

set to a given value, the case statements on lines 2 and 7 will activate one of the lines in the range 3–5 and another line in the range 8–10, depending on the result of the evaluation of the procedure “(inherited-attribute-string "xml:lang")”. For instance, with a value of `xml:lang` equal to “de,” for German, `FIGNAME` would get the value “Abbildung” (line 3), and `TABNAME` would get the value “Tabelle” (line 8). With these constant definitions we can construct the `caption` element using the appropriate character strings for the language considered. On line 17 the `if` statement verifies whether the ancestor of the `caption` element is `figure`, in which case the literal string `FIGNAME` will be placed into the `paragraph` flow object at that point, otherwise `TABNAME`. Similarly, lines 18–24 take care of putting the right figure or table number following the text string chosen earlier. In fact, to get the number, DSSSL counts the number of `figure` or `table` elements (the “if” test and its branches on lines 19–21). This number is then formatted using the `format-number` procedure (line 18) and represented as a “decimal” number (hence the “1” on line 23 specifies the format to be used for the number),⁷ followed by a dot (line 24).

This example shows clearly the kind of manipulations that are possible with DSSSL using the many procedures and flow objects defined in the DSSSL standard. Another useful feature is the use of “modes,” which allows certain elements to be processed (and output) more than once in different modes. For instance, heading titles or figure and table captions, can be output once when one is composing the main text, and a second time when constructing the table of contents and lists of tables and figures.

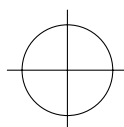
If you want to find out how DSSSL is used in real-life applications dealing with large documents, you should take a look at Norman Walsh’s modular DocBook style sheets [`↔`DBDSSSL] for formatting SGML documents marked up using the DocBook DTD. Walsh provides two style sheets: a generic one for printing DocBook documents using RTF, \TeX , or MIF, and another one for transforming them into HTML. Several hooks are provided to allow the user to customize the output. Apart from their practical usefulness, these complex style sheets are also a good place to learn about DSSSL.

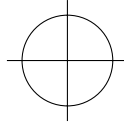
7.6 Extensible Stylesheet Language

XSL [`↔`XSLSPEC] is a language for expressing style sheets that describe rules for the presentation of XML elements. It proposes a syntax for handling two subprocesses:

1. a *transformation* of the source tree of the XML document into a result tree; and
2. an interpretation of the result tree to produce *formatting objects* for output on various media, such as a computer screen, paper, or audio.

⁷Section 8 of the DSSSL Standard describes the expression language. It is here that you will find an explanation of the various procedures we have used in this and other examples. In particular, Section 8.5.7.24 details the `format-number` procedure.





The transformation part XSLT can be used independently of the formatting objects. It builds upon the XML Path Language (XPath) for addressing the components of an XML document. In this section we shall look in some detail at the XPath and XSLT languages, but we shall only look at an example for the formatting objects, since work is still ongoing in that area.

7.6.1 XPath for addressing parts of an XML document

XPath [\leftrightarrow XPATH] provides syntax and semantics for functionality common to the XSLT [\leftrightarrow XSLT] and XPointer [\leftrightarrow XPTR] languages. The primary purpose of XPath is to address parts of an XML document. XPath uses a compact *non-XML* syntax to facilitate its use within URIs and XML attribute values.

XPath models an XML document as a tree of nodes. There are different types of nodes and XPath will compute a string-value for each type of node. Some types of nodes also have names.

The primary syntactic construct in XPath is the *expression*, which is evaluated to yield an object having one of the following four basic types:

- a *node-set* (an unordered collection of nodes without duplicates);
- a *boolean* (true or false);
- a *number* (a floating-point number); or
- a *string* (a sequence of Unicode characters).

Expression evaluation occurs with respect to a *context* which consists of:

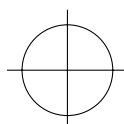
- a node (the context node);
- a pair of non-zero positive integers (for determining the position of the node with respect to the context size);
- a set of variable bindings providing mappings from variable names to variable values, where values are objects;
- a function library consisting of a mapping from function names to functions, each function taking zero or more arguments and returning a single result;
- the set of namespace declarations in scope for the expression.

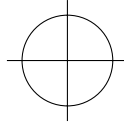
XPath defines a “Location Path” which typically consists of a /-separated list of location steps of the form

```
axis-name :: node-test[predicate]*
```

For example, the expression `child::emph` locates all children of the context node that are of element type `emph`, while `child::par/child::emph` locates the `emph` children of the `par` children of the context node.

An `axis-name` specifies a sequence of candidate locations, given certain bindings called a “context” and in particular a current location called the “context node”.





The context node is initially the document root, and more generally the results, in turn, of a prior location step.

Predicates are evaluated for each candidate location along the specified axis, and typically test the element type, attributes, positions, and/or other properties of nodes. Multiple predicates may be specified, and serve as successive filters.

For example, the following expression selects the second following sibling of type `emph` of the child element of type `par` of the current node.

```
child::par/following-sibling::emph[position()=2]
```

Each location step locates an ordered list of data portions, which is called a “context node list” or “context”, and any following location steps are interpreted relative to the nodes in that context node list. Location steps can be used in a sequence, interpreted from left to right.

A location step can be *absolute* when it has an initial `/`, which by itself selects the root element of the document, or it can be *relative*. In the relative case it consists of one or more location steps separated by `/` composed together from left to right. Each step in turn selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together.

7.6.1.1 Axes

XPath axes depend on the existence of a context node list, and locate other nodes relative to each of its nodes in turn. Candidate nodes along a given axis are ordered by distance from the “context node”. In the absence of a context node list, the context is the root element of the containing resource.

Each of the relative axes locates a list of nodes that are candidates for membership in the resulting context node list. Actual results are selected from the candidate by predicate arguments (see Section 7.6.1.3). If no predicate is specified, all candidates from the axis are selected. All relative axes accept the same form of predicates as arguments. The XPath relative axes are:

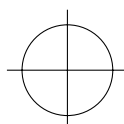
`child` Locates all types of child nodes of the context node (attributes and namespace are not considered children).

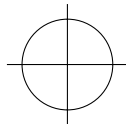
`descendant` Locates all types of descendant nodes appearing anywhere within the content of the context node.

`descendant-or-self` Identical to the `descendant` axis except that the context node itself is included as a candidate, preceding all descendants.

`parent` Locates the parent of the content node, if it exists.

`ancestor` Locates the ancestors of the context node, the first node in the list being the immediate parent, the last node in the list being the root of the document.





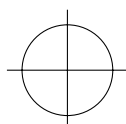
- ancestor-or-self** As the *ancestor* axis but also including the context node itself as a candidate, preceding all ancestors.
- preceding-sibling** Locates sibling nodes (sharing their parent with the context node) that appear before (preceding) the context node. The nodes will be in *reverse document order*, the first node in the list being the immediately preceding sibling, and the last node in the list being the first child of the parent.
- following-sibling** Locates sibling nodes (sharing their parent with the context node) that appear after (following) the context node. The nodes in the list appear in *document order*.
- preceding** Locates nodes that begin before (preceding) the entire context node. The list is in *reverse document order*: the node closest to the context node first, the root node last; ancestors are excluded, as well as attribute and namespace nodes.
- following** Locates nodes that begin after (following) the entire context node. The list is in *document order*: the first node in the list is for the first node whose start-tag occurs after the context node's end-tag; ancestors are excluded, as well as attribute and namespace nodes.
- self** Locates (for each context node in the context node list), a singleton nodelist containing that same context node. This is useful for applying multiple predicates to a single axis, particularly when predicates other than the first one must test a context node's position among all those context nodes that were selected by the prior predicates.
- attribute** The attributes of the context node. If the context node is not of type element, the list is empty. The order of nodes on this axis is undefined. Typically, a single attribute will be selected by name.
- namespace** The namespace nodes of the context node. If the context node is not of type element, the list is empty.

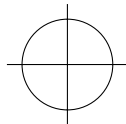
7.6.1.2 Node tests

Every axis has a *principal node type*, which is *attribute* for the *attribute* axis, *namespace* for the *namespace* axis and *element* for all other axes.

A node test that is a qualified name (a combination of namespace and a non-colon name) is "true" if and only if the node is of the principal node type and has a name equal to that specified by the qualified name. For example, `child::par` selects the `par` element children of the context node; if the context node has no `par` children, it will select an empty set of nodes. Similarly, `attribute::href` selects the `href` attribute of the context node; if the context node has no `href` attribute, it will select an empty set of nodes.

The node test `text()` is true for any text node, for example, to select the text node children of the context node use `child::text()`. Similarly, the node test





`processing-instruction()` is true for any processing instruction, and the node test `comment()` selects comment node. Finally, a node test `node()` is true for any node of any type whatsoever.

7.6.1.3 Predicates

What is located by a given location step depends on the axis and the predicate(s)—the node test may be considered a special case of a predicate. The axis defines an ordered list of potential candidates, and the predicates (if any) select actual results from among the candidates, as follows:

1. The particular axis in use defines an ordered list of candidate nodes. When the context node list has more than one node, the axes are applied using each node in the context node list as the context node, and the results are unioned together.
2. The node test and predicates (if any) are then evaluated, with each candidate in turn serving as the context of evaluation, or “context node”. This step eliminates all candidates for which the node-test and predicates do not evaluate to true.
3. The same process is repeated for each additional predicate, in order.

If no predicate or node test is specified, all candidates become members of the resulting node list.

For example `child::par[position()=2] / following-sibling::text()` selects all text nodes siblings of the second `par` child of the context node.

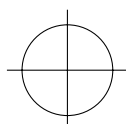
Another interesting case is `footnote[ancestor::footnote]` which finds `footnote` nodes a `footnote` ancestor, that is, we are dealing with nested `footnote` elements. As this is forbidden, for instance in \LaTeX , you can use such a pattern to warn about the presence of constructs that are otherwise allowed by the DTD (see Section B.4.5.1 for a discussion of this point).

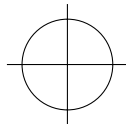
7.6.1.4 Abbreviated syntax

An *abbreviated* syntax for often-occurring combinations exists. For instance, `child::par[position()=2]` can be rewritten as `par[2]`. Attributes are identified with the “@” character, so that the `when` attribute of an `invitation` element just below the root element can be located with `/invitation/@when`.

More general ancestor-descendant relationships are expressed with the `//` operator that allows zero or more generations between the element at the left and right side of the operator. Hence `body//emph` selects all `emph` descendants of the `body` children of the current node.

The “wildcard” character “*” can be used to represent any single element type, so that the pattern `*` selects all children of the current node, and `*/emph` selects all `emph` grandchildren of the current node. On the other hand, the pattern `par/*` matches any element with a `par` element as parent.





The pattern “.” selects the current node. This can be used, for instance, to represent the current node explicitly in ancestor-descendant relations, as in `par//.`, a pattern that selects all `par` ancestors of the current node, or `../emph` that selects all its `emph` descendants. In a similar way, the pattern “..” selects the parent of the current node, so that `../par` selects `par` sibling elements of the current node.

7.6.1.5 Expressions

Several kinds of tests can be used within predicates. Below we list functions and operators that can be used. Zero or more arguments of the following types can be present: a boolean (*b*), a node-set (*ns*), a number (*nb*), an object (*o*), or a string (*s*).

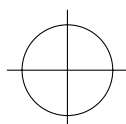
- *Node set functions*: `last()`, `position()`, `id(o)`, `local-name(ns?)`, `namespace-uri(ns?)`, `name(ns?)`.
- Functions to *cast* sub-expressions to particular types: `boolean(o)`, `string(o)`, `text()`, `processing-instruction(target)`, and `number(o)`.
- *Comparisons*: the infix operators `=`, `!=`, `<`, `<=`, `>`, and `>=`.
- *Booleans*: the operators `and`, `or`, and the functions `not(b)`, `true()`, `false()` and `lang(s)`, which returns true if the language of the context node as specified by the `xml:lang` attribute is the same as the language specified by the argument string.
- *String functions*: `string(o?)`, `concat(s, s, s*)`, `starts-with(s, s)`, `contains(s, s)`, `substring(s, nb, nb?)`, `substring-after(s, s)`, `substring-before(s, s)`, `string-length(s?)`, `normalize-space(s?)`, `translate(s, s, s)`.
- *Numeric operators*: `+`, unary and binary `-`, `*`, `/`, `//`, `|`, infix `div`, `mod`.
- *Number functions*: `sum(ns)`, `floor(n)`, `ceiling(n)`, and `round(n)`.

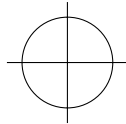
7.6.1.6 The data model

XPath operates on an XML document as a tree, which can contain seven types of node: root, element, text, attribute, namespace, processing instruction nodes, and comment nodes.

For every type of node, there is a way of determining a string-value for a node of that type, which for some types is part of the node while for other is computed from the string-value of descendant nodes.

There is an ordering, *document order*, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the root node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the





occurrence of their start-tag in the XML source (after expansion of entities). The attribute nodes and namespace nodes of an element occur before the children of the element. Namespace nodes occur before the attribute nodes. The relative order of namespace and attribute nodes themselves is undefined and thus implementation-dependent. *Reverse document order* is the reverse of document order.

Root nodes and element nodes have an ordered list of child nodes. Every node other than the root node has exactly one parent, which is either an element node or the root node. A root node or an element node is the parent of each of its child nodes. The descendants of a node are the children of the node and the descendants of the children of the node.

7.6.2 The XSL Transformation Language

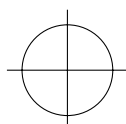
XSLT [\leftrightarrow XSLT] defines the syntax and semantics for the transformation part of the XSL Specification which describes how an XML document is transformed into another XML document. XSLT is not directly linked to XSL's formatting vocabulary, but it can also be used independently of XSL, as we shall show in the example sections later.

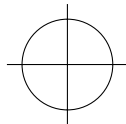
An XML transformation expressed in the XSLT language is a well-formed XML document that conforms to the namespaces recommendation [\leftrightarrow XMLNS]. It can include elements that are defined by XSLT (they then belong to the XSLT namespace) or are defined elsewhere.

An XSLT transformation is called a style sheet and consists of a set of rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

A template is instantiated for a particular source element to create part of the result tree. A template can contain elements that specify literal result element structure. A template can also contain elements from the XSLT namespace that are instructions for creating result tree fragments. When a template is instantiated, each instruction is executed and replaced by the result tree fragment that it creates. Instructions can select and process descendant source elements. Processing a descendant element creates a result tree fragment by finding the applicable template rule and instantiating its template. Note that elements are only processed when they have been selected by the execution of an instruction. The result tree is constructed by finding the template rule for the root node and instantiating its template.

In the process of finding the applicable template rule, more than one template rule may have a pattern that matches a given element. However, only one template rule will be applied by using conflict resolution procedures (see Section 7.6.2.4).





A single template can create structures of arbitrary complexity; it can pull string values out of arbitrary locations in the source tree; it can generate structures that are repeated according to the occurrence of elements in the source tree. For simple transformations where the structure of the result tree is independent of the structure of the source tree, a style sheet can often consist of only a single template, which functions as a template for the complete result tree. XSLT allows a simplified syntax for such style sheets.

When a template is instantiated, it is always instantiated with respect to a current node and a current node list. The current node is always a member of the current node list. Many operations in XSLT are relative to the current node.

XSLT makes use of the XPath expression language for selecting elements for processing, for conditional processing and for generating text.

7.6.2.1 The general structure of an XSL style sheet

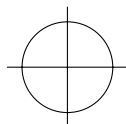
The URI `http://www.w3.org/1999/XSL/Transform` is the namespace identifying an XSLT style sheet to XSLT processors, that must use the XML namespaces mechanism to recognize elements and attributes from this namespace. Extensions (elements and functions) must be in a separate namespace that has to be defined separately.

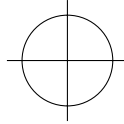
A style sheet is represented by an `xsl:stylesheet` element in an XML document. It must have a `version` attribute, indicating the version of XSLT that the style sheet requires. Presently only version 1.0 exists.

A general XSL style sheet can contain zero or more instances of each of the thirteen top-level elements (eight are empty, lines 3–10, and five can have content, lines 11–15) following. Ellipses (...) indicate possible additional content.

```
1 <xsl:stylesheet version="1.0"
2     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3   <xsl:import href="..." />
4   <xsl:include href="..." />
5   <xsl:decimal-format name="..." />
6   <xsl:key name="..." match="..." use="..." />
7   <xsl:namespace-alias stylesheet-prefix="..." result-prefix="..." />
8   <xsl:output method="..." />
9   <xsl:preserve-space elements="..." />
10  <xsl:strip-space elements="..." />
11  <xsl:attribute-set name="..."> ... </xsl:attribute-set>
12  <xsl:param name="..."> ... </xsl:param>
13  <xsl:template match="..."> ... </xsl:template>
14  <xsl:template name="..."> ... </xsl:template>
15  <xsl:variable name="..."> ... </xsl:variable>
16 </xsl:stylesheet>
```

Lines 1–2 declare the `xsl` namespace and the version of XSLT. The order in which the child elements of the `xsl:stylesheet` element occur is not significant,





except that `xsl:import` elements *must* always be specified first, at the beginning of the style sheet. In the following sections we will explain the use of some of these elements in more detail as we need them.

The following is an example of a simple XSL style sheet that constructs a result tree for a sequence of `par` elements containing `emph` elements using XSL's formatting object vocabulary:

```

1  <?xml version='1.0'?>
2  <xsl:stylesheet version="1.0"
3      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4      xmlns:fo="http://www.w3.org/1999/XSL/Format">
5      <xsl:template match="/">
6          <fo:basic-page-sequence font-family="Helvetica" font-size="10pt" >
7              <xsl:apply-templates/>
8          </fo:basic-page-sequence>
9      </xsl:template>
10     <xsl:template match="par">
11         <fo:block indent-start="10pt" space-before="12pt">
12             <xsl:apply-templates/>
13         </fo:block>
14     </xsl:template>
15     <xsl:template match="emph">
16         <fo:inline-sequence font-style="italic">
17             <xsl:apply-templates/>
18         </fo:inline-sequence>
19     </xsl:template>
20 </xsl:stylesheet>

```

The string following `xmlns:` on lines 3 and 4 declares a shorthand for a namespace to allow the parser to interpret the elements in the document instance (see Section B.3 for details). In this case on these lines we declare the `xsl` transformation and `fo` formatting object namespaces. However, we could also use CSS formatting objects (see W3C note [↔XSLCSS]), in which case we could replace line 4 with the following namespace declaration:

```
xmlns:css="http://www.w3.org/TR/NOTE-XSL-and-CSS"
```

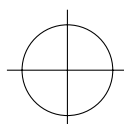
Next follow three template rules. The first rule (lines 5–9) is for the root node, and it specifies that the document as a whole should be formatted as a “page sequence” formatting object, set in a 10 pt Helvetica typeface. The second rule (lines 10–14) declares that each `par` element should result in a “block” formatting object, which is separated from the previous block by twelve points and whose first text line is indented by ten points. Finally the third rule (lines 15–19) declares that an `emph` element corresponds to a sequence formatting object, and that its contents are typeset in an italic typeface.

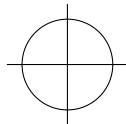
A simplified syntax is allowed for style sheets that consist of only a single template for the root node. For example

```

1  <html xsl:version="1.0"
2      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns="http://www.w3.org/TR/xhtml1/strict">
4      <head>
5          <title>XSL simplified syntax</title>
6      </head>

```





```

7   <body>
8     <p>A famous quote <xsl:value-of select="FamousQuote"/></p>
9   </body>
10  </html>

```

is equivalent to

```

1  <xsl:stylesheet version="1.0"
2     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3     xmlns="http://www.w3.org/TR/xhtml1/strict">
4  <xsl:template match="/">
5  <html>
6  <head>
7     <title>XSL simplified syntax</title>
8  </head>
9  <body>
10     <p>A famous quote <xsl:value-of select="FamousQuote"/></p>
11  </body>
12 </html>
13 </xsl:template>
14 </xsl:stylesheet>

```

Thus, users with simple applications need not bother with templates at all, but just write their HTML code and retrieve the information from the XML source file.

7.6.2.2 The XSLT model

The XSLT and XPath data models are similar. XSLT treats source, result and style sheet documents in the same way and two XML documents characterized by the same tree are handled identically. Comments and processing instructions in a style sheet are ignored.

The normal restrictions on the children of the root node are relaxed for the result tree. The result tree may have any sequence of nodes as children that would be possible for an element node. In particular, when writing using the XML output method, it is possible that a result tree will not be a well-formed XML document.

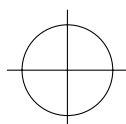
After the tree for a source document or style sheet document has been constructed, and before further processing, text nodes that contain only whitespace characters can be stripped. The handling of white space is controlled by the instructions `xsl:strip-space` and `xsl:preserve-space` elements. They both take an attribute `elements` where one specifies a blank-separated list of elements for which white space has to be stripped or preserved.

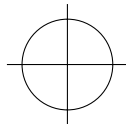
XSLT also adopts the expression language of XPath to select nodes for processing, specify conditions for different ways of processing a node, and generate text to be inserted in the result tree.

XSLT expressions occur as the value of certain attributes and within curly braces in attribute value templates.

7.6.2.3 Template rules

A list of source nodes is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. A list of source





nodes is processed by appending the result tree structure created by processing each of the members of the list in order. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. The process of matching, instantiation and selection is continued recursively until no new source nodes are selected for processing.

A template rule is specified with the `xsl:template` element. The `match` attribute is a pattern that identifies the source node or nodes to which the rule applies. The `match` attribute is required unless one is dealing with a *named* template. The `xsl:apply-templates` element recursively processes the children of the source element.

In the absence of a `select` attribute, the `xsl:apply-templates` instruction processes all of the children of the current node, including text nodes. The `select` attribute can be used to process nodes selected by an expression instead of processing all children. The expression must evaluate to a node-set. The selected set of nodes is processed in document order, unless a sorting specification is present.

Let us reproduce here lines 15–19 of the first example in Section 7.6.2.1:

```

15 <xsl:template match="emph"> <!-- match pattern -->
16 <fo:inline-sequence font-style="italic"> <!-- action template + -->
17 <xsl:apply-templates/> <!-- | -->
18 </fo:inline-sequence> <!-- action template + -->
19 </xsl:template>

```

This template will match all elements of type *emph* (line 15). For each occurrence of an *emph* element in the source tree, a `fo:inline-sequence` formatting object (lines 16–18) should be added to the result tree. The `xsl:apply-templates` element (line 17) will recursively process the children of the source element in question.

The `select` attribute allows one to choose which template has to be applied, allowing one to process elements that are not descendants of the current node. The example that follows assumes that a *faculty* element has *department* children and *professor* descendants. It first finds a *professor's* *faculty* and then processes the *department* children of the *faculty*.

```

1 <xsl:template match="professor">
2 <fo:block>
3 <xsl:apply-templates select="name"/> belongs to the
4 <xsl:apply-templates select="ancestor::faculty/department"/> Department.
5 </fo:block>
6 </xsl:template>

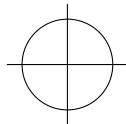
```

Multiple `xsl:apply-templates` elements can occur in a single template for instance to do simple reordering.

```

1 <xsl:template match="country">
2 <table><xsl:apply-templates select="capital"/></table>
3 <table><xsl:apply-templates select="currency"/></table>
4 </xsl:template>

```



In this case two HTML tables are created, one with the name of the capital and the other with the name of the currency of a country.

So we see that once a rule fires for a given element, the rule's template is instantiated. A template can add literal result elements, character data (text), and instructions for creating fragments of the result tree (copying, sorting, numbering, or executing macros).

In practice, as we shall see later in our examples, templates can output a set of formatting objects, or write HTML, \LaTeX code or Unicode text.

7.6.2.4 Conflict resolution for template rules

When a template matches more than one source node the following set of rules is used to resolve the conflict in a way that there is only one template which applies.

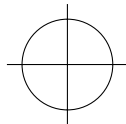
- Imported templates have a lower precedence.
- Lower priority templates have a lower precedence; priorities for templates are assigned using the `priority` attribute whose value is a real number (positive or negative).
- Default priorities are assigned as follows:
 - patterns with multiple `|` separators is like the set of all template rules taken separately;
 - the most common case of a child or attribute specifier followed by a qualified name has priority 0;
 - a child or attribute specifier followed by a wildcard selection has a priority of -0.25 ;
 - a single node test has a priority of -0.5 ;
 - otherwise, the priority is 0.5 .

7.6.2.5 Modes and default templates

An element can be processed multiple times, each time producing a different result by using an optional mode attribute with the `xsl:template` and `xsl:apply-templates` elements. If an `xsl:apply-templates` element has a mode attribute, then it applies only to those template rules from `xsl:template` elements that have a mode attribute with the same value. Without a mode attribute `xsl:apply-templates` applies only to template rules from `xsl:template` elements that also do not have a mode attribute.

When no successful pattern matches exist for a rule in the style sheet, a *default* built-in template rule is implicitly applied for the root and all element nodes. It corresponds to the following definition:

```
1 <xsl:template match="*/">
2   <xsl:apply-templates/>
3 </xsl:template>
```



This default rule stipulates that processing should continue with the child elements (`xs1:apply-templates` element on line 2). Although this template (line 1) matches the root and any other element, it has a smaller priority than any other template rule in a style sheet. This mechanism provides you with a convenient way to specify your own default rule by substituting your XSL commands for those on line 2. In this way your rule will override the default built-in behavior.

A similar built-in template rule exists for each mode, which allows recursive processing to continue in the same mode in the absence of a successful pattern match by an explicit template rule in the style sheet.

The built-in template rules for text and attribute nodes copy text through (lines 1–3), while the built-in template rule for processing instructions and comments is to do nothing (line 5).

```

1 <xs1:template match="text()|@*">
2   <xs1:value-of select="."/>
3 </xs1:template>
4
5 <xs1:template match="processing-instruction()|comment()"/>

```

The built-in template rules are treated as if they were imported implicitly before the style sheet and so have lower import precedence than all other template rules. Thus, the author can override a built-in template rule by including an explicit template rule.

As an example, consider the following minimal “empty” (and trivial) style sheet, which declares only the `xs1` namespace.

```

1 <xs1:stylesheet version="1.0" xmlns:xs1="http://www.w3.org/1999/XSL/Transform">
2 </xs1:stylesheet>

```

This style sheet, which we call `empty.xs1`, will (implicitly) apply the built-in default template rule and process all elements of the document. This is similar to the DSSSL file `empty.dsl` that we introduced in Section 7.5.3.1.

7.6.3 Formatting objects and their properties

Formatting objects are applied to the result tree node by being contained in the pattern part of the element. The general syntax is the following:

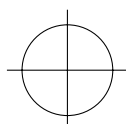
```

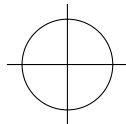
<xs1:template match="pattern">
  <fo:formatting-object (style property="value")*>
    [processing instructions]*
  </fo:formatting-object>
</xs1:template>

```

We have already seen an example of this syntax at the beginning of Section 7.6.2.1.

The XSL specification defines, as far as possible, a formatting model that is compatible with both CSS and DSSSL in the names for objects, definitions, and property names.





As the formatting objects vocabulary is not yet finalized we shall only show an example in Section 7.6.6. For more details you should consult the final version of the XSL Specification [↔XSLSPEC] when it becomes available.

7.6.4 XSL processors and tools

Several processors can handle XSL style files. To run our examples we have chosen James Clark's `xt` processor [↔XTPROC], a Java implementation of the XPath and XSLT parts of XSL. Other XSL parsers exist, such as LotusXSL [↔LOTUSXSL] and Saxon [↔SAXON]. For more on XSL and other processors, see [↔XSLOASIS].

7.6.4.1 Using the `xt` processor

By default the `xt` program uses Clark's `xp` XML parser [↔XPPARS], which is also written in Java. It is thus sufficient, if you have Java installed on your machine, to download the `xp` and `xt` zip archives from Clark's Web site. They contain the Java archives `sax.jar`, `xp.jar`, and `xt.jar`. Add them to your Java class path, and off you go. On Windows the commands could be something like (depending on which version of Java you have available) the following:

```
1 set classpath=d:\xml\xt.jar;d:\xml\xp.jar;d:\xml\sax.jar;d:\jdk1.1.6\src;
2 %JAVA_HOME%\bin\java com.jclark.xml.sax.Driver %1 %2 %3
```

where the `JAVA_HOME` environment variable (line 2) is the directory where your Java installation lives. Similarly on UNIX (Bourne shell), you could write the following:

```
1 DIR=/afs/cern.ch/asis/src/archive/java
2 CLASSPATH=$DIR/xt.jar:$DIR/xp.jar:$DIR/sax.jar:$CLASSPATH
3 export CLASSPATH
4 java com.jclark.xml.sax.Driver $1 $2 $3
```

The variable `DIR` (line 1) is the directory where you keep your Java archives, while line 4 assumes that the `java` program is in your search path for executables.

If you save these lines in a command script `xt.bat` (Windows) or `xt` (UNIX), you can execute `xt` by typing:

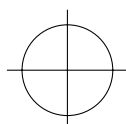
```
xt XML-source-file XSL-style-sheet Output-file
```

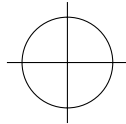
If the third argument is not specified, the output is written to the "standard output" (e.g., the computer screen). We can use the style sheet `empty.xml` and the XML example `invitation2.xml` of Section 7.4.5 with `xt` by entering the command:

```
xt invitation2.xml empty.xml
```

We get the following output:

```
1 I would like to invite you all to celebrate
2 the birth of Invitation, my
```





```

3 first XML document child.
4
5 Please do your best to come and join me next Friday
6 evening. And, do not forget to bring your friends.
7
8 I really look forward to see you soon!
```

As expected, we see the contents of the text nodes for all elements of the document. The attribute values (of the `invitation` element) are not copied to the output. The copying-through feature of text nodes can assist you in developing an XSL style sheet step-by-step. Indeed, for a complex document this allows you to construct and fine tune the rules gradually, since the content of all non-specified elements are copied to the output. It thus provides a convenient context for development and debugging.

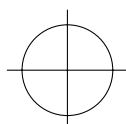
Two XSL style sheets [`↔SHOWTREE`] and [`↔FANCYTREE`] construct a convenient tree view of an XML document and can serve as the basis for exploring an advanced use of the XSL language.

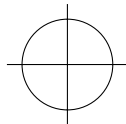
7.6.5 Using XSL to generate HTML or L^AT_EX

To show some of the possibilities of XSL in the area of formatting an XML document, once more we are going to use the two versions of the `invitation` XML files. First we will generate L^AT_EX, in much the same way as we did with Perl, and use the same class file. We have to define templates for all the XML elements that we want to treat, as shown in the following XSL style file `invlat1.xsl`:

```

1 <?xml version='1.0'?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4 <xsl:output method="text" indent="no" encoding="ISO-8859-1"/>
5
6 <xsl:strip-space elements="*" />
7
8 <xsl:template match="invitation">
9 <xsl:text>\documentclass[12pt]{article}
10 \usepackage{invitation}
11 \begin{document}
12 </xsl:text>
13 <xsl:apply-templates/>
14 <xsl:text>\end{document}
15 </xsl:text>
16 </xsl:template>
17
18 <xsl:template match="front">
19 <xsl:text>\begin{front}
20 \To{</xsl:text>
21 <xsl:value-of select="to"/>
22 <xsl:text>}
23 \Date{</xsl:text>
24 <xsl:value-of select="date"/>
25 <xsl:text>}
26 \Where{</xsl:text>
27 <xsl:value-of select="where"/>
28 <xsl:text>}
29 \Why{</xsl:text>
```





```

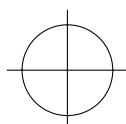
30 <xsl:value-of select="why"/>
31 <xsl:text>}
32 \end{Front}
33 </xsl:text>
34 </xsl:template>
35
36 <xsl:template match="body">
37 <xsl:text>\begin{Body}
38 </xsl:text>
39 <xsl:apply-templates/>
40 <xsl:text>
41 \end{Body}
42 </xsl:text>
43 </xsl:template>
44
45 <xsl:template match="par">
46 <xsl:text>
47 \par </xsl:text>
48 <xsl:apply-templates/>
49 </xsl:template>
50
51 <xsl:template match="emph">
52 <xsl:text>\emph{</xsl:text>
53 <xsl:apply-templates/>
54 <xsl:text>}</xsl:text>
55 </xsl:template>
56
57 <xsl:template match="back">
58 <xsl:text>\begin{Back}
59 \Signature{</xsl:text>
60 <xsl:value-of select="signature"/>
61 <xsl:text>}
62 \end{Back}
63 </xsl:text>
64 </xsl:template>
65
66 </xsl:stylesheet>

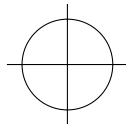
```

After declaring the usual XSLT namespace (line 2), we state that we want to output a text file using the ISO-8859-1 encoding (line 4). We also indicate (line 6) that we want all “default” space to be stripped. This means that only whitespace, which we specify ourselves, will be retained. This is useful because XML and XSL by default leave whitespace untouched, which can lead to strange effects with \LaTeX .

The first template (lines 8–16) treats the `invitation` element. It initializes \LaTeX (lines 9–10), starts the document (line 11), and then lets XSL process all its child elements (line 13) before finalizing the document (line 14). The \LaTeX commands are entered literally in the output stream by the `xsl:text` elements. Inside this element, space and newline characters are significant and are faithfully written to the output.

The second template (lines 18–34) deals with the `front` element and its children. First we open the `Front` environment (line 19), write the `\To` command, and then tell XSL to go and fetch the content of the `to` element (line 21) with the help of the `xsl:value` command. Similarly we get the information for the `\Date` (lines 23–25), `\Where` (lines 26–28), and `\Why` (lines 29–31) commands. Lines 31–33 take care of closing the `Front` environment.





We now arrive at the body element of the XML file, which is handled with lines 36–43. The action consists of starting the Body environment (lines 37–38), telling XSL to handle the children of the XML body element (line 39), and closing the Body environment (lines 40–42).

The par element is handled with the pattern on lines 45–49; a \LaTeX `\par` command is issued for each such element (line 47) before continuing to treat children elements of par itself (line 48).

The content of emph elements is transferred into the argument of a \LaTeX `\emph` command (lines 52–54).

Finally we arrive at the back element, which is handled on lines 57–64. Line 58 starts a \LaTeX Back environment, while lines 59–61 take care of copying the content of the signature element into the argument of \LaTeX 's `\Signature` command. Line 62 ends the Back environment.

We use the XML file `invitation.xml` defined on page 254 and the earlier XSL style sheet with the `xt` XSL processor, as follows:

```
xt invitation.xml invlat1.xsl invlat1.tex
```

The \LaTeX file `invlat1.tex` that we obtain follows. It is completely identical to the \TeX file we showed on page 295.

```

1 \documentclass[]{article}
2 \usepackage{invitation}
3 \begin{document}
4 \begin{Front}
5 \To{Anna, Bernard, Didier, Johanna}
6 \Date{Next Friday Evening at 8 pm}
7 \Where{The Web Cafe}
8 \Why{My first XML baby}
9 \end{Front}
10 \begin{Body}
11 \par
12 I would like to invite you all to celebrate
13 the birth of \emph{Invitation}, my
14 first XML document child.
15 \par
16 Please do your best to come and join me next Friday
17 evening. And, do not forget to bring your friends.
18 \par
19 I \emph{really} look forward to see you soon!
20 \end{Body}
21 \begin{Back}
22 \Signature{Michel}
23 \end{Back}
24 \end{document}

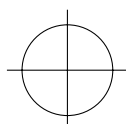
```

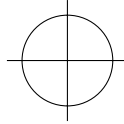
Next using the the XML file `invitation2.xml` described in Section 7.4.5 we will show how to transform it into HTML. The style sheet `invhtml2.xsl` follows:

```

1 <?xml version='1.0'?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4 <xsl:output method="html"/>
5 <xsl:preserve-space elements="*" />

```



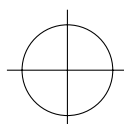


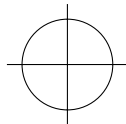
```

6
7 <xsl:template match="invitation">
8 <html>
9 <head>
10 <title> Invitation (XSL/CSS formatting) </title>
11 <link href="invit.css" rel="stylesheet" type="text/css"/>
12 <!-- 12 November 1998 mg -->
13 </head>
14 <body>
15 <h1>INVITATION</h1>
16 <table>
17 <tbody>
18 <tr><td class="front">To: </td>
19 <td><xsl:value-of select="@to"/></td></tr>
20 <tr><td class="front">When: </td>
21 <td><xsl:value-of select="@date"/></td></tr>
22 <tr><td class="front">Venue: </td>
23 <td><xsl:value-of select="@where"/></td></tr>
24 <tr><td class="front">Occasion: </td>
25 <td><xsl:value-of select="@why"/></td></tr>
26 </tbody>
27 </table>
28 <xsl:apply-templates/>
29 <p class="signature"><xsl:value-of select="@signature"/></p>
30 </body>
31 </html>
32 </xsl:template>
33
34 <xsl:template match="par">
35 <p><xsl:apply-templates/></p>
36 </xsl:template>
37
38 <xsl:template match="emph">
39 <em><xsl:apply-templates/></em>
40 </xsl:template>
41
42 </xsl:stylesheet>

```

Line 4 specifies that we want to generate HTML output, thus implicitly declaring HTML as the default namespace so that we do not have to prefix HTML instructions with an explicit namespace prefix. Lines 7–32 handle care of the document element `invitation`. In particular, the header of the HTML file is defined (lines 9–13) with a link to the CSS style sheet `invit.css` on line 11. A level 1 heading is inserted (line 15), and a table is started on line 16. This is to represent nicely the various attributes of the `invitation` element. For instance, on line 19 you can see how to get access to the attribute of an XML element by using XSL’s `xsl:value-of` element and the attribute `@` specifier on its `select` attribute. In this case, we are after the value of the `to` attribute, which is put in the right-hand cell of the first row of the table that contains the string “To: ” in the left-hand cell (line 18). The following pairs of lines 20 to 25 build rows for the other attributes of the `invitation` element. On lines 26–27 we end the table before instructing XSL first to process the children of `invitation` (line 28) and then to write an HTML `p` element of class `signature` with as content the value of the `signature` attribute of the `<invitation>` start tag (line 29). Lines 30–31 end the HTML document gracefully.





The remaining tasks are to handle the `par` and `emph` elements. These are dealt with in lines 34–36 and 38–40, respectively. Compare the procedure to the one described in Section 7.4.6.

The style sheet `invhtml2.xsl` and XML file `invitation2.xml` are presented to the `xt` processor.

`xt invitation2.xml invhtml2.xsl invhtml2.html`

We obtain the HTML file `invhtml2.html`, listed here. It is equivalent to the HTML file shown on page 308.

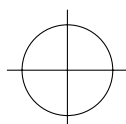
```

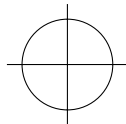
1  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2  <html>
3  <head>
4  <title> Invitation (sgmlpl/CSS formatting) </title>
5  <link href="invit.css" rel="stylesheet" type="text/css">
6  </head>
7  <body>
8  <h1>INVITATION</h1>
9  <table>
10 <tbody>
11 <tr><td class="front">To: </td>
12 <td>Anna, Bernard, Didier, Johanna</td></tr>
13 <tr><td class="front">When: </td>
14 <td>Next Friday Evening at 8 pm</td></tr>
15 <tr><td class="front">Venue: </td>
16 <td>The Web Cafe</td></tr>
17 <tr><td class="front">Occasion: </td>
18 <td>My first XML baby</td></tr>
19 </tbody>
20 </table>
21 <p>I would like to invite you all to celebrate
22 the birth of <em>Invitation</em>, my
23 first XML document child.</p>
24 <p>Please do your best to come and join me next Friday
25 evening. And, do not forget to bring your friends.</p>
26 <p>I <em>really</em> look forward to see you soon!</p>
27 <p class="signature">Michel</p>
28 </body>
29 </html>

```

7.6.6 Using XSL to generate formatting objects

As we have already explained several times, the best approach for setting up a style sheet for a given class of documents is to express it in terms of generic formatting objects. This way your style sheet can be translated into various output formats. We have shown that this works well in the case of DSSSL, where we used Jade as a processing engine (see Section 7.5.3). Similarly we can write an XSL style sheet using only XSL's formatting objects and translate those into various output representations. However, at present, because the XSL standard is not yet finalized, no general-purpose tool, such as Jade, is available. Nevertheless, we can use James Tauber's `fop` Java processor or Sebastian Rahtz' `PassiveTeX` [`↔`PASSIVETEX], which transform (a subset of) XSL formatting objects into PDF.



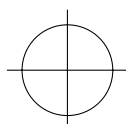


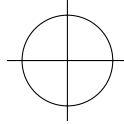
Let us first look at the XSL style sheet `invfo1.xsl` that we have prepared for use with the invitation XML source file.

```

1  <?xml version='1.0'?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:fo="http://www.w3.org/1999/XSL/Format">
4
5  <xsl:strip-space elements="*" />
6
7  <!-- Parameterizations -->
8
9  <xsl:variable name="PageMarginTop">75pt</xsl:variable>
10 <xsl:variable name="PageMarginBottom">125pt</xsl:variable>
11 <xsl:variable name="PageMarginLeft">80pt</xsl:variable>
12 <xsl:variable name="PageMarginRight">150pt</xsl:variable>
13 <xsl:variable name="BodyFont">Times-Roman</xsl:variable>
14 <xsl:variable name="BodySize">12pt</xsl:variable>
15 <xsl:variable name="TypeWriterFont">Computer-Modern-Typewriter</xsl:variable>
16 <xsl:variable name="SansFont">Helvetica</xsl:variable>
17 <xsl:variable name="ListRightMargin">12pt</xsl:variable>
18 <xsl:variable name="ListAbove">12pt</xsl:variable>
19 <xsl:variable name="ListBelow">12pt</xsl:variable>
20 <xsl:variable name="ListNormalIndent">15pt</xsl:variable>
21 <xsl:variable name="BulletOne">&#x2022;</xsl:variable>
22
23 <xsl:template name="listitem">
24   <xsl:param name="labeltext">labeltext</xsl:param>
25   <xsl:param name="itemid">itemid</xsl:param>
26   <xsl:param name="itemtext">itemtext</xsl:param>
27   <fo:list-item id="{ $itemid }">
28     <fo:list-item-label font-style="italic">
29       <fo:block>
30         <xsl:value-of select="$labeltext" />
31         <xsl:text>:</xsl:text>
32       </fo:block>
33     </fo:list-item-label>
34     <fo:list-item-body>
35       <fo:block><xsl:value-of select="$itemtext" /></fo:block>
36     </fo:list-item-body>
37   </fo:list-item>
38 </xsl:template>
39
40 <xsl:template match='/'>
41   <fo:root>
42     <fo:layout-master-set>
43       <fo:simple-page-master>
44         page-master-name="allpages"
45         margin-top="{ $PageMarginTop }"
46         margin-bottom="{ $PageMarginBottom }"
47         margin-left="{ $PageMarginLeft }"
48         margin-right="{ $PageMarginRight }"
49         <fo:region-body margin-bottom="100pt" />
50         <fo:region-after extent="25pt" />
51       </fo:simple-page-master>
52     </fo:layout-master-set>
53     <fo:page-sequence>
54       <fo:sequence-specification>
55         <fo:sequence-specifier-repeating>
56           page-master-first="allpages"
57           page-master-repeating="allpages" />
58         </fo:sequence-specification>
59         <fo:flow font-family="serif">
60           <xsl:apply-templates />
61         </fo:flow>

```



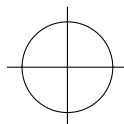


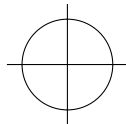
```

62     </fo:page-sequence>
63 </fo:root>
64 </xsl:template>
65
66 <xsl:template match="invitation/front">
67 <fo:block font-family="sans-serif" font-size="24pt"
68     font-weight="bold" text-align-last="centered"
69     space-after.optimum="24pt">
70 <xsl:text>INVITATION</xsl:text>
71 </fo:block>
72
73 <fo:list-block provisional-distance-between-starts="2cm"
74     provisional-label-separation="6pt">
75 <xsl:call-template name="listitem">
76 <xsl:with-param name="labeltext">To</xsl:with-param>
77 <xsl:with-param name="itemid">listto</xsl:with-param>
78 <xsl:with-param name="itemtext"><xsl:value-of select="to"/></xsl:with-param>
79 </xsl:call-template>
80 <xsl:call-template name="listitem">
81 <xsl:with-param name="labeltext">When</xsl:with-param>
82 <xsl:with-param name="itemid">listdate</xsl:with-param>
83 <xsl:with-param name="itemtext"><xsl:value-of select="date"/></xsl:with-param>
84 </xsl:call-template>
85 <xsl:call-template name="listitem">
86 <xsl:with-param name="labeltext">Venue</xsl:with-param>
87 <xsl:with-param name="itemid">listwhere</xsl:with-param>
88 <xsl:with-param name="itemtext" select="where"/>
89 </xsl:call-template>
90 <xsl:call-template name="listitem">
91 <xsl:with-param name="labeltext">0occasion</xsl:with-param>
92 <xsl:with-param name="itemid">listwhy</xsl:with-param>
93 <xsl:with-param name="itemtext"><xsl:value-of select="why"/></xsl:with-param>
94 </xsl:call-template>
95 </fo:list-block>
96 </xsl:template>
97
98 <xsl:template match="invitation/body/par">
99 <fo:block space-before.optimum="{BodySize}"
100 <xsl:apply-templates/>
101 </fo:block>
102 </xsl:template>
103
104 <xsl:template match="invitation/body/par/emph">
105 <fo:inline-sequence font-style="italic">
106 <xsl:apply-templates/>
107 </fo:inline-sequence>
108 </xsl:template>
109
110 <xsl:template match="invitation/back">
111 <fo:block space-before.optimum="{BodySize}"
112     font-weight="bold" text-align-last="end">
113 <xsl:text>From: </xsl:text>
114 <xsl:value-of select="signature"/>
115 </fo:block>
116 </xsl:template>
117
118 </xsl:stylesheet>

```

We first specify (line 3) that the fo namespace prefix addresses the XSL formatting object vocabulary which we will use to construct the result tree. We declare a series of variables to set various global formatting parameters (lines 9-21). In particular the variable BodySize (line 14) is equivalent to *FontSize* on line 8 of the





DSSSL style sheet of Section 7.5.2.2). They both control the default size of the document font. To show how named templates are handled in XSL, we define a named template `listitem` on lines 23–38. It has three parameters: `labeltext` (line 24), which contains the text for the list label (used on line 30), `itemid` (line 25), an identifier of the list item (used on line 27), and `itemtext` (line 26), which contains the text for the body of the item (used on line 35). Notice how the syntax `$name` is used to access the actual value of a parameter or variable.

The layout of the document is specified in the root template (lines 40–64). In particular, the page dimensions are defined (lines 45–48) using the variables set at the beginning of the style sheet (lines 9–12).

We then select the front part of the document (lines 66–96) and start by writing the text “INVITATION” in a sans serif 24 pt font centered on the output medium as a block object (lines 67–71). The rest of the front matter is displayed in a list (lines 73–95). The list has a label width of 2 cm to write the fixed texts (line 73). For each of the four elements inside the front element, we invoke the `listitem` named template defined on lines 23–38. Let us review in detail one of the invocations. On lines 75–79 we deal with the `to` element. First we set the `labeltext` parameter to the string “To”, the `itemid` identifier parameter to “listto”, and the `itemtext` parameter to the content of the `to` element in the XML source document, which is retrieved with the `<xsl:value of select="to"/>` syntax. Lines 80–94 handle the `date`, `where`, and `why` elements in a similar way.

We then go to the body part of the document, where for each paragraph (paragraph element type) we generate a block object (lines 99–101), skipping a space equal to the font size (line 99). The `emph` elements (lines 104–108) are transformed into a sequence using an italic font (lines 105–107).

Finally, for the back matter (lines 110–116) we create a block object (lines 111–115), where, after setting the space before, we require a bold font and right-justified text (line 112). We output the string “From: ” (line 113) followed by the contents of the `signature` element type (line 114).

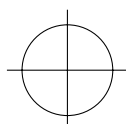
This style sheet `invfo1.xsl` and the XML source file `invitation.xml` can be compiled by James Clark’s `xt` tool, as follows:

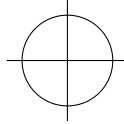
```
xt invitation.xml invfo1.xsl invfo1.fop
```

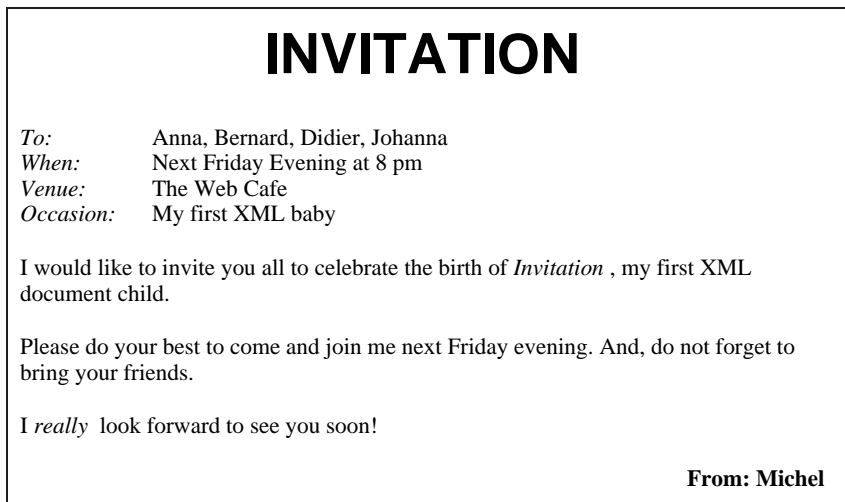
The file `invfo1.fop` contains the formatting objects and can now be input into James Tauber’s `fop` program, as follows:

```
java com.jtauber.fop.FOP invfo1.fob invfo1.pdf
James Tauber's FOP 0.11.0
successfully read and parsed temp.fob
successfully wrote invfo1.pdf
```

This program generates the PDF file `invfo1.pdf` shown in Figure 7.11. We also show the output generated by Sebastian Rahtz’ `PassiveTeX`, which translates





$$\text{XML} \xrightarrow{\text{XSL}} \text{FO} \xrightarrow{\text{FOP}} \text{PDF}$$


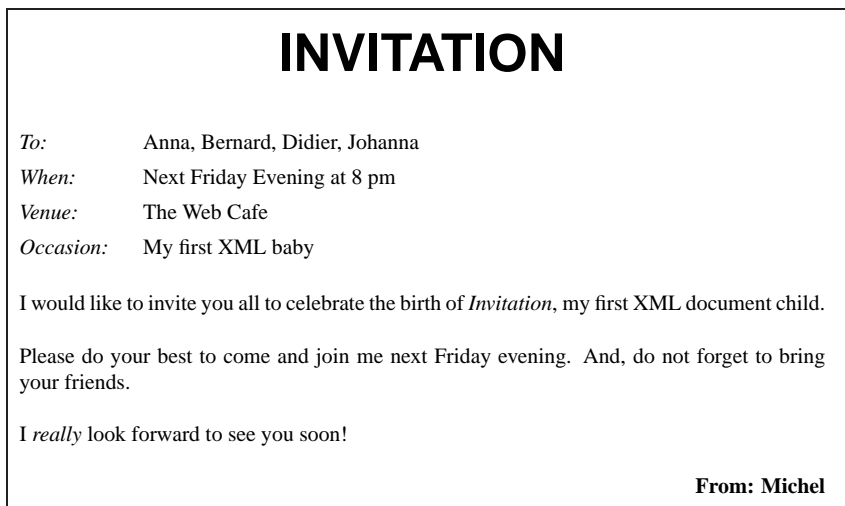
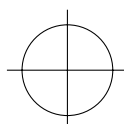
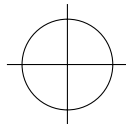
$$\text{XML} \xrightarrow{\text{XSL}} \text{FO} \xrightarrow{\text{PassiveTeX}} \text{PDF}$$


Figure 7.11: PDF generated from flow objects with fop and PassiveTeX





XSL formatting objects directly into PDF using the \TeX formatter. It is quite remarkable that, although we use a completely different style sheet language and data model, we obtain results that are quite similar to the one in Figure 7.8. This shows that XSL is already a powerful technology that allows us to use XML-based syntax and tools for all stages of our document handling.

7.6.7 XML, XSL and databases

Until now we have used XSL mainly to visualize information in an optimal way. However, we can use XSL also to query and manipulate data in a database. As an example we have prepared a database file containing a list of countries, their capital, their currencies, etc. In the following sections we will describe the data and show how to generate various reports.

7.6.7.1 The data format

The data are organized in a file `entable.xml` as follows:

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <countries>
3    <country>
4      <shortname>Afghanistan</shortname>
5      <fullname>Islamic State of Afghanistan</fullname>
6      <isocountry>AF</isocountry>
7      <capital>Kabul</capital>
8      <citizen>Afghan</citizen>
9      <adjective>Afghan</adjective>
10     <currency>afghani</currency>
11     <isocurrency>AFA</isocurrency>
12     <currensubunit>pul</currensubunit>
13   </country>
14   ...

```

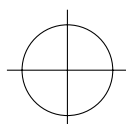
Each country element has nine children elements: `shortname`, `fullname`, ...

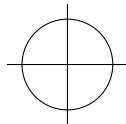
An alternate representation of these data is using attributes of the single element `country`. We transform the form using elements shown above into one based on attributes with the XSL style sheet `isotab1to2.xsl` that follows:

```

1  <?xml version='1.0' encoding="ISO-8859-1"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4  <xsl:output method="xml" encoding="ISO-8859-1"/>
5
6  <xsl:template match="/">
7    <xsl:element name="countries"><xsl:text>&#xA</xsl:text>
8      <xsl:for-each select="countries/country">
9        <xsl:element name="country">
10         <xsl:attribute name="shortname">
11           <xsl:value-of select="shortname"/>
12         </xsl:attribute>
13         <xsl:attribute name="fullname">
14           <xsl:value-of select="fullname"/>
15         </xsl:attribute>
16         <xsl:attribute name="isocountry">
17           <xsl:value-of select="isocountry"/>

```





```

18     </xsl:attribute>
19     <xsl:attribute name="capital">
20       <xsl:value-of select="capital"/>
21     </xsl:attribute>
22     <xsl:attribute name="citizen">
23       <xsl:value-of select="citizen"/>
24     </xsl:attribute>
25     <xsl:attribute name="adjective">
26       <xsl:value-of select="adjective"/>
27     </xsl:attribute>
28     <xsl:attribute name="currency">
29       <xsl:value-of select="currency"/>
30     </xsl:attribute>
31     <xsl:attribute name="isocurrency">
32       <xsl:value-of select="isocurrency"/>
33     </xsl:attribute>
34     <xsl:attribute name="currsubunit">
35       <xsl:value-of select="currsubunit"/>
36     </xsl:attribute>
37     </xsl:element><xsl:text>&#xA;</xsl:text>
38   </xsl:for-each>
39 </xsl:element><xsl:text>&#xA;</xsl:text>
40 </xsl:template>
41
42 </xsl:stylesheet>

```

There is only a single template that matches the root node (lines 6–40). We output the document element `countries` (lines 7–39), which includes thanks to the `xsl:for-each` loop (lines 8–38) all country elements and their attributes (lines 9–37). We get the content for each child element of the country element in the source file and transfer it into a attribute with the same name. For instance, on line 10–12 we create the `shortname` attribute using the `xsl:attribute` instruction, and give it the value of the `shortname` child element using the `xsl:value-of` instruction on line 11.

The start of the resulting file `entable-alt.xml` follows:

```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <countries>
3   <country shortname="Afghanistan"
4     fullname="Islamic State of Afghanistan"
5     isocountry="AF"
6     capital="Kabul"
7     citizen="Afghan"
8     adjective="Afghan"
9     currency="afghani"
10    isocurrency="AFA"
11    currsubunit="pul"
12  />
13   ...

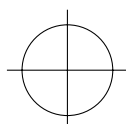
```

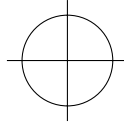
In fact we could have written a much shorter XSL style sheet to do this transformation by using the `xsl:copy` instruction, which copies the current node, its namespace but not its attributes and child elements.

```

1 <?xml version='1.0' encoding="ISO-8859-1"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4 <xsl:output method="xml" encoding="ISO-8859-1"/>

```





```

5
6 <xsl:template match="countries">
7   <xsl:copy>
8     <xsl:apply-templates/>
9   </xsl:copy><xsl:text>&#xA;</xsl:text>
10 </xsl:template>
11
12 <xsl:template match="country">
13   <xsl:copy>
14     <xsl:for-each select="*">
15       <xsl:attribute name="{name(.)}">
16         <xsl:value-of select="."/>
17       </xsl:attribute>
18     </xsl:for-each>
19   </xsl:copy>
20 </xsl:template>
21
22 </xsl:stylesheet>

```

The first template (line 6–10) copies the contents of the document element through. Since the `xsl:copy` instruction does not copy the child nodes, we need a second template (lines 12–20) to copy the content of the country element, selecting each of the child elements (line 14) and creating an attribute with the name of the child element in question (value of the `name` attribute is put to `name(.)` on line 15). The content of the current child node is transferred into the attribute with the `value-of` instruction and its `select="."` attribute (line 16).

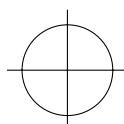
7.6.7.2 Using the database

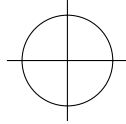
Now it is time to see how we can access and use the information in our database. For instance, we could list all capitals sorted alphabetically, together with the country and its ISO code. The XSL style sheet `isotab1exa1.xsl` to perform this task on the alternate attribute-based version of the database (`entable-alt.xml` on page 361) follows:

```

1 <?xml version='1.0' encoding="ISO-8859-1"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4 <xsl:output method="text" encoding="ISO-8859-1"/>
5
6 <xsl:template match="/countries">
7   <xsl:for-each select="country">
8     <xsl:sort select="@capital"/>
9     <xsl:if test="(@capital != '-' ) and (@capital != ' ')">
10      <xsl:value-of select="@capital"/>
11      <xsl:text> is the capital of </xsl:text>
12      <xsl:value-of select="@shortname"/>
13      <xsl:text> (ISO-code </xsl:text>
14      <xsl:value-of select="@isocountry"/>
15      <xsl:text>)&#xA;</xsl:text><!-- new line -->
16    </xsl:if>
17  </xsl:for-each>
18 </xsl:template>
19
20 </xsl:stylesheet>

```





7.6 Extensible Stylesheet Language

363

We loop over all countries (`xsl:for-each` of lines 7–17) and indicate we want to sort on the `@capital` attribute (line 8). Some countries or territories do not have a formal capital and they are flagged in the database with a hyphen or blank entry for the `capital` attribute. Thanks to the `xsl:if` statement (lines 9–16) we ensure that we use only valid entries, for which we write the information we want (lines 11–15). The entries are sorted alphabetically by capital, as the start of the resulting output shows:

```

1 Abu Dhabi is the capital of United Arab Emirates (ISO-code AE)
2 Abuja is the capital of Nigeria (ISO-code NG)
3 Accra is the capital of Ghana (ISO-code GH)
4 Adamstown is the capital of Pitcairn Islands (ISO-code PN)
5 Addis Ababa is the capital of Ethiopia (ISO-code ET)
6 Al aaiun is the capital of Western Sahara (ISO-code EH)
7 Algiers is the capital of Algeria (ISO-code DZ)
8 Alofi is the capital of Niue (ISO-code NU)
9 Amman is the capital of Jordan (ISO-code JO)

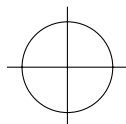
```

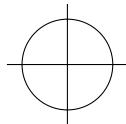
A rather more complicated example where we also introduce extensions follows. Note that using extensions will probably limit the portability of your style sheet between XSL applications. The style sheet uses the first variant (based on element children) of the country database (`entable.xml` on page 360).

```

1 <?xml version='1.0' encoding="ISO-8859-1"?>
2 <xsl:stylesheet version="1.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:date="http://www.jclark.com/xt/java/java.util.Date"
5   xmlns:xt="http://www.jclark.com/xt"
6   extension-element-prefixes="xt">
7
8 <xsl:template match="/">
9   <xsl:call-template name="newfile">
10    <xsl:with-param name="abc" select="'ABCDEFGHJKLMNOPQRSTUVWXYZ'"/>
11  </xsl:call-template>
12 </xsl:template>
13
14 <xsl:template name="newfile">
15   <xsl:param name="abc" />
16   <xsl:variable name="letter" select="substring($abc,1,1)" />
17   <xsl:variable name="file" select="concat($letter,'-currency.txt')"/>
18   <xt:document method="text" href="{ $file }" encoding="ISO-8859-1">
19     <xsl:for-each select="//country[starts-with(isocurrency, $letter)]">
20       <xsl:sort select="shortname"/>
21       <xsl:sort select="currency"/>
22       <xsl:variable name="Currency" select="normalize-space(string(currency))"/>
23       <xsl:if test="($Currency != '-') and ($Currency != '')">
24         <xsl:text>The </xsl:text>
25         <xsl:value-of select="currency"/>
26         <xsl:text> is used by </xsl:text>
27         <xsl:variable name="L1" select="substring(citizen,1,1)"/>
28         <xsl:choose>
29           <xsl:when test="$L1 = 'A' or $L1 = 'E' or $L1 = 'I' or
30             $L1 = 'O' or $L1 = 'U' ">
31             <xsl:text>an </xsl:text>
32           </xsl:when>
33           <xsl:otherwise>
34             <xsl:text>a </xsl:text>
35           </xsl:otherwise>
36         </xsl:choose>

```





```

37     <xsl:value-of select="citizen"/>
38     <xsl:text> living in </xsl:text>
39     <xsl:value-of select="shortname"/>
40     <xsl:text>.&#xA;</xsl:text><!-- new line -->
41     </xsl:if>
42   </xsl:for-each>
43   <xsl:call-template name="footer"/>
44 </xt:document>
45
46 <xsl:if test="string-length($abc) != 1">
47   <xsl:call-template name="newfile">
48     <xsl:with-param name="abc" select="substring($abc, 2)" />
49   </xsl:call-template>
50 </xsl:if>
51 </xsl:template>
52
53 <xsl:template name="footer">
54   <xsl:text>Last modification : Michel Goossens, </xsl:text>
55   <xsl:choose>
56     <xsl:when test="function-available('date:to-string') and
57       function-available('date:new')">
58       <!-- date format : Fri Dec 31 23:59:59 PDT 1999 -->
59       <!-- 1234567890123456789012345678 -->
60       <xsl:variable name="datetemp" select="date:to-string(date:new())"/>
61       <xsl:variable name="month" select="substring($datetemp,5,3)"/>
62       <xsl:variable name="day" select="substring($datetemp,9,2)"/>
63       <xsl:variable name="year" select="substring($datetemp,string-length($datetemp)-3,4)"/>
64       <xsl:variable name="Date" select="concat($day,' ', $month,'.', '$year)"/>
65       <xsl:value-of select="$Date"/>
66     </xsl:when>
67     <xsl:otherwise>
68       <xsl:text>11 Nov. 1999</xsl:text>
69     </xsl:otherwise>
70   </xsl:choose>
71 </xsl:template>
72
73 </xsl:stylesheet>

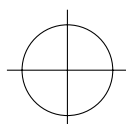
```

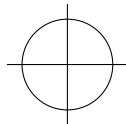
Since we want to use extension elements and functions provided by James Clark's `xt` program, we first must declare them. Thus the date namespace for an extension function is declared on line 4, while the extension element prefix `xt` and its namespace are declared on lines 5 and 6.

In this example we introduce also two new techniques: how to write multiple output files and how to use recursion. Since in XSLT one cannot use variables to store intermediate results (the value of an XSL variable cannot be changed) recursive techniques are needed in many circumstances.

The template for the root element (lines 8–12) merely calls the named template `newfile` (lines 9–11) and initializes a parameter `abc` by setting it to a string consisting of all 26 English uppercase letters (line 10). This string will be used to control the recursion by extracting each letter in turn.

The `newfile` named template (lines 14–51) writes a separate file for each set of currencies starting with a given letter of the alphabet. First we get the current first character of the parameter `abc` (line 16) and set the variable `file` to the concatenation of that letter and the string `“-currency.txt”` (line 17). Then we open a new file with as name the value of the `file` variable using the `xt:document` ex-





tension element declared on lines 5 and 6. We loop over all country elements and select those that have an ISO abbreviation equal to the letter we are considering (so for each letter in turn we do what is specified on lines 19–42 and the recursion makes sure we call the template for all letters, see below). The selected elements are sorted first by the name of the country (line 20) and then by currency (line 21). As we want to make sure that only valid entries are chosen we eliminate unwanted ones (lines 22–23). For correct entries we output the desired information (lines 24–40), where the only new construct is `xs1:choose`, which selects one among a number of possible alternatives. It consists of a sequence of `xs1:when` elements, whose `test` attributes are evaluated in turn and the content of the first true `xs1:when` is instantiated. If none of the tests is true, the contents of the `otherwise` branch is instantiated. In our case we want to generate the correct indefinite article (“an” before a vowel, “a” otherwise), and therefore the test on lines 29–30 checks whether the content of the `citizen` element starts with a vowel, in which case line 31 is instantiated, and line 34 otherwise. At the end of each page we call the footer named template (line 43) and close the output document (line 44).

Finally, we go one level deeper in the recursion by calling the same `newfile` template once more, but with the front character cut off the head of the `abc` parameter (lines 47–49). To make sure that the recursion terminates, we verify whether there is more than one character left in the `abc` string (if test on line 46 before calling the recursion template).

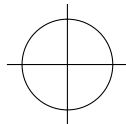
The named template `footer` (lines 53–71) adds a line at the bottom of each document. It uses `xt`’s `date` extension function, and we make sure that it exists on lines 56–57. If the function is available, we get the system date and manipulate it to obtain a form we like (lines 60–65), otherwise we just output a hard-wired date (line 68).

One of the 26 files—`A-currency.txt`, grouping information for all countries for which the ISO code for the currency starts with “A”—follows. Note the alternation of the indefinite article “an” (lines 1–9, 14) and “a” (lines 10–13, 15–16). The last line is generated by the named template `footer` discussed previously.

```

1 The afghani is used by an Afghan living in Afghanistan.
2 The lek is used by an Albanian living in Albania.
3 The readjusted kwanza is used by an Angolan living in Angola.
4 The Argentine peso is used by an Argentinian living in Argentina.
5 The dram (inv.) is used by an Armenian living in Armenia.
6 The Aruban guilder is used by an Aruban living in Aruba.
7 The Australian dollar is used by an Australian living in Australia.
8 The Austrian schilling is used by an Austrian living in Austria.
9 The Azerbaijani manat is used by an Azeri living in Azerbaijan.
10 The Australian dollar is used by a Christmas Islander living in Christmas Island.
11 The Australian dollar is used by a Cocos Islander living in Cocos (Keeling) Islands.
12 The Australian dollar is used by a Kiribatian living in Kiribati.
13 The Australian dollar is used by a Nauruan living in Nauru.
14 The Netherlands Antillean guilder is used by an Antillean living in Netherlands Antilles.
15 The Australian dollar is used by a Norfolk Islander living in Norfolk Island.
16 The Australian dollar is used by a Tuvaluan living in Tuvalu.
17 Last modification : Michel Goossens, 09 Nov. 1999

```



Summary

It should be clear from the discussion in this chapter that currently there are various interesting approaches to handling XML-tagged files. Most of the methods presented rely on interpreting the output from an XML parser (directly or indirectly) and applying “action” rules to the various XML input elements. Interpreters, such as Perl, Java, or Python, are available to drive the process of generating \LaTeX or HTML output. However, we stress once again that it is important to program the translation at a high level of abstraction, via the use of a \LaTeX class or CSS style sheet that can be reused by various document instances, even those constructed according to a different DTD.

We have looked in detail into three style sheet languages: CSS, DSSSL, and XSL. *Cascading Style Sheets* are linked strongly with the Web and are thus particularly well-suited for separating form and content for screen display. More recently the second edition of the CSS specification has added capabilities in the area of multimedia and printing.

Today the most complete formatting model is offered by the *Document Style Semantics and Specification Language*. It is an ISO standard and can handle complex page layouts, tables, and mathematics. It is the only possible choice if one needs high-quality typography of nontrivial documents.

The latest arrival on the style sheet market is the *Extensible Style Language*. It is supposed to offer, in due course, at least the functionality of both CSS and DSSSL, but with a syntax that is well integrated with the “X (extensible)” family of tools. Efforts are underway to ensure that the formatting models of all W3C activities, including CSS, XSL, and SVG, will converge. However, the XSL formatting model is not finalized yet, so that it is at present impossible to decide whether XSL will live up to its promises of offering a complete solution for handling XML documents in all circumstances.

In the few pages dedicated to each of these style sheet languages, we could only scratch the surface of each. Nevertheless, we have given you an idea of their basic functionality and general syntax. This should allow you to read and understand files written in any of these languages, modify them according to your needs, or even write your own.

The information and examples presented in this chapter should give you enough insight to decide which of the various possibilities (CSS, DSSSL, and XSL) is most useful for solving your problem today. Moreover, thanks to the references in the text, you can choose to follow the evolution of these tools. XSL and CSS are developing into full-blown style sheet languages, the former as a viable replacement for DSSSL for complex offline documents, the latter for multimedia applications on the Web.

