XSLT

December 16, 2008

XML is used in a large number of applications, 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, which would be 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 and is the topic of this chapter. 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. We then discuss more advanced features of the language in Section 2. We also evoke some limitations of XSLT 1.0 and see the possibilities to overcome them in Section 3 before providing reference material in Section 4 and exercises in Section **??**.

1 A First Look at XSLT

1.1 Basics

An *XSLT program* (or *XSLT stylesheet*, or just *stylesheet*) has an XML document as input, and is built out of rules that are called *templates*. A given template is associated to an XPath expression that indicates the kind of nodes this templates applies to, and produces a fragment of the output document, in one of the following ways:

- by creating *literal nodes*;
- by *copying values* and *fragments* from the input document;
- by *instantiating* (or calling) other templates.

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

The execution model of XSLT, that we will detail later on, is as follows: initially, the *context node* is the *document node* of the XML document and the best-matching template is instantiated on this node. This template may in turn change the context node and initiate a traversal of the input document.

An original characteristic of XSLT is that XSLT stylesheets are themselves XML documents, 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:`.

```
<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 1: Hello World XSLT Stylesheet

Let us have a look at the sample XSLT stylesheet shown in Figure 1. 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 a < **xsl:output** > one, which specifies to generate an XML document, with the UTF-8 encoding (actually, both are defaults and could be omitted);
- some *template rules*, in this case a template that applies to the root node.

An XSLT stylesheet may be invoked in different ways:

- *programmatically*, through one of the various XSLT libraries;
- through a *command line* interface.
- in a Web Publishing context, by including a styling processing instruction in the XML document (cf Figure 2):
  - the transformation can be processed on the *server side* by a script (e.g., in CGI, PHP, ASP, JSP);
  - or on the *client side* through the XSLT engines integrated to most browsers.

This last case is illustrated in Figure 3.

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

<doc>
  <titi />
</doc>
```

Figure 2: XML file with an XML processing instruction

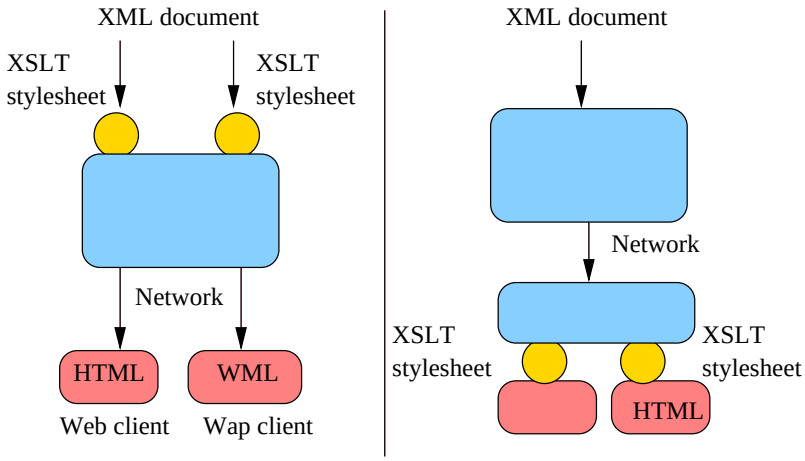

Figure 3: Web publishing with XSLT

```
<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>
```

Figure 4: Example template

1.2 Templates

Consider the (slightly more elaborated) template that is displayed in Figure 4.

A template is defined by two components:

- a *pattern*, that is an XPath expression (with some restrictions, detailed further on) which determines the node to which the template applies. This pattern is the value of the `match` attribute.
- a body, that is a (well-formed!) XML fragment which is inserted in the output when the template is instantiated.

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 are called *patterns*. They must comply to the following requirements:

- 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. Thus, although <**xsl:template** `match='preceding::*[12]'`> is meaningful, it is quite difficult to evaluate and is therefore forbidden.

More formally:

Definition 1.1 A *pattern* is a valid XPath expression which uses only the `child` and `@` axes, and the abbreviation `//`. Predicates are allowed.

Here are a few examples of patterns:

- <**xsl:template** `match='B'`> applies to any `B` element.
- <**xsl:template** `match='A/B'`> applies to any `B` element, child of an `A` element.
- <**xsl:template** `match='@att1'`> applies to any `att1` attribute, whatever its parent element.
- <**xsl:template** `match='A//@att1'`> applies to any `att1` attribute, if its parent element is a descendant of an `A` element.

We can define the semantics of a pattern in the following way:

Definition 1.2 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 $N \in P(T,C)$.

Basically, the content of `xsl:template` may consist of:

Literal elements and text. Example: `<h2>Authors</h2>`. This creates in the output document an element `h2`, with a **Text** child node '`Authors`'.

Values and elements from the input document. Example: <**xsl:value−of** `select='title'/`>. This inserts in the output document a node set, result of the XPath expression `title`.

Call to other templates. Example: <**xsl:apply−templates** `select='authors'/`>. Applies a template to each node in the node set result of the XPath expression `authors`.

Advanced programmatic XSLT elements. Will be described in Section 2.

Templates are then instantiated (from an <**xsl:apply−templates**> invocation, see further on) according to the following principles:

- **Literal elements** (those that don't belong to the XSLT namespace) and **text** are simply copied to the output document.
- **Context node**: A template is always instantiated in the context of a node from the input document.
- **XPath expressions**: all the (relative) XPath expression found in the template are evaluated with respect to the context node (an exception: <**xsl:for−each**>).
- **Calls with** `xsl:apply-templates`: find and instantiate a template for each node selected by the XPath expression `select`.
- **Template call substitution**: any call to other templates is eventually replaced by the instantiation of these templates.

1.3 Applying templates

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

Figure 5: Example of use of <**xsl:apply−templates**>

The <**xsl:apply−templates**> element is used to instantiate templates, as shown in Figure 5. It expects the following attributes:

select an optional XPath expression which, if relative, is interpreted with respect to the context node;

Note: the default value is `child::node()`, i.e., select all the children of the context node.

mode an optional label which can be used to specify which kind of template is required (see Section 2).

A practical example of use of <**xsl:apply−templates**> is given in Figure 6: with this set of rules, the input document of Figure 7 is transformed into the document of Figure 8.

We can now describe the execution model of XSLT. An XSLT stylesheet consists of a set of templates. The transformation of an input document *I* proceeds as follows:

1. The engine considers the *root node R* of *I*, and selects the template that applies to *R*.
2. The template body is copied in the output document *O*.
3. Next, the engine considers all the <**xsl:apply−templates**> that have been copied in *O*, and evaluate the `match` XPath expression, taking *R* as context node.

```
<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 6: Two example templates, one instantiating the other

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

Figure 7: Example document

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

Figure 8: Result of applying the templates from Figure 6 to the document of Figure 7

4. For each node result of the XPath evaluation, a template is selected, and its body replaces the <**xsl:apply−templates**> call.
5. The process iterates, as new <**xsl:apply−templates**> may have been inserted in *O*.
6. The transform terminates when *O* is free of `xsl:` instructions.

2 Advanced Features

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.

```
<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 9: Example of use of <**xsl:output**>

Let us first discuss <**xsl:output**>, an elaborated example of which is given in Figure 9. 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.
- `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`).

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

Figure 10: Example of use of `xsl:import`

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

Figure 11: Example of use of `xsl:include`

An important feature of XSLT is *modularization*, the possibility of reusing templates across stylesheets. Two XSLT elements control this reuse. With <**xsl:import**> (see Figure 10), template rules are imported this way from another stylesheet. Their *precedence* is less that that of the local templates. Note that <**xsl:import**> must be the *first* declaration of the stylesheet. With < **xsl:include** > (see Figure 11), 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.

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 rules apply 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 as follows:
  - A first built-in rule applies to the **Element** nodes and to the root node, cf Figure 12. This can be interpreted as a recursive call to the children of the context node.

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

Figure 12: First built-in rule

  - A second built-in rule applies to **Attribute** and **Text** nodes, cf Figure 13. This rule copies the textual value of the context node to the output document.

The last two declarations discussed here, <**xsl:param**> and < **xsl:variable** > are used to declare global parameters and variables, respectively, as in Figure 14. Global parameters are passed to the stylesheet through some *implementation-defined* way. The `select` attribute gives the default value,

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

Figure 13: Second built-in rule

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

<xsl:variable name="pi" select="3.14159" />
```

Figure 14: Declarations of global parameters and variables

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

```
<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 15: Example of use of 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 15.

Remark 2.1 A call to a named template does not change the context node.

```
<xsl:template name="print">
  <xsl:param name="message" select="string('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="string('Element node')"/>
 </xsl:call-template>
</xsl:template>
```

Figure 16: 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 16 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 17. 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 17: 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 18: Example of use of < **xsl:if** >

Consider the example of use of < **xsl:if** > given in Figure 18. 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.

Remark 2.2

- An XSLT program is an XML document: we must use entities for literals <, >, &.
- XSLT is closely associated to XPath (node selection, node matching, and here data manipulation)

```
<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 19: Example of use of both <**xsl:choose**> and < **xsl:if** >

The other conditional construct, <**xsl:choose**>, is illustrated in Figure 19 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:for−each**>, an example of which is given in 20, 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`);

```
<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 20: Example of use of <**xsl:for−each**>

- Each node of the set becomes in turn the alertcontext 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 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'value='3.14116'/>

  < **xsl:variable** name='children'value='//child'/>

- or as the content of the < **xsl:variable** > element, as shown in Figure 21.

```
<xsl:variable name="Signature">
  Franck Sampori<br/>
  Institution: INRIA<br/>
  Email: <i>franck.sampori@inria.fr</i>
</xsl:variable>
```

Figure 21: Variable definition in the content of an < **xsl:variable** >

Remark 2.3 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 < **xsl:text** >, < **xsl:strip−space** >, < **xsl:preserve−space** >, and `normalize-space` function;

Dynamic creation of elements and attributes < **xsl:element** > and < **xsl:attribute** >.

Multiple document input, and multiple document output `document` function, `<xsl:document>` element (XSLT 2.0, but widely implemented in XSLT 1.0 as an extension function)

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 (cf in particular the associated exercises and projects).

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.

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

Figure 22: Handling whitespace explicitly

Whitespace in the source document can also be handled explicitly, through < **xsl:strip−space** > and < **xsl:preserve−space** >, as shown in Figure 22. < **xsl:strip−space** > specifies the set of nodes whose whitespace-only text child nodes will be removed (or ∗ for all), and < **xsl:preserve−space** > allows for exceptions to this list.

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 23, that transforms document from Figure 24 into the document of Figure 25.

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

Figure 23: Dynamic creation of elements and attributes

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

Figure 24: Example XML document

Remark 2.4 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: `<toto titi="{$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`

Example: `document("toto.xml")/*`

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

Generating unique identifiers

`generate-id($s)` (cf Figure 26) returns a *unique identifier string* for the first node of the nodeset `$s` in document order.

This is especially useful for testing the identity of two different nodes, or to generate HTML anchor names.

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

Figure 25: Result of the transformation of document from Figure 24 by stylesheet of Figure 23

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

Figure 26: Use of `generate-id` to generate link anchors

3 Beyond XSLT 1.0

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.
- 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 27.

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

Figure 27: XSLT equivalent of $y = f(2)$

- 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 precise XSLT engine used. *Extension elements* also exist.

```
<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 28: Example of use of extension functions in XSLT

Once they are defined, such extension functions can be used in XSLT as shown in Figure 28.

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 solves one of the main limitation of XSLT, by allowing to *process temporary trees* stored in a variable (cf Figure 29);

```
<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 29: Example of use of exsl:node-set

date library for formatting dates and times;

math library of mathematical (in particular, trigonometric) functions;

regexp library 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;
- 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 30. 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 30: Example XSLT 2.0 stylesheet (from *XSLT 2.0 Programmer's Reference*, Michael Kay)

4 Further Readings

4.1 Implementations

There are a large number of implementations of XSLT 1.0, some of which are listed below. Most of the libraries for using XSLT from a host programming language also provide a command line interface to a XSLT processor.

Browsers All modern graphical browsers (Internet Explorer, Firefox, Opera, Safari) include XSLT engines, used to process `xml-stylesheet` references. Also available via *JavaScript*, with various interfaces.

libxslt Free *C* library for XSLT transformations. Includes easy-to-use `xsltproc` command-line tool. *Perl* and *Python* wrappers exist.

Sablotron Free *C++* XSLT engine.

Xalan-C++ Free *C++* XSLT engine.

JAXP *Java* API for Transformation. Common interface for various JAVA XSLT engines (e.g., SAXON, Xalan, Oracle). Starting from JDK 1.4, a version of Xalan is bundled with Java.

System.Xml *.NET* XML and XSLT library.

php-xslt XSLT extension for *PHP*, based on Sablotron.

4XSLT Free XSLT *Python* library.

On the other hand, there are very few implementations of XSLT 2.0:

SAXON *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.

Oracle XML Developer's Kit *Java* implementation of various XML technologies, including XSLT 2.0, XQuery 1.0, with full support of XML Schema. Less mature than SAXON.

4.2 References

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`
- `http://www.w3.org/TR/xslt20/`

Here are a few books of interest:

XSLT 1.0

- *XML in a nutshell*, Eliotte Rusty Harold & W. Scott Means, O'Reilly
- *Comprendre XSLT*, Bernd Amman & Philippe Rigaux, O'Reilly

XSLT 2.0 *XSLT 2.0 Programmer's Reference*, Michael Kay, Wrox