text`.
- `encoding` is the desired encoding of the result.

```
<xsl:output
  method="html"
  encoding="iso-8859-1"
  doctype-public="-//W3C//DTD HTML 4.01//EN"
  doctype-system="http://www.w3.org/TR/html4/strict.dtd"
  indent="yes"
/>
```

Figure 10: Example of use of <**xsl:output**>

- `doctype-public` and `doctype-system` make it possible to add a document type declaration in the resulting document. Note that document type declarations are not part of the XPath document model and, as such, cannot be tested or manipulated in the source document in XSLT.
- `indent` specifies whether the resulting XML document will be indented (default is `no`).

An important feature of XSLT is *modularization*, the possibility of reusing templates across stylesheets. Two XSLT elements control this reuse. With <**xsl:import**>, template rules are imported this way from another stylesheet:

```
<xsl:import href="lib_templates.xsl" />
```

The *precedence* of imported templates is less that that of the local templates. Note that <**xsl:import**> must be the *first* declaration of the stylesheet. With <**xsl:include**>, template rules are included from another stylesheet. No specific precedence rule is applied, that is, templates work as if the included templates were local ones.

Preferences, priorities and built-in rules

These elements raise the more general question of solving conflicts between different templates that match the same node. The following principles are applied:

- Rules from imported stylesheets are *overridden* by rules of the stylesheet which imports them.
- Rules with *highest priority* (as specified by the `priority` attribute of <**xsl:template**>) prevail. If no priority is specified on a template rule, a default priority is assigned according to the *specificity* of the XPath expression (the more specific, the highest).
- If there are still conflicts, it is an error.
- If no rule applies for the node currently processed (the document node at the start, or the nodes selected by a <**xsl:apply−templates**> instruction), *built-in* rules are applied.

Built-in rules are important to understand the behavior of an XSLT stylesheet when the set of templates provided by the programmer is not complete. A first built-in rule applies to the **Element** nodes and to the document root node,

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

This can be interpreted as a recursive attempt to apply templates to the children of the context node. A second built-in rule applies to **Attribute** and **Text** nodes. This rule copies the textual value of the context node to the output document.

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

It follows that an *empty* XSLT stylesheet, when applied to an input document, triggers a recursive top-down traversal of the input tree. This traversal *ignores* attribute nodes. When the input leaves (text nodes) are finally met, their textual content is copied to the output document. The final result is therefore the concatenation of the textual content of the input document, without any markup. Incidently, this result is not a well-formed XML document, and a browser would fail to display it.

Variables and parameters

The last two declarations discussed here, <**xsl:param**> and <**xsl:variable**> are used to declare global parameters and variables, respectively.

```
<xsl:param name="nom" select="'John Doe'" />
<xsl:variable name="pi" select="3.14159" />
```

Global parameters are passed to the stylesheet through some *implementation-defined* way. The `select` attribute gives the default value, in case the parameter is not passed, as an XPath expression. Global variables, as well as local variables which are defined in the same way inside template rules, are immutable in XSLT, since it is a side-effect-free language. The `select` content may be replaced in both cases by the content of the <**xsl:param**> or <**xsl:variable**> elements.

2.2 XSLT Programming

We discuss here various XSLT elements that make of XSLT a regular, Turing-complete, programming language.

Named templates

Named templates play, in XSLT, a role analogous to functions in traditional programming languages. They are defined and invoked with, respectively, an <**xsl:template**> with a `name` parameter, and a <**xsl:call-template**> element, as shown in Figure 11. A call to a named template does not change the context node.

```
<xsl:template name="print">
  <xsl:value-of select="position()"/>:
     <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="*">
 <xsl:call-template name="print"/>
</xsl:template>

<xsl:template match="text()">
 <xsl:call-template name="print"/>
</xsl:template>
```

Figure 11: Example of use of named templates

```
<xsl:template name="print">
  <xsl:param name="message" select="'nothing'"/>

  <xsl:value-of select="position()"/>:
      <xsl:value-of select="$message"/>
</xsl:template>

<xsl:template match="*">
 <xsl:call-template name="print">
   <xsl:with-param name="message"
           select="'Element node'"/>
 </xsl:call-template>
</xsl:template>
```

Figure 12: Example of use of parameters in named templates

Named templates can also have *parameters* which are declared with the same <**xsl:param**> element that is used to define global parameters, and are instantiated, when the template is invoked, with a <**xsl:with−param**> element, as the example in Figure 12 shows. Named templates, with parameters, can be used recursively to perform computations. This is actually often the only way to perform an iterative process, in the absence of mutable variables in XSLT. Thus, the factorial function can be computed in XSLT with the (quite verbose!) named template of Figure 13. It makes use of the conditional constructs that are detailed next. Note that regular (unnamed) templates can also have parameters.

```
<xsl:template name="factorial">
  <xsl:param name="n" />

  <xsl:choose>
    <xsl:when test="$n&lt;=1">1</xsl:when>
    <xsl:otherwise>
      <xsl:variable name="fact">
        <xsl:call-template name="factorial">
          <xsl:with-param name="n" select="$n - 1" />
        </xsl:call-template>
      </xsl:variable>
      <xsl:value-of select="$fact * $n" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figure 13: Factorial computation in XSLT

Conditional constructs

Two conditional constructs can be used in XSLT: <**xsl:if**> when a part of the production of a template body is conditioned, and <**xsl:choose**>, when several alternatives lead to different productions.

```
<xsl:template match="Movie">
  <xsl:if test="year &lt; 1970">
    <xsl:copy-of select="."/>
  </xsl:if>
</xsl:template>
```

Figure 14: Example of use of <**xsl:if**>

Consider the example of use of <**xsl:if**> given in Figure 14. It also makes use of the <**xsl:copy−of**> element, that copies the nodes of the source document into the resulting document in a recursive way. A similar <**xsl:copy**> construct can be used to copy elements without their descendants. The `test` attribute of the <**xsl:if**> element is a Boolean expression that

conditions the production of the content of the element. (Recall also that an XSLT program is an XML document: we must use entities for literals <, >, &.)

```
<xsl:choose>
  <xsl:when test="$year mod 4">no</xsl:when>
  <xsl:when test="$year mod 100">yes</xsl:when>
  <xsl:when test="$year mod 400">no</xsl:when>
  <xsl:otherwise>yes</xsl:otherwise>
</xsl:choose>

<xsl:value-of select="count(a)"/>
<xsl:text> item</xsl:text>
<xsl:if test="count(a)>1">s</xsl:if>
```

Figure 15: Example of use of both <**xsl:choose**> and <**xsl:if**>

The other conditional construct, <**xsl:choose**>, is illustrated in Figure 15 along with <**xsl:if**>, and is roughly the equivalent of both `switch` and `if ... then ... else` instructions in a programming language like C (while <**xsl:if**> plays the same role as a `if ... then` without an `else`). Inside an <**xsl:choose**>, <**xsl:otherwise**> is optional. There can be any number of <**xsl:when**>, only the content of the first matching one will be processed.

Loops

```
<xsl:template match="person">
[...]
  <xsl:for-each select="child">
    <xsl:sort select="@age" order="ascending"
              data-type="number"/>

    <xsl:value-of select="name" />
    <xsl:text> is </xsl:text>
    <xsl:value-of select="@age" />
</xsl:for-each>
[...]
</xsl:template>
```

Figure 16: Example of use of <**xsl:for-each**>

<**xsl:for-each**>, an example of which is given in 16, is an instruction for looping over a set of nodes. It is more or less an alternative to the use of <**xsl:template**>/<**xsl:apply-templates**>. The set of nodes is obtained with an XPath expression (attribute `select`), and each node of the set becomes in turn the *context node* (which temporarily replaces the template context node). The body of <**xsl:for-each**> is instantiated for each context node. As there is no need to call another template, the use of <**xsl:for-each**> is somewhat simpler to read, and likely to be more efficient. On the other hand, it is much less subject to modularization than the use of separate

templates. <**xsl:sort**>, which can also be used as a direct child of an <**xsl:apply−templates**> element, is optional and controls the sorting of the sequence of nodes which is iterated on.

Variables in XSLT

A (local or global) variable is a $(name, value)$ pair. It may be defined either as the result of an XPath expression

```
<xsl:variable name='pi'ï¿½select='3.14116'/>
```

or as the content of the <**xsl:variable**> element,

```
<xsl:variable name='children'ï¿½select='//child'/>
```

A variable has a *scope* (all its siblings, and their descendants) and *cannot* be redefined within this scope. It is thus really the equivalent of a constant in classical programming languages. Recursive use of a template is the only way for a variable or a parameter to vary, for each different instantiation of the template.

2.3 Complements

There are many more features of the XSLT language, including control of text output with <**xsl:text**>, <**xsl:strip−space**>, <**xsl:preserve−space**>, and `normalize-space` function; dynamic creation of elements and attributes with <**xsl:element**> and <**xsl:attribute**>; multiple document input and output with the `document` function and `<xsl:document>` element (XSLT 2.0, but widely implemented in XSLT 1.0 as an extension function); and generation of hypertext documents with links and anchors (`generate-id` function). We describe some of them here, but the best way to discover the possibilities of XSLT is through practice (see in particular the associated exercises and Chapter **??**).

Handling whitespace and blank nodes

The two following rules are respected in XSLT regarding whitespace and especially blank nodes (that is, nodes that only consist of whitespace):

- All the whitespace is kept in the input document, *including* blank nodes.
- All the whitespace is kept in the XSLT program document, *except* blank nodes. As this rule only apply to whitespace-only node, and not to nodes starting or ending with whitespace, it is often useful to enclose **Text** nodes in an <**xsl:text**> whose sole purpose is to create a **Text** node.

Whitespace in the source document can also be handled explicitly, through <**xsl:strip−space**> and <**xsl:preserve−space**>, as shown in Figure 17. <**xsl:strip−space**> specifies the set of nodes whose whitespace-only text child nodes will be removed (or $\star$ for all), and <**xsl:preserve−space**> allows for exceptions to this list.

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

Figure 17: Handling whitespace explicitly

```
<xsl:element name="{concat('p',@age)}"
  namespace="http://ns">
  <xsl:attribute name="name">
    <xsl:value-of select="name" />
  </xsl:attribute>
</xsl:element>
```

Figure 18: Dynamic creation of elements and attributes

Dynamic Elements and dynamic attributes

Creation of element and attribute nodes whose names are not known at the writing of the stylesheet is possible through the use of <**xsl:element**> and <**xsl:attribute**>, as shown in the self-explanatory example of Figure 18, that transforms the following document

```
<person age="12">
  <name>Doe</name>
</person>
```

into the document

```
<p12
  name="Doe"
  xmlns="http://ns" />
```

The value of the `name` attribute is here an *attribute template*: this attribute normally requires a string, not an XPath expression, but XPath expressions between curly braces are evaluated. This is often used with literal result elements: `<foo val="{$var + 1}"/>`. Literal curly braces must be doubled. Attribute templates can be used in similar contexts, where a literal string is expected.

Working with multiple documents

document($s) returns the *document node* of the document at the URL `$s`. Note that `$s` can by computed dynamically (e.g., by an XPath expression). The result can be manipulated as the root node of the returned document.

```
<xsl:template match="Person">
  <h2 id="{generate-id(.)}">
    <xsl:value-of
      select="concat(first_name, ' ', last_name)"/>
  </h2>
</xsl:template>
```

Figure 19: Use of `generate-id` to generate link anchors

Generating unique identifiers

generate-id($s) (cf Figure 19) returns a *unique identifier string* for the first node of the nodeset `$s` in document order. This is especially useful for testing the identity of two different nodes, or to generate HTML anchor names.

3 Further Reading

XSLT 1.0 and XSLT 2.0 are both W3C recommendations and can be found at the following URLs: `http://www.w3.org/TR/xslt` and `http://www.w3.org/TR/xslt20/`. Here are a few books of interest:

- *XML in a nutshell*, Eliotte Rusty Harold & W. Scott Means, O'Reilly (covers XSLT 1.0)
- *XSLT 2.0 Programmer's Reference*, Michael Kay, Wrox

Most of the current implementations (at the date of writing) are limited to XSLT 1.0, to which belongs all the material presented so far in this Chapter. A brief discussion on the limitations of XSLT 1.0 and the impriovements of XSLT 2.0 follows.

3.1 Limitations of XSLT 1.0

XSLT 1.0 has a number of annoying limitations or caveats. It is impossible to process a temporary tree stored into a variable (with <**xsl:variable** `name="t"><toto a="3"/></`**xsl:variable**>). This is sometimes indispensable! Manipulation of strings is not very easy and manipulation of sequences of nodes (for instance, for extracting all nodes with a distinct value) is awkward. It is impossible to define in a portable way new functions to be used in XPath expressions. Using named templates for the same purpose is often verbose, since something equivalent to $y = f(2)$ needs to be written as shown in Figure 20.

Finally, the vast majority of implementations require to store the original document into memory, which makes it impossible to use it with very large (>100MB) documents. Apart from the last one, all these limitations are raised by extensions of XSLT 1.0, either with the extension mechanism that is provided, or in XSLT 2.0. We describe some of these quite briefly.

3.2 Extension Functions

XSLT allows for *extension functions*, defined in specific namespaces. These functions are typically written in a classical programming language, but the mechanism depends on the

```
<xsl:variable name="y">
  <xsl:call-template name="f">
    <xsl:with-param name="x" select="2" />
  </xsl:call-template>
</xsl:variable>
```

Figure 20: XSLT equivalent of $y = f(2)$

precise XSLT engine used. *Extension elements* also exist. Once they are defined, such extension functions can be used in XSLT as shown in Figure 21.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:math="http://exslt.org/math"
  version="1.0"
  extension-element-prefixes="math">

  ...

  <xsl:value-of select="math:cos($angle)" />
```

Figure 21: Example of use of extension functions in XSLT

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:exsl="http://exslt.org/common"
  version="1.0" extension-element-prefixes="exsl">

  ...

  <xsl:variable name="t"><toto a="3" /></xsl:variable>

  <xsl:value-of select="exsl:node-set($t)/*/@a" />
```

Figure 22: Example of use of exsl:node-set

EXSLT (`http://www.exslt.org/`) is a collection of extensions to XSLT which are portable across some XSLT implementations. See the website for the description of the extensions, and which XSLT engines support them (varies greatly). It includes `exsl:node-set` that solves one of the main limitation of XSLT, by allowing to *process temporary trees* stored in a variable (cf Figure 22); `date`, a library for formatting dates and times; `math`, a library of mathematical (in particular, trigonometric) functions; `regexp`, for regular expressions; `strings` library for manipulating strings; etc. Other extension functions outside EXSLT may be provided by each XSLT engine (it is usually especially easy with Java-based engines).

3.3 XSLT 2.0

XSLT 2.0 is a 2007 W3C Recommendation. Like XQuery 1.0, it uses XPath 2.0, a much more powerful language than XPath 1.0 (strong typing, in relation with XML Schemas; regular expressions; loop and conditional expressions; manipulation of sequences of nodes and values; etc.). There are also new functionalities in XSLT 2.0 itself, including native processing of temporary trees, multiple output documents, grouping functionalities, and user-defined functions. All in all, XSLT 2.0 stylesheets tend to be much more concise and readable than XSLT 1.0 stylesheets. XSLT 2.0 is also a much more complex programming language to master and implement (as shown by the disappointingly low number of current implementations).

As a mouth-watering illustration of its capabilities, an XSLT 2.0 stylesheet for producing the list of each word appearing in an XML document with their frequency is given in Figure 23. The same (elementary) task would require a much more complex XSLT 1.0 program.

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

</xsl:stylesheet>
```

Figure 23: Example XSLT 2.0 stylesheet (from *XSLT 2.0 Programmer's Reference*, Michael Kay)

3.4 Implementations

There are a large number of implementations of XSLT 1.0. Most of the libraries for using XSLT from a host programming language also provide a command line interface to a XSLT processor. All modern graphical browsers (Internet Explorer, Firefox, Opera, Safari, Google Chrome) include XSLT engines, used to process `xml-stylesheet` references. Also available via *JavaScript*, with various interfaces.

On the other hand, there are very few implementations of XSLT 2.0. We recommend using SAXON, a Java and .NET implementation of XSLT 2.0 and XQuery 1.0. The full version is commercial (free evaluation version), but a GPL version is available without support of external XML Schemas.

4 Exercises

All the XML sample documents mentioned in the following can be obtained on the Web site. Any Web browser can be used to experiment XSLT transforms. Just put the following processing instruction in the prologue of the XML document that must be transformed:

```
<?xml−stylesheet href="prog.xsl"type="text/xsl"?>
```

where "`prog.xslt`" is of course the name of the XSLT stylesheet.

Exercise 4.1 *Take any XML document, write an XSLT program without any template, and apply the transformation to the document. Explain what is going on.*

Exercise 4.2 *Write a program with a single template that applies to the document root (set the* `match` *attribute to* `'/'`*). The body of the template must instantiate a complete yet simple HTML document showing the main information on a movie: title, year, genre and abstract, as well as director's name. Use only* <**xsl:value−of**> *XSLT elements, and XPath expressions.*

Apply the program to an XML document describig a single movie (e.g., Spider-Man.xml).

Exercise 4.3 *Now write a program that uses the* <**xsl:apply−templates**> *mechanism. The program applies to movies.xml, and creates an HTML representation that shows the list of movies, with the following templates:*

- *the first template applies to the document root and produces a valid HTML document (tags* `<html>`, `<head>`, `<body>`*) ; in* `<body>`*, call for the application of the appropriate templates to the* `<movie>` *elements ;*
- *a second template that applies to a* `<movie>` *element and shows the title, genre and country ; also add an XSLT call for the application of the appropriate template to the* `<actor>` *elements.*
- *finally the last template applies to the actors, and produces a list with the name of each actor and his/her role in the movie.*

The following exercises are meant to complement the material presented in the slides and text. They require some XSLT features which have not been presented but can easily be found on the Web. We provide some hint if needed. These exercises may constitute the basis for larger projects.

Exercise 4.4 *Write an XSLT program which displays the list of name and value of all the attributes of an input document. Hint: the name() XPath functions gives the name of the context node.*

Exercise 4.5 *Write a program that shows the names of the elements of a document, and for each element:*

- *the number of its attributes;*
- *the number of its descendant of type* ***Element*** *;*
- *its number in the document order.*

Hint: the number of nodes in a nodeset is obtained with the XPath function count().

Exercise 4.6 *Write an XSLT program that takes any XML document I as input, and produces as output an XHTML document O which shows the structure and content of I. O should be displayed by a Web browser as it appears in an XML editor, using indentation to represent the hierarchical structure, and showing the opening and ending tags for elements.*

Use nested XHTML lists (`<ol>`) for indentation, and XHTML entities `&lt;` and `&gt;` for displaying element tags.

- *Create a first version which takes only account of **Element** and **Text** nodes.*
- *Then extend the first version by adding attributes (put them in the opening tag, following the XML syntax).*

Exercise 4.7 *Write an XSLT program that produces, from the Movies.xml document, an XHTML document with:*

1. *A table of content, at the top of the page, showing the years, sorted in ascending order, of all the movies.*
2. *Then the list of movies, sorted by year in ascending order. For each movie, show the director, title (in italics) and the summary.*

Of course, each year Y in the table of content is a link that leads to the beginning of the group of movies published in Y. Hint: use XHTML internal anchors (`<li id='a'></a>` can be referred to with `<a href='#a'></a>`.

Exercise 4.8 *This exercise considers MathML documents and their processing with XSLT programs. Recall that in MathML, formulas are represented with prefix notation (operators appear* before *their operands). For instance the prefix representation of the formula $x^2 + 4x + 4$ is:*

```
(+ (^ x 2) (* 4 x) 4)
```

The MathML document is given below.

```
<?xml version='1.0'?>
<apply>
  <plus/>
  <apply>
    <power/>
     <ci>x</ci>
     <cn>2</cn>
   </apply>
   <apply>
     <times/>
     <cn>4</cn>
     <ci>x</ci>
   </apply>
   <cn>4</cn>
</apply>
```

- *Write an XSLT program that takes a MathML document and produces a formula with infix notation. One should obtain, for the previous example:*

  `((x ^ 2) + (4 * x) + 4)`

- *Write an XSLT program that applies to the class of MathML documents without variables and without the* `power` *operator. The program produces the evaluation of the MathML formula. Applied to the following example:*

```
<?xml version='1.0'?>
<apply>
  <plus/>
  <cn>2</cn>
  <cn>4</cn>
</apply>
```

The result of the program should be 6.