582304 The Metalanguage XML : Lecture Notes - Spring 2001

Lindén, Greger

University of Helsinki, Department of Computer Science
2001

Lindén , G 2001 , 582304 The Metalanguage XML : Lecture Notes - Spring 2001 . Series of Publications C , no. C-2001-1 , University of Helsinki, Department of Computer Science , Helsinki .

http://hdl.handle.net/10138/17729

acceptedVersion

582304 The metalanguage XML
Lecture notes — spring 2001

Greger Lindén

Department of Computer Science
University of Helsinki
Series of Publications C
Report C–2001–1

Contact information
Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

Computing Reviews (1998) Classification: I.7.2, I.7.4
Helsinki 2001

582304 The metalanguage XML: Lecture notes — spring 2001

Greger Lindén

Department of Computer Science, University of Helsinki
Report C–2001-1
March 2001
76+31 pages

Abstract

The eXtensible Markup Language (XML) is a generalised markup language based on SGML. The following lecture notes present the XML standard as well as event-driven XML processing and XML tree manipulation (SAX, DOM, XSL).

CR Categories and Subject Descriptors:

I.7.2 [Document and Text processing]: Document Preparation.
I.7.4 [Document and Text processing]: Electronic Publishing.

Keywords: XML, markup language, SGML, HTML, SAX, DOM, XSL, XSLT

Contents

1 Introduction

Computers handle data in many ways and formats, e.g., text, images, audio, and video. When exchanging data, there is often a need for a standardised format that many applications can read and write. One such standard is the **Extensible Markup Language (XML)** developed by the World Wide Web Consortium (W^3C) and released in 1999.

XML uses **tags** to mark logical parts or **elements** in a text (see Example). Every element is surrounded by a start tag (e.g., `<name>` and an end tag (e.g., `</name>`). Tags are useful when they are self-describing; the text is not only computer readable but also easily understood by human readers.

```
<!-- Example of part of document instance -->

<university>
   <department>
      <name>
         Department of Computer Science
      </name>
      <address>
         Teollisuuskatu 23
      </address>
   </department>
   <department>
      <name>
         Department of Mathematics
      </name>
      <address>
         Yliopistonkatu 5
      </address>
   </department>
</university>
```

In the example above, we have a small example of a **document instance**. (Indentation and line breaks are used to make the text more readable.). The instance (usually referred to only as 'document') contains a list of departments and their addresses. The tags have been properly named to be self-describing. Element names such as `university` and `address` give the human reader a clue how s/he should interpret the contents of the elements. As seen in the example, elements may contain **subelements** that in turn may contain subelements, etc. Note also how a **comment** is delimited by `<!--` and `-->`.

The markup specifies the **logical structure** as opposed to the **physical structure** of the document. The physical structure tells, e.g., how the different document parts are placed on a physical page (paper, screen, etc.).

An XML document instance must conform to several constraints. An **XML parser** reads an XML document instance and checks for errors. For example, all start tags must have corresponding end tags (Note: there is one exception to this), and elements may not be interleaved where an element partly overlaps another element, e.g., a start tag `<address>` may not be followed by the end tag `</name>` of another element, as in

Wrong! `<name><address>Teollisuuskatu 23</name></address>`.

Non-XML data, such as images in different formats may be stored separately as entities. An **entity manager** assembles an XML document instance by including entities stored in other files. XML parsers and entity managers are both examples of **XML processors**.

XML is actually a **metalanguage** that lets the user define a markup language. This gives the user the freedom to choose any element names s/he wants (there are some system names that are not allowed). Defining elements is done in a **document type definition (DTD)**. For each element, the user must give one rule specifying the name of the element and what its content may be. We have an example of a DTD below.

```
<!-- Example of part of document type definition -->

<!ELEMENT university (department+)>
<!ELEMENT department (name, address)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
```

In the example, we have defined the elements in the document instance example. Each element definition starts with the reserved word `<!ELEMENT`, followed by the element name and its contents. The plus operator (+) indicates that a `university` element may have one or more `department` subelements. A `department` element in turn consists of two subelements, a `name` element and an `address` element. Both these elements contain only text (defined by `#PCDATA`), i.e., they have no further subelements.

A DTD describes a **document type**. Every document conforming to this document type is an instance of the DTD. DTDs are usually defined in separate files and referenced to in XML document instances, but a DTD may also be included with an instance in a file.

XML markup should describe the nature of the object, instead of describing how elements are displayed and printed. XML markup can therefore be said to be **generalised**. It must be noted that XML markup says nothing about how the tags should be interpreted. XML markup describes the **syntactical structure** (form), not the **semantics** (meaning). Self-describing tags are useful for human readers, but a computer must still be explicitly told how to interpret that tags.

We also distinguish between **descriptive markup** and **procedural markup**. Markup is descriptive when it denotes logical structure only. Descriptive markup also may use self-describing tags to delimit elements. Procedural markup, overt (in old systems) or now usually covert, means formatting operations, such as character font descriptions and printing commands. Descriptive markup clearly separates, while procedural markup mixes, **content** and **form**.

The output format is specified in a **stylesheet**. For every document type, we may specify several stylesheets. By using alternative stylesheets for outputting the same document instance, the user may produce very different versions of the same document (Figure 1).

Stylesheets can be defined in several languages, e.g., **Cascading Style Sheets (CSS)** and **XML Stylesheet Language (XSL)**.

XML is useful for **data exchange applications**. It may be used as a format for relational

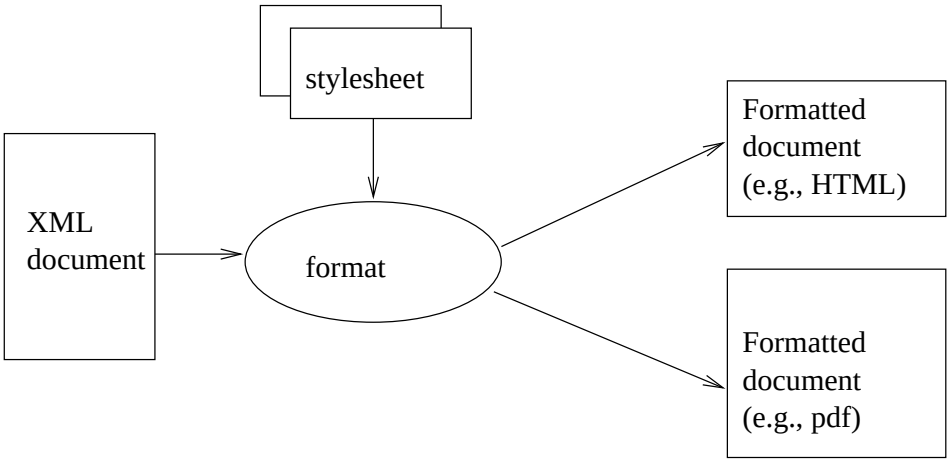


Figure 1: Formatting of an XML document

database systems. XML is also popular in **document publishing applications**, where a database of XML documents may be used as a source for assembling and publishing documents on different media. Related standards support hypertext links between documents and searching and finding document parts (**XLink**, **Xpath**, and **XPointer**), and formatting documents (**XSLT** and **XSL**). Other standards describe the interface through which XML documents may be accessed. Two such standards are **SAX** and **DOM**. We will describe these standards later.

The **Standard Generalised Markup Language (SGML)** is a similar standard for document markup, which was defined in 1986. SGML is a rather complex standard, providing a lot a different possibilities for defining document markup. Despite being available for such a long time (now 14 years), it has never been a success, mainly due to its complexity. There has also been a lack for simple, cheap, easy-to-use SGML processors.

The **Hypertext Markup Language (HTML)**, on the other hand has been a great success, due to the immense popularity of the World Wide Web. As a matter of fact, it has been so successful as a web page language that developers have provided their own feature extensions to the language aimed for use in their own browser. Users on the other hand have had (until recently) few possibilities to affect the outlook of HTML documents.

XML tries to combine the best aspects of these languages. XML is simple enough to use, and to build tools for. On the other hand it clearly separates logical form from physical, also giving the user the possibility to use self-describing element names and alternative stylesheets. Quite a few tools and processors have already been developed for XML. In the future, **XHTML**, an HTML version based on XML, is expected to replace HTML.

It must be said though, that XML may not be useful for all problems and their solutions. Simply presenting database contents on the web, or building complex web pages is at the moment done much more easily in HTML (including extensions and plug-ins). However, at the moment all three languages seem to be needed.

In this basic course on XML we will learn

- How to write an XML document instance.

- How to specify an XML document type definition (DTD).
- How to use XML processors, such as XML editors, XML parsers, formatters.
- How to write a style sheet for an XML file.

Other interesting subjects would be (if there is time) to learn

- How to use XML for defining a web page.
- How to use XML with Java.
- How to use XML as a format for exchanging data.

There are no required prerequisites for this course, but it does help if you are used to programming, word formatting and HTML.

In the next chapter, we will take a detailed look at XML syntax.

Literature and references

This chapter has been mainly based on Bradley's *The XML Companion* [Bra00].

There is a huge amount of literature on XML: beginners' guides, experts' guides, XML with Java, XML in publishing, etc. Here are only a few examples of what you can read beside these lecture notes. Bradley [Bra00] gives an introduction to the subject. Many parts of these lecture notes are based on his book. A book by Goldfarb and Prescod [GP00] works best as a reference manual due to its extent.

XML is widely described on the web. Some useful pages to look at for a starters are:

- `http://www.w3.org/TR/REC-xml` (XML definition)
- `http://www.xmlinfo.com`
- `http://www.xmlsoftware.com`
- `http://www.xslinfo.com`

I also recommend *Johdatus XML-tekniikkaan* on the web page

`http://www.cs.Helsinki.FI/u/ruini/structure/xml/`

by Henry Ruini. Henry gives an introduction to XML **in Finnish** and has translated XML terms into Finnish.

Relevant information for the course, e.g., **example files** and **programs** in these lecture notes, is collected at

`http://www.cs.Helsinki.FI/u/linden/opetus/xml/index.html`

Exercises

E-1-1 Give examples of a) descriptive markup languages, b) procedural markup languages, and c) meta markup languages.

E-1-2 Why do you want to separate logical structure from physical layout?

E-1-3 Why do you think DTDs are defined in separate files, not in the same files as document instances?

E-1-4 Visit the URLs listed above. What useful information do you find?

E-1-5 Search the web for a) XML beginners' guides b) the complete XML standard.

2 Document instances

In this chapter we will learn how to write an XML document instance.

2.1 Elements

In an XML document, we use **tags** to separate **logical elements**. A logical element may consists of any text string and/or subelements in the document. We have, e.g., the following element

```
<capital>Helsinki</capital>
```

The element starts with a **start tag** `<capital>` and ends with an end tag `</capital>`. Between the tags, we find the contents `Helsinki` of the element. Note that an element includes its start and end tags as well as its contents. A tag always start with a less-than character (`<`) or the string `</` and ends with the greater-than character (`>`, or in some cases `/>`). These characters/strings are also called delimiters. The **element name** is given between these characters (strings) and is context-sensitive, e.g, the start tag `<capital>` differs from the start tag `<Capital>`. A start tag must always have a corresponding end tag (with the same name). Element names may include small or capital letters, hyphens, underscores, periods, colons and digits. An element name must always start with a letter, an underscore or a colon, but there is no restriction on its length.

In the above example, the element includes text as content. But, an element may also include subelements as in

```
<country>
        <cname>Finland</cname>
        <capital>Helsinki</capital>
</country>
```

where we have a `country` element including two elements, a `cname` and a `capital` element. If an element includes other elements and only elements, we say that it has **element content**. If it includes only text (as the `cname` and `capital` elements), we say that it has **data content**. An element containing both element and data content has **mixed content**. As we shall see later, e.g., spaces and line-ends included in elements with either element or data content are usually processed differently.

Subelements included in elements must always be completely enclosed in the (super)element (i.e. elements are **nested**). For example, in the following example

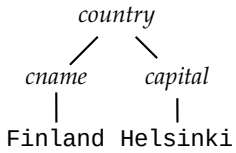
```
<small_example>
        <first>this is<second></first>so wrong</second>
</small_example>
```

two elements, the `first` and the `second` elements are interleaved, and the example does not form a **well-formed**[1] document instance. A document is well-formed when it

[1] The concept well-formedness differs from the concept validity, which is defined in the next chapter.

has exactly one root element that contains all other documents and when all start tags have corresponding end tags and and all subelements are properly contained in their superelements (i.e., no interleaving is allowed).

A document instance then always forms a **hierarchy of elements**, where we have one **root** element in which all other elements are included. Therefore, XML document instances are often presented as trees (which are hierarchical!). The well-formed example above may be presented as



A tree is a collection of **nodes**, one of which is distinguished as the **root**, and a relation (ancestor) that places a hierarchical structure on the nodes. In this tree, the `country` element represents the root, and the subelements `cname` and `capital` **children** of the root, which thereby is their **parent**. Nodes that have the same parent are called **siblings**. Nodes that do not have any children are **leaves**. In the tree above, the data contents of the data elements `cname` and `capital` have been represented as children. (We have used only the element names, not the tags, to **label** the tree; a tree is **labelled** when the nodes are given names.)

An element may have empty content as in

```
<nothing></nothing>
```

Empty elements can (and usually should) be represented with a single tag as

```
<nothing/>
```

Note that the tags slightly differs from the start and end tags above; here we have a slash at the end of the tag.

But why do we need empty tags. There are several reasons. Some times the real contents of the elements is found elsewhere (outside the document instance), and the (empty) element only includes a reference to where the contents are as in the following `image` element

```
<image file="small_picture.jpg"/>
```

An XML processor will read the empty tag and its reference and replace it with the image found in the referenced file.

If we for some reason also want to mark document parts that are not properly nested, we can do it with empty tags, e.g.,

```
<another_small_example>

        <paragraph>This is <revised_start/> a text that has
```

```
        been revised.</paragraph>

        <paragraph>And the revisions<revised_end/> spans
        several paragraphs.</paragraph>

</another_small_example>
```

In the above example, the revised text spans several paragraphs. We cannot have a start tag `<revised` and an end tag `</revised>` replacing the empty tags `<revised_start/>` and `<revised_end/>` (why not?).

We can create an XML document instance by simply inserting two tags, a start tag at the beginning and an end tag at the end, in any ordinary text document. But more useful may be to use finer **granularity**. Markup is useful for presenting the document in different formats, and also for structural search. If we use the following granularity

```
<usa>
<state>
        Washington
        Seattle
</state>
<state>
        Washington D.C.
        Washington
</state>
</usa>
```

we can look for the string `Washington` in the full text and find all occurrences, i.e., both states listed above. But if we use finer granularity, such as

```
<usa>
<state>
        <state_name>Washington</state_name>
        <capital>Seattle</capital>
</state>
<state>
        <state_name>Washington D.C.</state_name>
        <capital>Washington</capital>
</state>
</usa>
```

we can look for states whose capital is Washington and find only the correct occurrence (i.e., the state Washington D.C., not the state Washington whose capital is Seattle). An XML processor may also format the subelements differently, such

Washington: *Seattle*

Washington D.C.: *Washington*

Usually, writes of XML documents are recommended not to use lesser-than or greater-than characters in data content in order not to confuse data with tags. Instead there are

certain reserved entities, such as `<` and `>` to use instead. However, many XML processors can handle these characters. For example, an XML editor lets the user write `2 < 3` as data content which is also the string shown on the screen (paper etc.). But internally, the editor may replace this string with `2 < 3`.

An XML processor can check if a document instance is well-formed. It would be useful, if we could also define what type of elements an instance may contain. This is done in a **document type definition (DTD)** which we define in the next chapter. But first we shall take a look at some other concepts that may be included in an XML document instance.

2.2 Attributes

Elements may have **attributes**. An attribute holds additional information about the element. An attribute and its value is given in the start tag of the element. For example, we have the following example.

```
<book author="Albert Einstein">
...
</book>
```

The `book` element contains one attribute with name `author` and whose value is a string `Albert Einstein`. **Attribute names** are also case-sensitive and follow the same restrictions as element names. As element names, they may contain letters, digits, periods, colons, hyphens, and underscores and must begin with a letter, an underscore or a colon. An attribute value may be any string surrounded by single or double quotes. Single or double quotes in the value may be included by using entities `&quot;` (double quote) and `&apos;` (single quote).

Note that attributes are always **single-valued**. If an attribute has several values, they are are specified in one string, such as

```
<book keywords="XML SGML">
...
</book>
```

In this case it is up to the XML processor to analyse the string value and extract two keywords.

There are a few **reserved attributes** in XML. The attribute `xml:lang` specifies a language for an element. Its values should be taken from the standard ISO 639 defining languages (see Appendix B). Some language codes are `en`, `fi`, `sv` for English, Finnish and Swedish. We can have for example

```
<product>
<paragraph xml:lang="en">
...
</paragraph>
<paragraph xml:lang="fi">
...
</paragraph>
</product>
```

for a passage in a book containing descriptions of a product in two different languages, English and Finnish.

The language code may be combined with a country code from the standard ISO 3166 (see Appendix C). Some examples of country codes are `GB`, `US`, `FI` and `SE` for Great Britain, the United States, Finland and Sweden. We may have

```
<product>
<paragraph xml:lang="en-GB">
What colour is it?
</paragraph>
<paragraph xml:lang="en-US">
What color is it?
</paragraph>
</product>
```

for specifying a `paragraph` element in British or American English or

```
<paragraph xml:lang="sv-FI">
...
</paragraph>
```

for specifying Finnish Swedish.

The content may also be a user-defined code in which case it must be preceded by `x-` (such as `x-turku_dialect`) or it may be one registered with IANA (Internet Assigned Number Authority) in which case it must be preceded by `i-` (such as `i-yi`, meaning Yiddish).

White space (spaces, line ends, tabs) are usually a problem in XML documents when they are processed. Should the white space be included in the output or removed. By default, space included in elements with only element content is considered **insignificant space** (and is not included in any processing) while space included in elements with data or mixed content is considered **significant**. The reserved attribute `xml:space` lets the user override the default interpretation. A value of `preserve` will preserve the white space in any element. (The other value is `default` and does not have to be set if the user does not want to change the default interpretation.)

We will look at how to define attributes and their value types in the next chapter.

2.3 Entities

Entities are units of information that may be stored in separate files. Entities are used to allow duplication or to allow different representations (other than XML data). Entities are also used for splitting large documents into subdocuments.

Internal text entities represent constant strings. This is a convenient way to use a shorter reference to a long string that may be used several times. A text entity may also be **external** (located outside the document). Such an entity contains (also called **parsed entity**)

Entity code	corresponding character
>	>
<	<
"	"
'	’
&amps;	&

Table 1: Some predefined entities

contains replacement text. External entities can also be **binary entities** containing drawings, pictures, sound, video, etc. Finally we distinguish between **general entities** which are used in document instances and **parameter entities** which are allowed in DTDs only.

Assume the text entity `&univ;` contains the string `University of Helsinki` and the text entity `dep;` the string `Department of Computer Science`. Then we can write

```
I study at the &dep; of the &univ;.
```

In order to use such entities, they must be defined. We will study the definition of entities in the next chapter.

Some entities have been predefined (see Table 1).

Many XML editors are able to hide to entity codes from the user and use corresponding characters instead. Predefined entities are used as text entities.

```
The symbol &lt;capital&gt; is an XML start tag.
```

An XML processor must of course be able to process entities correctly, e.g., replace text entities with replacement text before outputting a document and including binary entities (e.g. a picture) and presenting it in a required way in the document.

2.4 Processing instructions

Processing instructions allow documents to contain instructions for applications. A processing instruction is surrounded by delimiters `<?` and `?>` and contains a keyword (what application does the instructions concern) and and information (what should the application do).

All XML documents should include an XML declaration in the form of a processing instruction specifying what version of XML is used, what character encoding the document uses and where its DTD can be found (if it has one). An example

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

This declaration says that the version used is 1.0, that the character encoding is UTF-8 and the standalone parameter tells that there are no external definitions that need to be taken into account when processing this document.

Literature and references

The best source for XML syntax is of course the W3C recommendation found at

`http://www.w3.org/TR/REC-xml`.

Readers are referred to this definition for a more detailed description of XML.

Exercises

E-2-1 What distinguishes a well-formed document from an "unwell-formed" one? Give examples.

E-2-2 What XML markup would you suggest for the following telephone catalog?

```
Suomalainen Kaisa arch. Pelimannink 679 .............3241 9884
  Kalle Tuusula Hiekkaharju Koivukylä Koivut 5 J 6..4110 8956
  Kari and Rita Korppaanmäent. 21 B ................ 345 4302
Suomela Maila salesman  ............................ 789 986
```

E-2-3 What XML markup would you suggest for the following description?

The Department of Computer Science was founded in 1967 when the first professorship in computer science was established at the University of Helsinki. The Department belongs to the Faculty of Science, and it is located in Vallila, Helsinki. In Finland, the Department is the largest computer science department in a full-fledged university.

The educational programme of the Department is divided into three subprogrammes: Computer Science, Applied Computer Science and Teacher in Computer Science. The subprogramme Computer Science is further divided into five specialisation areas, which correspond to the research areas of the Department: algorithms and data structures, intelligent systems, software engineering, distributed systems and data communication, and information systems.

The aim of the programme is twofold. Firstly, the programme strives to provide a modern and all-round advanced level education of computer experts for the needs of the industry sector. Secondly, the Department seeks to be among the top-most research institutes within selected research areas of computer science.

Education at the Department is based on the established core computer science but adapts to new needs of the evolving discipline. The combination of theory and practice is stressed, and special emphasis is given to software design and engineering.

In research and in education, cooperation with the rest of the university and with the industry sector is considered to be of great significance. The Department has joint research activities with several departments of the University of Helsinki and with several information-technology oriented departments of other universities. In addition, the Department has research cooperation with approximately 40 different companies (1999).

3 DTD syntax

A **DTD** or **document type definition** defines the structure of a document (or document type). It contains rules that regulate how different elements and attributes relate to each other: what subelements (children) may an element have, which parents are possible, etc. In short, what hierarchy of elements should be allowed in a document instance. A DTD also tells what attributes elements can have, and what default values these attributes may have.

We have seen earlier that a document that conforms to certain restrictions is a **well-formed document**. In a well-formed document, there is one element that represents the root of the hierarchy and all other other elements are properly contained in the root element and/or other elements. A well-formed document is not dependable on a DTD.

When a document instance conforms to a certain DTD (i.e., it follows all the rules specified in the DTD) it is said to be **valid**. Validity is usually checked with an XML parser that checks well-formedness as well. As a matter of fact, it may be hard to find an XML parser that checks only well-formedness.

In short, well-formedness does not depend on a DTD; we can always say if a document instance is well-formed or not, even if we do not know if it has a DTD or not. Validity always requires that the document instance has a corresponding DTD against which the instance can be checked.

A DTD is optional, but usually recommended. A DTD makes XML processing easier, when we know what structures a set of document instances may have.

In the following, we shall take a look at defining DTDs, and especially see how we define elements and their possible contents, attributes of elements and value domains and default values of attributes, and entities.

3.1 Element definitions

An element is defined in an **element declaration**. We have the following example

```
<!ELEMENT country (capital)>
```

An element declaration starts with the delimiter `<!` and the keyword `ELEMENT` and ends with the delimiter `>`. After the keyword follows the **element name** (in this case `country`) and the **element content**. Element content declaration, called the **content model**, is surrounded by delimiters `(` and `)`. This declaration says that a `country` element has one subelement, a `capital` element (no more, and no less).

An element may also have several subelements. These elements are declared as a **sequence** in the content model separated by a comma (`,`). For example, in the declaration

```
<!ELEMENT country (cname, capital, population)>
```

we have a `country` element that contains exactly three subelements, a `cname` element, a `capital` element, and a `population` element, in that order.

An element may have **alternative** subelements, separated by a bar (|). For example, we have the declaration

```
<!ELEMENT country (cname | official_name)>
```

where the `country` is required to have exactly one subelement, either a `cname` element or a `official_name` element, but not both.

An element may also be declared to have **optional** subelements and/or repeatable subelements. Optional elements are marked by a question mark (?) as in

```
<!ELEMENT country (cname, capital, population?)>
```

where the `country` element may have two or three subelements — the `population` element is optional.

Repetition is specified by a star (*) or a plus (+); A star means that a subelement may be repeated zero (i.e. it is optional) or more times; a plus means that a subelement may be repeated one or more times. For example, in

```
<!ELEMENT country (cname, capital, city*, neighbour_country+)>
```

the `country` element may have zero or more `city` subelements, but it must have at least one (or more) `neighbour_country` subelements.

Elements are grouped by parentheses in a content model. A **repeatable group**, e.g., is specified by

```
(city, city_population)*
```

Elements may also have **data** content. Data is specified by the reserved word #PCDATA, which stands for **parsable character data**. We have, e.g.,

```
<!ELEMENT cname (#PCDATA)>
```

which declares that a `cname` element contains text (instead of subelements).

An element that may contain only other elements are said to have **element content**. If it contains only data it is said to have **data content**. If it may contain both, it has **mixed content**. We have, e.g.,

```
<!ELEMENT paragraph (#PCDATA|sub|super)>
```

where a `paragraph` may contain either data, a `sub` element or a `super` element. In the element declaration, the first token in the content model must always be #PCDATA when the element has mixed content.

An empty element is declared by

```
<!ELEMENT image EMPTY>
```

This means that a valid document may only contain `<image/>` (empty) elements, but not, e.g., `<image></image>` elements.

Finally, an element may be declared to have any content, as in

```
<!ELEMENT some_element ANY>
```

This means that the `some_element` may contain any other elements that are defined in the DTD.

Finally let's take a look at a short DTD for describing a dictionary.

```
<!ELEMENT dictionary (word_article*)>
<!ELEMENT word_article (head_word, pronunciation, sense+)>
<!ELEMENT head_word (#PCDATA)>
<!ELEMENT pronunciation (#PCDATA)>
<!ELEMENT sense (definition, example*)>
<!ELEMENT definition (#PCDATA)>
<!ELEMENT example (#PCDATA)>
```

A valid document instance may contain elements such as

```
<document>
  <word_article>
    <head_word>
      kill
    </head_word>
    <pronunciation>
      kil
    </pronunciation>
    <sense>
      <definition>
        If you say that you will kill someone ...
      </definition>
      <example>
        I'll kill you if you don't ...
      </example>
      <example>
        I could have killed them by accident.
      </example>
    </sense>
    <sense>
      <definition>If you kill time, ...
      </definition>
      <example>
        He spent long hours keeping out of the way, killing time.
      </example>
  </word_article>
```

```
  <word_article>
    <head_word>
      laps
    </head_word>
    <pronunciation>
    læps
    </pronunciation>
    <sense>
      <definition>
        A lapse is a period of time.
      </definition>
    </sense>
  </word_article>
</document>
```

3.2 Attribute definitions

Attribute of an element are defined in an **attribute list**. All attributes of an element are usually defined in one list. If there are multiple definitions of an attribute of the same element, the first definition takes precedence. We have the following example of an attribute list

```
<!ATTLIST country population NMTOKEN #IMPLIED
                  language   CDATA #REQUIRED
                  continent  (Europe | America | Asia ) "Europe">
```

An attribute list starts with the string `<!ATTLIST` followed by the name of the element to which the attributes belong, in this case `country`. After the element name follows a list of attribute names, their value types and their default values. Attribute names are restricted in the same way as element names. In the example above, the `country` element has three attributes, `population`, `language`, and `continent`.

The second parameter describes the value type. In XML, there are the following value types:

`CDATA`
`NMTOKEN`
`NMTOKENS`
`ENTITY`
`ENTITIES`
`ID`
`IDREF`
`IDREFS`
notation
name group

Attributes of the type `CDATA` may contain values that are (any) text strings, e.g.

```
<country language = "American English">
```

The NMTOKEN type indicates any (single) token or word that is formed as element names except that it may start with any allowed character, e.g., a digit. The NMTOKENS type indicates that the attribute may have multiple values. These values must be separated by spaces. Regarding the example attribute list, we could have

```
<country population = "100">
```

Assume that there is an attribute coordinates that has been declared NMTOKENS. Then we could use it in the following way.

```
<country coordinates = "60 25">
```

The ENTITY type indicates that the value is actually an entity (Entities, see next subsection). An element may be a placeholder for an image as in

```
<!ENTITY mypicture "123.jpg">
<!ELEMENT pic EMPTY>
<!ATTLIST pic picfile ENTITY ...>
```

In a document instance, we may write

```
<pic picfile="mypicture">
```

and an XML processor would know where to find the image.

As in the case with NMTOKENS, an attribute can be declared to have multiple entities as its values (with ENTITIES).

The value types ID, IDREF and IDREFS are used for hypertext linking.

The name token group restricts values to a finite set of values. In the case above, the continent attribute may only take values from the enumerated list following the attribute name.

All attribute values are case-sensitive.

The third parameter in the attribute list is the default value of the attribute. A default value #REQUIRED means that every time the element is used in an instance, the attribute must be given a value. If the default value is #IMPLIED, the attribute may be absent from the element. In the example above we may have

```
<country population="5" language="Finnish" continent="Europe">
```

and also

```
<country language="Finnish" continent="Europe">
```

but we may not leave out the `language` attribute because it is declared `#REQUIRED`. Note also that we cannot write

```
Wrong! <country population language="French" continent="Europe">
```

If an attribute name is given, also its value must be given.

A name token group can be followed by a specific default value as in the case above where the `continent` attribute can take three values, out of which the token `Europe` is the default value. If the attribute is absent in an element, the default value is assumed. In the example above, we could have

```
<country language="Finnish">
```

instead and the attribute `continent` would be assumed to have the value `Europe`. Note that required or implied attributes cannot have a specific default value.

Attributes can also have constant values. These values are preceded by `FIXED` as in

```
<!ATTLIST country position #FIXED "independent">
```

It might seem strange to use constant attributes, but they are useful in some cases not discussed here.

An element can have multiple attribute lists. If an attribute with the same name is declared in a list appearing later, the first declaration takes precedence.

3.3 Element or Attribute?

It is sometimes difficult to say, when designing a DTD, if one should use an element or an attribute to describe something. The rule of thumb says: Use an element, if the structure may contain substructures, otherwise use an attribute, especially if it concerns administrative information not shown in the output. Note also that attributes can be constrained and validated, but text in elements cannot. On the other hand, e.g., style sheets may have limited capabilities for processing attributes.

3.4 Entity definitions

An XML document may be divided into several separate files. In this way, it is easy to reuse a document part in several XML documents. Each unit is called an **entity** and must be assigned a name to recognise it.

Only the top document, the **document entity**, does not have a name. The document entity includes references to other entities located outside it. A parser will be given the document entity to read and check; the parser will know from the entity references how to assemble to complete document. The document entity may also be the only entity involved (i.e., the document is completely included in one file).

All entities must be defined using **entity declarations**. Entities are used through **entity references**. Entities can consists of XML data or some other format (binary). The content of the entity may be stored within the main document (**internal entity**) or in a separate file (**external entity**). An entity that contains XML data is said to be a **parsed entity** (either internal or external) because it can be validated by an XML parser.

Internal text entities represent a pre-defined constant string. The user may define a short entity name representing a long string; instead of repeating the long name s/he ony gives the entity reference wherever the string is needed in the document.

An internal text entity is defined in the following way.

```
<!ENTITY univ "University of Helsinki">
```

The declaration begins with the delimiter `<!` and reserved word `ENTITY`. Thereafter follows the entity name and the replacement text, and finally the delimiter `>`. Whenever the user wants to include the replacement text in the document, s/he only has to repeat the entity reference starting with an ampersand and ending with a semi colon such as

```
I study at the &univ;.
```

An XML processor (parser) will know to replace the entity reference with the replacement text and output

```
I study at the University of Helsinki.
```

The replacement text can be surrounded by either double or single quotes. If a quote should be included in the replacement string, the entity can either be defined with the other sort of quotes, or the entity may include references to built-in entities. For example, the declaration

```
<!ENTITY sentence 'His foot is 12" long'>
```

can be replaced by the declaration

```
<!ENTITY sentence "His foot is 12&quot; long">
```

Built-in entities There is a small number of built-in entities (see previous chapter). Within text, the user should use references to these built-in entities instead of the actual strings themselves. Especially, care should be taken not to include double quotes in an entity declaration surrounded by double quotes (and correspondingly with single quotes).

Character entities A character entity contains one character and its reference begins with an ampersand and a hash symbol. For example, the reference

```
&#60;
```

refers to the symbol <, the character with the decimal value 60 in the ASCII character set.

Parameter entities make it possible to define replacement texts also in the DTD. A parameter entity may be used only in the DTD! A parameter entity is defined as

```
<!ENTITY % parapart "(emph | supersc | subsc)">
```

It is used, e.g., in an element rule, in the following way.

```
<!ELEMENT paragraph (%parapart | bold)>
```

Especially, note the use of the percent character. The element declaration is equivalent to the declaration

```
<!ELEMENT paragraph (emph | supersc | subsc | bold)>
```

A parameter entity is useful when many element declarations contain similar parts.

External text entities An external entity must be defined with a **system identifier**, a reference to the file where it is located. The reference must be preceded by the `SYSTEM` keyword such as in

```
<!ENTITY myfile SYSTEM "/extra_files/file.xml">
```

(The system identifier can also be a public identifier, a general reference to more remote information. In this case, the entity is declared using the `PUBLIC` keyword. An application must then map the entity reference against a catalog in order to find the entity.)

Binary entities A binary entity may contain, e.g, a picture or or some other non-XML data. Including the system identifier (the location), we must also specify the format of the binary data as in

```
<!ENTITY myphoto SYSTEM "/figures/photo.gif" NDATA GIF>
```

References as above cannot be used in this case. Instead an empty element is used with an attribute of type `ENTITY`:

```
Take a look at my photo <picture name="myphoto"/>.
```

Usage of entities

A general entity such as a text entity reference may be used in an element but not in DTDs. Parameter entities may be used only in DTDS. There may be entity hierarchies, i.e., an entity declaration may contain the reference to another entity (but no cyclic declarations!).

No textual references are allowed if the entity contains non-XML data. The only place such an entity reference can occur is as an attribute value.

Parameter entities may not be used in a document instance, only in a DTD.

3.5 Processing instruction definitions

Processing instructions are not defined.

3.6 Document type declaration

If a certain DTD has been used in an XML document, it should start with a **document type declaration**. The declarationm identifies the document element (the root element) of the instance. In its simplest form, the declaration looks like

```
<!DOCTYPE catalog>
```

The declaration is delimited by `<!` and `>` and includes the reserved word `DOCTYPE` and the name of the root element, which would follow the declaration (containing then all the other elements). If the DTD is inlcuded in the same file as the instance, we would have

```
<!DOCTYPE catalog [
... The catalog DTD appears here ...
]>
```

If the DTD is located in a separate file, there must be a reference from the instance to the DTD. In that case we have

```
<!DOCTYPE catalog SYSTEM "catalog.dtd">
```

The file where the DTD is defined is specified after the `SYSTEM` reserved word.

Literature and references

This chapter has been mainly based on [Bra00].

Exercises

E-3-1 Write a DTD for the markup you made for the text in example E-2-2.

E-3-2 Write a DTD for the markup you made for the text in example E-2-3.

E-3-3 Give examples of different entity types. (This task will require you to look things up in an XML book, or on the web).

E-3-4 Why is the following document instance not valid? (The DTD is given in Section 3.1.)

```
<document>
  <word_article>
    <head_word>
      kill
```

```
    </head_word>
    <pronunciation>
      kil
    </pronunciation>
    <sense>
      <definition>
        If you say that you will kill someone ...
      </definition>
      <example>
        I'll kill you if you don't ...
      </example>
      <example>
        I could have killed them by accident.
      </example>
    </sense>
    <sense>
      <definition>If you kill time, ...
      </definition>
      <pronunciation>
        kill
      </pronunciation>
      <example>
        He spent long hours keeping out of the way, killing time.
      </example>
  </word_article>
  <word_article>
    <head_word>
      <word>
        laps
      </word>
    </head_word>
    <pronunciation>
    læps
    </pronunciation>
    <example>
      Hours lapsed between the phone calls.
    </example>
  </word_article>
</document>
```

E-3-5 XMLpro

Try out the XMLPro editor by Vervet Logic (`www.vervet.com`). You find a demo version in the directory `/home/group/xmltools/editors/xmlpro/`

Start the editor with the command

```
java -jar /home/group/xmltools/editors/xmlpro/xmlpro.jar
```

You find example XML files in `/home/group/xmltools/editors/xmlpro/Examples/`

Answer the following questions

1. Load an example file, e.g., `catalog.xml`. Take a look at the XML document instance and the DTD. What happens when you insert an element? IN the wrong place (according to the DTD)?

2. Create your own small DTD and instance, e.g., based on earlier exercises.

E-3-6 Morphon XMLEditor

Try out the Morphon XML editor in the same way as above. You find a demo version in the directory `/home/group/xmltools/editors/morphon/`.

Start the editor with the command

`/home/group/xmltools/editors/morphon/XMLEditor/runXMLEditor_1.2.sh`

E-3-7 Read in and check the document instance in the Exercise E-3-1 (or E-3-4 available at the course home page, see exercise 2) and its corresponding DTD in some XML editor. Did you find all the faults the first time you saw the instance? Does the editor help finding other errors?

E-3-8 Take a look at `http:www.xmlsoftware.com`. What other XML editors can you find. Try out a few.

4 Document modelling

DTDs are very important. They show authors of XML documents what structure the instances should have. They are also needed when validating document instances. If the DTD is not appropriate or suitable for the application, or if it simply contains the wrong declarations, the consequences could be disastrous. A change in a DTD at a later stage, e.g., of including an obligatory element or attribute, would mean that all document instances that (should) conform to the DTD must be changed. If the DTD has been widely used, i.e., if there are a lot of document instances, an update in the DTD would require a lot of effort. So as a good design principle, great take should be taken when designing a DTD.

Building a DTD may require many skills where the DTD designer should be aware of both features and limitations of the DTDs.

4.1 Document analysis

An XML based system is usually built from an existing procedural document system. A standardised DTD is chosen (and possibly modified) or a DTD is constructed from scratch. The procedural markup is replaced with XML markup. In-house styles may have been used for different (procedural) documents; the DTD designer may benefit from descriptions of these as well as from the procedural documents themselves. If there are, however, plenty of documents, using them all as background for building the DTD may not be feasible.

Procedural markup often gives the document writer artistic freedom to write different kinds of documents, where s/he, e.g, can take into account the outlook of the document on different media (by including formatting commands in the markup). This freedom may not be implementable in XML which separates the logical content from the physical layout. XML, on the other hand, ensures that applications get predictable input in a way defined in the DTD.

Still however, by simply studying examples of documents, we can set up a DTD by asking for each feature (information object, or the like)

- if it has a name,
- if it appears more than once,
- or if it must not appear more than once,
- what information should directly precede or succeed it (i.e., sequence order),
- if it can be divided into smaller units, and
- if it contains constant text that does not change in objects of the same type (this constant text could be generated automatically)

If the documents have been stored in a database, there might be a **database schema** describing their structure. This schema could be highly useful for defining the DTD. An

example notation for concept schemas is the entity-relationship diagram (ER diagram)[2]. ER diagrams contain descriptions of entities (= objects) and their relationships, such as one-to-one and one-to-many relationships that could have direct correspondences in the DTD.

The DTD designer should also take into account future uses of the DTD (and its document instances). For example, finer granularity could be used than what is necessary at the moment. In current practice, the user may not want to separate between first and last names, but uses one element that contains both names, such as

`<name>Matti Meikäläinen</name>`.

In the future, s/he would perhaps like to output the last name in a different style (font, size, etc.) than the first name; then it would be useful to have, from the start, markup that separates the two names such as

`<name><fname>Matti</fname><lname>Meikäläinen</lname></name>`.

4.2 DTD design

There are several specific questions to answer when designing a DTD: should an existing DTD be used or a new one be constructed, what names should be given to elements, should we use an element or an attribute, how should the rules be arranged in the DTD, what comments should be included, should the DTD be modularised somehow, etc. Design is also concerned with determining what elements and subelements to use, how to determine in what order and relationships they should occur, etc.

When it comes to naming options, a consistent style should be used. Element and attribute names are case-sensitive, and the designer should decide on upper or lower case (such as `someelement` or `SOMEELEMENT`). In some cases, also a mix of lower and upper case might be motivated (such as `SomeElement`), but an arbitrary mix should be avoided. The names might of course also include underscores or hyphens for readability (`some_element`).

Self-describing names are useful, when the designer and especially document markup writers have to remember what the objects (elements, attributes, entities) stand for. Names should naturally also be unambiguous. These two principles might, however, lead to very long names resulting in longer documents where the markup takes considerable space compared to the text content of the document. Shorter names, when used regularly, may be easily remembered, but are usually not self-describing. Here a reasonable balance between clarity and brevity should be found.

4.3 Recognising content, structure and presentation

Components can reflect different amounts of information about meaning versus formatting. **Content-based components** (elements) indicate what the information means in an

[2] Do not confuse ER diagram entities with XML entities. An entity in an ER diagram corresponds to a (real-world) object and the corresponding component in an XML document instance could be an element!

abstract sense but avoids saying anything about its appearance. Content-based components correspond most closely to real world objects. Examples are

- addresses, streets and postal codes
- product names, quantities, prices
- recipes, ingredients, temperatures, preparation times

Content-based components can increase the processing potential of the data and they are the ones that best agree with the XML philosophy of separating content and form.

Structural components indicate the publishing structure. Examples include

- paragraphs
- lists and items
- chapters, sections, subsections, etc.
- tables

They usually describe a more general document structure but are limited when it comes to the processing potential. A table, e.g., may contain product data or addresses or ingredients; a processor would not know the difference.

Presentational components indicate how the information should look. Examples include

- special fonts or point size
- regions to keep on a line or a page
- line breaks, page breaks.
- indented regions

Presentational components might be used when absolute control of presentation is needed, but they do go against the XML philosophy of separating content and form. They should be avoided because their place is usually in the style sheet.

The goal would be to avoid purely presentational items when a formatter can provide the formatting information automatically deduced on content-based (or structural) information. For example, page numbers should not be included in XML documents. Single-source components should be identified, i.e., components that are defined once but printed in multiple locations. Examples are marginal headers and footers which are repeated on each page within a section, then changed when a new section begins. A processor/formatter should be able to find the component and then output it where appropriate. Generally, if a component could be retrieved from a database (existing or planned), the information most likely needs a content-based component.

Another problem could be to recognise nested components where they are needed. Procedural markup is often flat, while XML (almost) assumes logical components containing subcomponents. Subcomponents may be identified, e.g., by noticing that there are several components of the same type in sequence. The question then to ask is whether there is a supercomponent containing them all. The same goes for components that appear in a group with a specific order; taken together they might represent a higher-level component.

4.4 Element or attribute

It is sometimes difficult to say, when designing a DTD, if one should use an element or an attribute to describe something. The rule of thumb says: Use an element, if the structure may contain substructures, otherwise use an attribute, especially if it concerns administrative information not shown in the output. Also, if the information is a simple choice of options, we would use an attribute instead of an element, selected from a finite set. Note also that attributes can be constrained and validated, but text in elements cannot. On the other hand, e.g., style sheets may have limited capabilities for processing attributes.

4.5 Industry standard DTDs

A DTD does not always have to be built from scratch. An industry standard DTD is a DTD that has been agreed on by several organisations. Several DTDs have been standardised or proposed, such as DTDS for publishing (BOOK DTD), for encoding mathematical formulas (MathML), etc. Many DTDs are also being converted from SGML to XML (there are, after all, small syntactical differences).

A standardised DTD is useful when sharing information. Two organisations may agree on a specific DTD for describing their data, and applications at both organisations will then rely on this structure. Standardised DTDs tend, however, to be rather complex. In order to satisfy many needs, they are usually big and flexible. In fact, they may be too flexible (allow too many different document structures) and must be adjusted to the needs of a certain organisation.

Usually it is better to choose an existing DTD (and modify it) instead of creating a new DTD. Modification may include removing redundant elements and tightening rules (e.g., requiring that an optional element is always present or absent). Tightening of context and occurrence rules does not affect the validity of the document instances. Nor does removal of optional elements. Instances conforming to the modified DTD should also conform to the original one. Removal or addition of required elements to the DTD, however, will probably result in cases where document instances do not conform to the original DTD. (But before transfer, extra elements can be removed or missing elements added where data content is empty.)

4.6 Setting up the DTD

For writing a DTD, a standard text editor may be used. The user must then take care not to make any mistakes in the DTD syntax. For more professional use, an XML editor or a

special DTD design tool give more support for building a DTD.

A DTD should include a comment of its purpose and scope: what is its primary use, who wrote the DTD, when was it created, etc. Notation declarations[3] should start the DTD and entity declarations should follow. Thereafter, the element declarations are included, usually in the order they appear in the structure hierarchy, starting with the root element. Attributes of an element are usually declared in one list, following the element declaration. Comments could be used to explain about (non- self-describing) element and attribute names.

In the following is a list of tasks to be performed when designing a new DTD.

1. Identify potential components (elements, attributes, entities).
2. Classify components — sort components into classes and subclasses.
3. Validate the needs — compare your list of potential components with work done before. Do you need all the components or are some components missing.
4. Select semantic components — select the components you will actually need from your original list.
5. Build the upper part of the document hierarchy — decide which component represent the root, what subcomponents does it have, etc. Use tree diagrams.
6. Build the information units, i.e., specify lower-level parts of the document hierarchy.
7. Build the data-level elements that have only data-content (no subelements).
8. Populate the branches — specify what text you can find in data-level elements.
9. Make connections — identify links, special features (attributes for presentational matters, processing instructions, etc.) and entities.
10. Validate the design.

Literature and references

Bradley [Bra00] contains a general chapter (Chapter 6) on document modelling. Maler and El Andaloussi [MEA96], although talking about SGML DTD design, give several good principles for building DTDs. The more detailed description of DTD modelling including the task list is due to them.

Exercises

E-4-1 We claimed that one-to-one and one-to-many relationships in ER diagrams could be represented in DTDs, e.g., by defining an element to have a content model that contains certain subelements or a content model that is a repeatable group. How would you represent a many-to-many relationship in a DTD? Take as an example

[3] Notation declarations are not described in this course

the relationship between courses and students. A student may attend many courses and you may have many students attending a certain course. Hint: Read about ID and IDREF attributes in the standard.

5 XML processing

How do we use XML documents? How do we read them? And how do we write them?

5.1 XML processors

When reading XML, an application should be able to understand the markup and the DTD. It should possibly also be able to validate the document and process specific parts of it, such as entities and attributes. Fortunately, there exist software libraries that are able to understand XML. An **XML aware processing module** reads an XML document and makes the content available to an application. It will detect problems such as errors and file formats that the application cannot process. It may replace entity references with corresponding entities and process attribute values. When validating the document, the processor must also be able to read the DTD and interpret its rules.

A **validating parser** is an example of such a processor. A validating parser reads an XML document and checks the instance against the DTD. If an error occurs in the markup (an element is missing, the document is not well-formed, or the document hierarchy is not correct, etc.), an error message should be reported.

Writing XML is easier, the output should only be enhanced with appropriate tags. But some decision must be taken on how to handle white space.

5.2 Event-driven and tree-manipulation processing

There are two main approaches to reading and processing XML documents, **event-driven processing** and **tree-manipulation**.

Event-driven processing is a simple way of processing an XML document by reading it as a stream of data. Every time an element is encountered in the stream, some sort of action is taken by the processor. The event-driven processor may be implemented easily in (procedural) programming language, such Java or C. For each element the processor would contain a procedure for processing the element. There could be separate instructions on what to do when encountering an element start tag, element content, or element end tag. As an example we could have the following instructions for transforming some XML into LaTeX (the procedure is given in pseudo language).

```
proc list()
if tag = "<list>" then
        print "\begin{itemize}\n"
        read next tag
        while tag = not "</list>" do
                if tag ="<list_item>" then
                        list_item()
                read next tag
        if tag = "</list>" then
                print "\end{itemize}"
end proc
```

```
proc list_item()
if tag = "<list_item>" then
        print "\item"
        read next tag
        while tag = not "<\item>"
                print content
                read next
        if tag = "</item>" then
                print "\n"
end proc
```

There would be no way to refer to data that occurs later in the stream (because it has not yet been read). If such reference was necessary, the event-driven processor would need to perform several passes over the data.

More advanced processing is provided through tree manipulation. Tree-manipulation gives random access to the entire document which is kept by the software in memory. Any part of the document can be accessed, interrogated and manipulated no matter what its position is in the hierarchy. Tree-manipulation is especially suitable for editors and browsers.

There are prepackaged software libraries that perform event-driven and/or tree-manipulation processing. An application communicates with these packages through an API (Application Programmers Interface). A standard has been proposed for event-driven processing, called SAX (Simple API to XML). Another standard, called DOM (Document Object Model), has been proposed for tree-manipulation processing.

Event-driven processing does not require much of the document to be held in memory, as document parts are read and immediately passed to the application. It is therefore also faster. But, for example, to build a table of contents, it is necessary to extract all section titles. An event-driven processor would need two passes over the data, while, if the document is kept in memory, titles could be found more easily by one structural query. Tree-manipulation is also more suitable, if the document parts have to be reordered. But a note of warning: some parser that use SAX might as well build the whole document tree in memory before processing it, and there will hence be no memory usage advantage compared with the tree-manipulation process.

5.3 Transformation tools

If we want to change an XML document structure, there are simpler ways to do it than to use an application that reads and modifies the document, and writes out the new structure. If the transformations are not too difficult, there are several tools available that can change and modify elements and element tags. Some can even perform more difficult transformations such as changing the order of elements. The XML Stylesheet Language Transformation (XSLT) can also be used to transform one XML document into another (although designed for formatting and presenting XML documents).

5.4 The white space problem

White space is used to describe characters that have no visual appearance but do affect the formatting of the document. Such characters are the **space** and **tab characters** as well as the **carriage return** and **line feed** characters. White space may become a problem when it is not clear if it has been introduced because it is part of the document and should be included in the output (to separate text), or if it has been introduced only for easy reading etc. in the XML document. XML processors must know a way to process this **ambiguous white space**.

An XML processor uses only the line feed character for terminating lines (as does UNIX). If the processor finds a carriage return character, it is converted into a line feed characters. If both characters are found in sequence, they are replaced by a line feed character. This is called **line end normalisation**.

Within markup, multiple white space is equivalent to a single white space character. White space can be used to separate attributes and values, etc. (But, for example, multiple white space in attribute values is not collapsed into a single white space character!). So the following markup is equivalent.

```
<country language="Finnish Swedish" capital="Helsinki" population="5.2">
```

and

```
<country
language        =       "Finnish Swedish"
capital         =       "Helsinki"
population      =       "5.2"
>
```

If an element may have only element content (so specified in a DTD), a validating parser must inform the application if the the element contains white space, in this case called **ignorable white space**. The application, for example, a publishing system may then choose to ignore this white space. However, if the element may have mixed content or data content, white space cannot be ignored.

A non-validating parser will not know which element have element content. If the `standalone` parameter in the XML declaration of a document is given the value `no`, such a parser will be warned that it will probably misinterpret some of the white space in the document.

If the attribute `xml:space` is set to `preserve`, white space will be deemed significant and included in the output (it is said to be **preserved**). If multiple white space is **normalised** into one single white space character, it is said to be **collapsed**. This attribute can be set (as FIXED) in the DTD for a certain element that will always preserve (or collapse) its white space.

```
<!ELEMENT pre (#PCDATA)>
<!ATTLIST pre xml:space #FIXED "preserve">
```

There are no issues in the XML standard relating to the white space ambiguities. Different

applications may process white space differently. In a publishing system (web browser), the following rules are recommended by Bradley [Bra00].

- block and in-line elements must be identified
- white space surrounding a block element should be removed
- a line containing nothing but declarations and/or comments should be entirely removed
- leading and trailing white space inside a block element should be removed (except when content is explicitly preserved)
- a line-end code within a block element should be converted into space
- a sequence of white space characters should be reduced to a single space character.

A block element contains text that is separated from preceding and following text, such as a paragraph. An in-line element does not generate a break in the text; an example is an emphasis element. (Compare with HTML.)

Literature and references

Again, this chapter has been mainly based on Bradley [Bra00].

Exercises

The two first exercises are due to Ahonen [AM00]

E-5-1 Try out the Apache Xerces parser (`http://xml.apache.org/`). You will find it in the directory

`/home/group/xmltools/parsers/xerces/`

Copy the file `setup_xerces` in that directory to your own test directory and run the command

`source setup_xerces`

in the directory. The command will update your CLASSPATH variable with the correct path to the parser.

Copy the files `SAXParserDemo.java`, `personal.xml` and `personal.dtd` from the directory

`/home/group/xmltools/parsers/xerces/exercise/`

to your own test directory. Compile the parser with the command

`javac SAXParserDemo.java`.

Study the output of the parser with the command

`java SAXParserDemo personal.xml`.

Make the file `personal.xml` invalid by removing some obligatory element and try parsing again. What happens?

E-5-2 Validate the file `personal.xml`.

Edit the file `SAXParserDemo.java` and remove the comment marker (`//`) in front of the line

```
//parser.setFeature("http://xml.org/sax/features/...
```

Run the parser with the file `personal.xml` again. What happens?

E-5-3 Parse/validate your XML document in exercise E-3-4 (or E-3-1, or E-3-2).

E-5-4 Parse a file that does not contain a DTD. What happens? What about validating?

E-5-5 How does Xerces process white space in a) elements with element contents, b) in elements with mixed content, c) in elements with data content? Study the output of the parser in the first exercise.

6 SAX

SAX, the Simple API for XML, is a standard interface for event-based XML processing. In this chapter we shall take a look at version 2.0, which was released in May 2000.

6.1 Introduction

Many XML parsers work in a stand-alone manner. They read an XML document, checks for wellformedness, and possibly for validity if a DTD is given. This does not yet help an application which wants to access the contents of the XML document. Many parsers can also pass on the information they read through an application programmer's interface.

Instead of each parser developer building their own API, the XML community has agreed on some standard APIs. The application is therefore not dependent on a certain parser, but parsers can be replaced as long as they follow they standard.

One example of such an API is SAX, the Simple API to XML. It is an interface standard for event-based XML processing, free for private and commercial use. It should be noted that SAX is not a parser. Different developers and organisations have built XML parsers that use the standard for passing on data to applications. Many of these parsers are free and can be downloaded from the web.

A new version of SAX, version 2.0, was recently released. It is hereafter referred to as SAX2.

6.2 Interfaces of SAX2

In order to process different kind of events that occur in event-based processing, SAX2 contains the following main interfaces (`org.xml.sax`).

Attributes: Interface for a list of XML attributes.

ContentHandler: Receive notification of the logical content of a document.

DTDHandler: Receive notification of basic DTD-related events.

EntityResolver: Basic interface for resolving entities.

ErrorHandler: Basic interface for SAX error handlers.

Locator: Interface for associating a SAX event with a document location.

XMLFilter: Interface for an XML filter.

XMLReader: Interface for reading an XML document using callbacks.

6.3 A Java implementation

We will take a closer look at a Java implementation of a parser that follows the SAX2 standard. There are several such parsers, but we will concentrate on the Apache Xerces parser (available at `http://xml.apache,org`).

This section will require some basic knowledge of Java. We will concentrate on interfaces that support processing of XML components such as elements, attributes and entities, and will spend less time on error handling and more advanced features.

The SAX2 application could contain the following instructions

```
public void performDemo(String uri) {
    System.out.println("Parsing XML File: " + uri + "\n\n");

    // Get instances of our handlers
    ContentHandler contentHandler = new MyContentHandler();
    ErrorHandler errorHandler = new MyErrorHandler();

    try {
        // Instantiate a parser
        XMLReader parser = XMLReaderFactory.createXMLReader(
                 "org.apache.xerces.parsers.SAXParser");

        // Register the content handler
        parser.setContentHandler(contentHandler);

        //parser.setFeature("http://xml.org/sax/features/validation",
            true);

        // Register the error handler
        parser.setErrorHandler(errorHandler);

        // Parse the document
        parser.parse(uri);

    } catch (IOException e) {
        System.out.println("Error reading URI: " + e.getMessage());
    } catch (SAXException e) {
        System.out.println("Error in parsing: " + e.getMessage());
    }
}
```

The procedure is given a reference to the XML document file as input. The `ContentHandler` will process the events encountered in the XML document, the `ErrorHandler` the errors that occur when running the application. The `XMLReader` creates the parser and the `ContentHandler` and `ErrorHandler` are registered for this instantiation of the parser. The different handlers can also be replaced by one single `DefaultHandler`.

For a list of `import` declarations, see the example files in the Appendix.

6.4 The XMLReader

The `XMLReader` is used for reading the XML document.

Among its methods, the most important is the `parse` method. The parser that is to be used can be set when this interface is instantiated.

```
XMLReader parser =
  XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
```

Here we use the Apache Xerces parser. The parser is called with the file name as input

```
parser.parse(uri);
```

It will read the file and return the events for the `ContentHandler`.

6.5 The ContentHandler

The `ContentHandler` is the main interface that most applications implement. The parser reports basic events, such as the start and end of elements as well as their character contents to this handler, which then can make the required modifications to the data. The handler contains, among others, the following methods.

`void characters(char ch[], int start, int length)`: For processing of character data.

`void endDocument()`: For processing the end of a document.

`void endElement(String uri, String name, String qName)`: For processing the end of an element.

`void ignorableWhitespace(char[] ch, int start, int length)`: For processing ignorable whitespace in element content.

`void processingInstruction(String target, String data)`: For processing a processing instruction.

`void skippedEntity(String name)`: For processing a skipped entity. (When the parser is non-validating.)

`void startDocument()`: For processing the beginning of a document.

`void startElement(String uri, String name, String qName, Attributes atts)`: For processing the beginning of an element.

The `StartDocument` and `EndDocument` methods contain actions to be taken on the start and end of the document. For example, we can have

```
public void startDocument ()
{
        InArticleHeadline = false;
        System.out.println("\\documentclass{report}\n\\begin{document}\n");
}
```

When the parsing begins, this method will print out two strings that start a LaTeX document including two newline characters. It will also set the variable `InArticleHeadline` to false.

The `EndDocument` method could contain the following instructions.

```
public void endDocument ()
{
        System.out.println("\n\\end{document}\n");
}
```

When the end is reach, the parser prints out the end instruction of a LaTeX document surrounded by newline characters.

The `StartElement` and `EndElement` methods contain actions to be taken on the start and end of elements. An example is

```
public void startElement (String uri, String name,
                        \item[\command{String qName, Attributes atts)
{
        if ( name.equals("HEADLINE") ) {
                InArticleHeadline = true;
                System.out.print("\\section{");
        }
        if ( name.equals("BODY") ) {
                InArticleBody = true;
        }
}
```

and

```
public void endElement (String uri, String name, String qName)
{
        if ( name.equals("HEADLINE") ) {
                InArticleHeadline = false;
                System.out.println("}");
        }
        if ( name.equals("BODY") ) InArticleBody = false;
}
```

Some different actions are taken in these methods. At the start of `HEADLINE` elements, the variable `InArticleHeadline` is set to `true`. The start of a LaTeX section is also printed. In consequence, at the end of `HEADLINE` elements, the variable is set to `false` and the LaTeX section is completed. (But in none of these methods is the actual section

title printed.) The BODY element results in no output, but a variable is set according to whether the parser ”enters” or ”leaves” such an element.

The character method can look as follows.

```
public void characters (char ch[], int start, int length)
{
        String chars = new String(ch, start, length);

        if (InArticleHeadline == true)
                System.out.print(chars);
        if (InArticleBody == true) {
                System.out.println("\\begin{quote}");
                System.out.println(chars);
                System.out.println("\\end{quote}");
        }
}
```

The method will print out only the contents of the elements BODY andHEADLINE. The BODY content will be surrounded by LaTeX quote commands.

Let’s assume that our XML document instance and DTD look as follows

```
<!--   A Sample Newspaper Article DTD  -->

<!ENTITY NEWSPAPER "Vervet Logic Times">
<!ENTITY PUBLISHER "Vervet Logic Press">
<!ENTITY COPYRIGHT "Copyright 1998 Vervet Logic Press">
<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (HEADLINE, BYLINE, LEAD, BODY, NOTES)>
<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED
                  EDITOR CDATA #IMPLIED
                  DATE CDATA #IMPLIED
                  EDITION CDATA #IMPLIED>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT BYLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE NEWSPAPER SYSTEM "newspaper.dtd">
<NEWSPAPER>
<!--This is a sample file from Vervet Logic, makers of XML Pro-->
    <ARTICLE EDITOR="Ernie Pyle" DATE="11/15/98" EDITION="Evening"
        AUTHOR="Jane Doe">
        <HEADLINE>Extensible Markup Language Proposed</HEADLINE>
        <BYLINE>Jane Doe, Staff Writer</BYLINE>
        <LEAD>The newly proposed XML Specification has been making a
         splash in the Internet development community.</LEAD>
        <BODY>The newly proposed XML draft stands to revolutionize
         the exchange of data and markup on the www, by allowing
         organizations and companies to develop their own markup
```

```
         languages quickly and easily.</BODY>
        <NOTES>No Notes</NOTES>
    </ARTICLE>
    <ARTICLE AUTHOR="John Doe" EDITOR="Ernie Pyle" DATE="02/15/98"
        EDITION="Morning">
        <HEADLINE>XML 1.0 Recommendation Released</HEADLINE>
        <BYLINE>John E. Doe, Reporter</BYLINE>
        <LEAD>The W3C today released the final recommendation for
         XML to excited developers world wide.</LEAD>
        <BODY>XML Developers, such as Vervet Logic are already
         using the released recommendation to ensure compliance
          of their products with XML 1.0. </BODY>
        <NOTES>See www.w3c.org for more information</NOTES>
    </ARTICLE>
</NEWSPAPER>
```

Running the parser will produce the output

```
\documentclass{report}
\begin{document}

\section{Extensible Markup Language Proposed}
\begin{quote}
The newly proposed XML draft stands to revolutionize
the exchange of data and markup on the www, by allowing
organizations and companies to develop their own markup
languages quickly and easily.
\end{quote}
\section{XML 1.0 Recommendation Released}
\begin{quote}
XML Developers, such as Vervet Logic are already
using the released recommendation to ensure compliance
of their products with XML 1.0.
\end{quote}
\end{document}
```

6.6 The Attributes Interface

The `Attributes` interface allows access to the attributes of an element. Some of the methods in the interface are

`int getLength()`: Return the number of attributes in the list.

`String getLocalName(int index)` : Look up an attribute's local name by index.

`String getType(int index)`: Look up an attribute's type by index.

`String getValue(int index)`: Look up an attribute's value by index.

The parser returns a list of attributes for all elements. Attributes that were declared `#IMPLIED` in the DTD and not specified in the instance are not included in the list

The following instructions in the StartElement method

```
public void startElement (String uri, String name,
                                String qName, Attributes atts)
{
        if ( name.equals("ARTICLE") )

        for (int i = 0; i < atts.getLength(); i++)
        {
                System.out.print(atts.getLocalName(i) + " " +
                        atts.getType(i) + " " +
                        atts.getValue(i) + "\n");
        }
}
```

will, when applied on the example above, produce the following list of attribute names, types and values.

```
EDITOR CDATA Ernie Pyle
DATE CDATA 11/15/98
EDITION CDATA Evening
AUTHOR CDATA Jane Doe

AUTHOR CDATA John Doe
EDITOR CDATA Ernie Pyle
DATE CDATA 02/15/98
EDITION CDATA Morning
```

Literature and references

See the Appendix for a simple SAX2 application file in Java. See also

`http://www.cs.helsinki.fi/group/xmltools/`

for more documentation. More information about SAX2 can be found on the web site

`http://www.megginson.com/SAX/index.html`

The new version 2.0 was released in May 2000. Bradley [Bra00] refers to the old version 1.0 released in May 1998. Newer versions of Bradley's book may have updated the chapter on SAX to concern version 2.0.

The Xerces parser is a product of the Apache project. The project (`http://xml.apache.org`) has among its goals to provide commercial-quality standards-based XML solutions that are developed in an open and cooperative fashion. There are several other software available for free also at the Apache web site.

Exercises

E-6-1 Build a transformation using the Xerces parser that reads XML files conforming

to the `personal.dtd` (e.g., `personal.xml`, see exercise E-5-2. The parser would change the file into a LaTeX file: the file should start with the following definitions

```
\documentclass{article}
\begin{document}
\section*{Workers at ABC}
\begin{itemize}
```

Every `person` element, e.g.,

```
<person id="one.worker">
  <name><family>Worker</family> <given>One</given></name>
  <email>one@foo.com</email>
  <link manager="Big.Boss"/>
</person>
```

should be replaced in the output with

```
\item Worker, One; e-mail: one@foo.com
```

and the LaTeX file should end with

```
\end{itemize}
\end{document}
```

(You run LaTeX on this file with the commands `latex file.tex` and `dvips file.dvi` will make the file into Postscript.)

Hint: Copy the file `SAXParserDemo.java` from the directory

`/home/group/xmltools/parsers/xerces/exercise/`

and make changes in the `ContentHandler` that will produce the required output from the input file `personal.xml`.

E-6-2 Modify the parser to transform the file `personal.xml` into HTML. You can choose the appropriate outlook yourself.

E-6-3 Build a transformation using the Xerces parser which reads XML files conforming to the `catalog.dtd` (e.g., `catalog.xml`) in the directory

`/home/group/xmltools/parsers/xerces/exercise/`

The transformation should write out the product names according to the following format:

```
Product name            Weight  Power   Street Price    Shipping
Speed Drill Pro         8lbs    120v    $129.95         $15.00
Speed Drill             7.5lbs  120v    $79.95          $10.00
...
```

If you feel energetic, you could make the program compute the total price of the products in the list, the total shipping costs and the total costs (products and shipping).

E-6-4 Study the SAX2 documentation on `http://www.cs.helsinki.fi/group/xmltools/`

Find out how the interface reacts to

- white space
- entities
- processing instructions

7 DOM

Many parsers are able to build a tree of the XML document they parse. The XML document as a tree gives an application the possibility to access any part of the document at any moment during the processing.

The **Document Object model** (DOM) has been developed by the W3C to provide a standard interface for how applications should access XML and HTML (!) documents. The **core standard** applies to both XML and HTML and some extensions to the standard describe more specific HTML processing.

The arguments for the DOM standard are mainly the same as for the SAX standard: by providing a standard interface to the applications, parser developers do not have to define their own interfaces between their parsers and possible applications.

Also as we saw in an earlier chapter, tree processing gives more power to the processing but can also result in less efficient processing when it comes to speed and memory usage. The DOM standard is more complex than the SAX standard and all DOM parsers may not provide all the features described in the standard. Some parsers use a SAX implementation to provide some of the objects needed.

In this chapter we shall take a look at the core standard and its most important features. We will not go into details needed for processing HTML specific details.

7.1 A Java implementation

As an example parser we shall again use the Java implementation of Apache's Xerces. (Note: there is both a SAX implementation and a DOM implementation of the same parser!)

To use the parser, we need the following import statements

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;
import org.xml.sax.*;
```

Suppose that we have a reference to the XML document in the variable `uri`. To invoke the parser, we write:

```
DOMParser parser = new DOMParser ();

try {
        parser.parse (uri);
        Document doc = parser.getDocument ();
}
```

The parser will read the document and represent it as a hierarchy of document objects or **nodes**. Nodes describe any kind of XML document object such as elements, text, comments, processing instructions, CDATA sections, entity references, attributes, etc. Some of the nodes may have children nodes, some are leaves that have no children.

The root to the tree is returned by the method `getDocument()`. Any part of the tree can now be accessed, e.g., through this variable. We can search for nodes, add or remove nodes, change the order of nodes, traverse the tree in any order (e.g., preorder) and print out node names or contents, etc.

All modifications that we make to a node are automatically reflected in the tree. If we want to produce output of some specific representation (say, e.g., XML), it is the task of the application to see that the tree is modified in a way that is consistent with what we expect.

The standard does not say anything about how to traverse the hierarchy. Many parsers, though, provide a **tree walker** class with methods for finding the next node in the tree in some order (e.g., finding the next node or the next element in preorder, etc.). The application may also define its own traversal methods. Tree traversal can be easily be done recursively as in

```
public void processNode (Node node) {

        // process node here if (preorder) processing required

        NodeList nodes = node.getChildNodes();
        if (nodes != null) {
                for (int i=0; i < nodes.getLength(); i++) {
                        processNode (nodes.item(i), "");
        }
        // process node here if (postorder) processing is required
}
```

The `getChildNodes()` method returns a list of possible child nodes of the current node. If the `processNode` method is called with the root of the document as parameter, it will traverse the entire document hierarchy in specified order.

7.2 Nodes

We specified that all kind of document objects are called nodes. We have the following possible types and their possible child types defined in the **Node interface**.

Node type	Possible child type
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType (maximum of one)
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	no children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	no children
Comment	no children
Text	no children
CDATASection	no children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	no children

Nodes can also be referenced in an ordered `NodeList` or and unordered list of nodes called `NamedNodeMap`.

The Node interface methods can be divided into three categories, methods processing node characteristics, navigation in the tree, and manipulation of nodes. In the following, we take a closer look at these categories.

7.2.1 Node characteristics

Node characteristics include the node's name, type and value, as well as information about its child nodes and attributes. We have the following methods for determining and modifying node characteristics.

```
String  getNodeName()
short   getNodeType()
String  getNodeValue()
void    setNodeValue(String nodeValue)

boolean hasChildNodes()
boolean hasAttributes()
```

The methods are used for finding the node name, its type and its value, for setting its value, and for finding out if the node has child nodes or attributes.

The `getNodeType()` method returns node types (represented by integers) such as

```
ELEMENT_NODE                = 1
ATTRIBUTE_NODE              = 2
TEXT_NODE                   = 3
CDATA_SECTION_NODE          = 4
```

```
ENTITY_REFERENCE_NODE           = 5
ENTITY_NODE                     = 6
PROCESSING_INSTRUCTION_NODE     = 7
COMMENT_NODE                    = 8
DOCUMENT_NODE                   = 9
DOCUMENT_TYPE_NODE              = 10
DOCUMENT_FRAGMENT_NODE          = 11
NOTATION_NODE                   = 12
```

For example, a tree traversal method can test for the node type before taking any action as in:

```
switch (node.getNodeType()) {
case Node.DOCUMENT_NODE:
            NodeList nodes = node.getChildNodes();
            if (nodes != null) {
                for (int i=0; i < nodes.getLength(); i++) {
                    processNode (nodes.item(i));
                }
            }
            break;
        case Node.ELEMENT_NODE:
            String name = node.getNodeName();
                // process element name
            NamedNodeMap attributes = node.getAttributes();
            for (int i = 0; i < attributes.getLength(); i++) {
                Node current = attributes.item(i);
                // process attributes
            }
            NodeList children = node.getChildNodes ();
            if (children != null) {
                for (int i=0; i < children.getLength(); i++) {
                    processNode (children.item(i), indent + "  ");
                }
            }
            break;
        case Node.TEXT_NODE:
        case Node.CDATA_SECTION_NODE:
                // process text and cdata nodes
            break;
        case Node.PROCESSING_INSTRUCTION_NODE:
                // process processing instruction
        ...
}
```

7.2.2 Tree navigation

The application can navigate freely in the document hierarchy. From a specific node, we can refer to the node's child nodes, sister nodes, parent node and attributes with the following methods

```
Node            getParentNode()
NodeList        getChildNodes()
Node            getFirstChild()
Node            getLastChild()
Node            getPreviousSibling();
Node            getNextSibling()
NamedNodeMap    getAttributes()
Document        getOwnerDocument()
```

The `getOwnerDocument()` method finds the root of the document that contains the node.

7.2.3 Node manipulation

The structure of the tree may also be modified. The children of a node may be removed, replaced or copied, and new children may be added. we have the following methods for performing these modifications.

```
Node    insertBefore(Node newChild, Node refChild)
Node    replaceChild(Node newChild, Node oldChild)
Node    removeChild(Node oldChild)
Node    appendChild(Node newChild)
Node    cloneNode(boolean deep)
```

7.2.4 Node lists

Some of the methods above return ordered lists of nodes. A `NodeList` has the methods

```
Node item(int index)
int getLength()
```

for accessing the items in the list and returning the length of the list.

7.2.5 Documents

The entire XML document is represented as a document node. We have methods for accessing the root of the document, its document type, and any element in the document based on the element's name.

```
DocumentType    getDoctype()
Element         getDocumentElement()
NodeList        getElementsByTagName(String tagname)
```

The Document interface also includes methods for creating new nodes such as

```
Element         createElement(String tagName)
```

```
Text            createTextNode(String data)
Comment         createComment(String data)
Attr            createAttribute(String name)
...
```

Once a node has been created it can be appended to the tree (see Node manipulation methods).

7.2.6 Elements, attributes and text

Elements, attributes and text all have separate interfaces for their processing (extending the Node interface). For general element processing we have

```
String getTagName()
NodeList getElementsByTagName(String name)
```

The `getElementsByTagName` method will return all elements with a certain name given as parameter and located in the subtree of the source element.

Attributes of an element can be processed with

```
String          getAttribute(String name)
void            setAttribute(String name, String value)
void            removeAttribute(String name)
Attr            getAttributeNode(String name)
Attr            setAttributeNode(Attr newAttr)
Attr            removeAttributeNode(Attr oldAttr)

String          getName()
boolean         getSpecified()
String          getValue()
void            setValue(String value)
Element         getOwnerElement()
```

The `getSpecified` method will return `true` if the attribute was specified in the element start tag and `false` if the attribute value was provided as a default value by the DTD.

There are also methods for normal string processing, such as

```
String          getData()
void            setData(String data)
int             getLength()

String          substringData(int offset, int count)
void            appendData(String arg)
void            insertData(int offset, String arg)
void            deleteData(int offset, int count)
void            replaceData(int offset, int count, String arg)
```

A text node may be split by the method

```
Text            splitText(int offset)
```

When a document is first made available via DOM, there is only one text node for each block of text. The user may choose to add adjacent text nodes within a given element. Such text nodes will be merged into a single node by the `normalize()` method in the Node interface.

Literature and references

For an example of using the DOM interface, see the appendix. See also

`http://www.cs.helsinki.fi/group/xmltools/`

for using DOM. More information about DOM can be found on the web site

`http://www.w3.org/TR/DOM-Level-2-Core/`

The Document Object Model (DOM) Level 2 Core Specification version 1.0 is from November 2000. (There is a Level 3 Core Draft from January 2001.) Bradley [Bra00] refers to an older version 1.0.

The Xerces parser is a product of the Apache project (`http://xml.apache.org`).

Exercises

E-7-1 How do the DOM and SAX standards differ? When would it be useful to use one instead of the other?

E-7-2 Try out the Apache Xerces parser again (`http://xml.apache.org/`), and this time the DOM interface. You will find Xerces in the directory

`/home/group/xmltools/parsers/xerces/`

Copy the file `setup_xerces` in that directory to your own test directory and run the command

```
source setup_xerces
```

in the directory. The command will update your CLASSPATH variable with the correct path to the parser.

Copy the files `DOMParserDemo.java`, `personal.xml` and `personal.dtd` from the directory

`/home/group/xmltools/parsers/xerces/exercise/`

to your own test directory. Compile the parser with the command

```
javac DOMParserDemo.java
```

Study the output of the parser with the command

```
java DOMParserDemo personal.xml
```

Make some small changes to the Java file, recompile, and study the output.

Note: The documentation for Xerces says that by setting a validation feature to true (just as in SAX), the parser would also validate the input. This seems not to be the case, validation has not been implemented in Xerces (or the implementation does not work?).

E-7-3 Build a transformation using the Xerces parser that reads XML files conforming to the `newspaper.dtd` (e.g., `newspaper.xml`). The parser collects all the article headlines and changes them into a HTML

The file starts with the following definitions

```
<HTML>
<BODY>
<UL>
```

Every `HEADLINE` element, e.g.,

```
<HEADLINE>Extensible Markup Language Proposed</HEADLINE>
```

is listed in the HTML file as

```
<LI>Extensible Markup Language Proposed
```

the list is concluded with the HTML markup

```
</UL>
```

Hint: Copy the file `DOMParserDemo.java` from the directory

`/home/group/xmltools/parsers/xerces/exercise/`

and make changes in the `printNode` method that will produce the required output from the input file `newspaper.xml`.

E-7-4 Continue the transformation above by having it constructing both a list of article headlines and the articles themselves. The headlines are transformed as described above and printed into an HTML list.

The newspaper articles follow the list, e.g., an article

```
<ARTICLE EDITOR="Ernie Pyle" DATE="11/15/98" EDITION="Evening"
    AUTHOR="Jane Doe">
    <HEADLINE>Extensible Markup Language Proposed</HEADLINE>
    <BYLINE>Jane Doe, Staff Writer</BYLINE>
    <LEAD>The newly proposed XML Specification has been making
     a splash in the Internet development community.</LEAD>
    <BODY>The newly proposed XML draft stands to revolutionize
     the exchange of data and markup on the www, by allowing
     organizations and companies to develop their own markup
     languages quickly and easily.</BODY>
    <NOTES>No Notes</NOTES>
</ARTICLE>
```

is represented in HTML as

```
<H2>Extensible Markup Language Proposed</H2>
<P>The newly proposed XML Specification has been making
a splash in the Internet development community.</P>
<P>The newly proposed XML draft stands to revolutionize
the exchange of data and markup on the www, by allowing
organizations and companies to develop their own markup
languages quickly and easily.</P>
<P>Jane Doe, Staff Writer</P>
```

Hint: You need to include a command for collecting the headlines into the list before printing out the actual articles.

E-7-5 Add working HTML links in the transformation above. When the user clicks on the headline of an article in the list and the browser ”jumps” to the article. (This exercise will require some extra reading about information not included in the lectures of the course.)

E-7-6 Build an application in DOM that reads in an XML file and prints out the same XML file in the same format.

E-7-7 Study the DOM documentation on `http://www.cs.helsinki.fi/group/xmltools/`

Find out how to process entities and processing instructions.

8 Document formatting

XML documents are not intended to be presented as such for reading. What we want is to have the content nicely presented excluding XML tags.

We have seen that it is possible to replace the tags with the SAX and DOM parsers in the previous chapters. This, however, requires that we build a suitable program that processes the tags for every DTD.

A more convenient way is to use a style sheet for defining how to present instances that conform to a certain DTD.

XSL (XML Stylesheet language) is a language for expressing such style sheets. It consists of two parts

- XSL Transformations (XSLT), a language for transforming XML documents, and
- An XML vocabulary for specifying formatting semantics (Formatting Objects).

An XSL style sheet describes how an XML document instance is transformed into an instance using the formatting vocabulary. There are already applications that are able to interpret XML documents using this vocabulary. We shall take a look at one, Apache's FOP, producing PDF files from such a representation.

8.1 XSL

In the following we look at examples of XSL but we will not make a thorough study of the subject. An example XSL specification can be found in the appendix.

An XSLT style sheet starts as follows

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

For each XML element, we can include a matching rule with some action. The following rule

```
<template match="para">
<p><apply_templates/></p>
</template>
```

matches a `para` element. It will replace `para` with `p` elements and recursively continue the processing of the document with the subelements of the `para` element (done by the `apply_templates` "command").

We may include a prefix text (or a suffix text) in the element as in

```
<xsl:template match="Order">
Order is:  <xsl:value-of select="@Name"/>
        <xsl:apply-templates select="Family"/><xsl:text>
</xsl:text>
</xsl:template>
```

This template inserts a constant text `Order is:` before continuing with the processing of the `Order` element. In this case, the value of the attribute `Name` is selected and output, and then only `Family` elements within the `Order` element are selected for further processing.

The template

```
<xsl:template match="Family">
        Family is:  <xsl:value-of select="@Name"/>
        <xsl:apply-templates select="Species | SubFamily | text()"/>
</xsl:template>
```

makes a similar selection of elements to be further processed, but includes an optional group to choose from. The option `text()` refers to text data in the element `Family`.

An example of a more complex template

```
  <xsl:template match="s2">
    <table width="100%" border="0" cellspacing="0" cellpadding="4">
      <tr>
        <td bgcolor="#006699">
          <font color="#ffffff" size="+1">
            <b><xsl:value-of select="@title"/></b>
          </font>
        </td>
      </tr>
    </table>
    <xsl:apply-templates/>
    <br/>
  </xsl:template>
```

The template matches `s2` elements. It will replace such elements with HTML `table` elements containing `tr`, `td` and other lements. The `s2` element has an attribute, `title`, whose value is selected and included in the new `table` element (in the `b` element in the `font` element in the `td` element in the `tr` element in the `table` element).

It is possible to specify in the XSL style sheet how to

- process white space
- change template ordering
- use sorting
- include automatic numbering

- use variables
- use conditions and repetitions
- etc.

As a matter of fact the power of XSL comes close to a normal programming language. Note, however, that instructions are always specified as XML elements; a style sheet will always conform to the XSL DTD.

The style sheet is ended with

```
</xsl:stylesheet>
```

When applying the above (complete) specification to an XSL Transformation, it will make the specified modifications/transformations and output the result.

8.2 Formatting Objects

The second part of the XSL standard is provided by the vocabulary for specifying **formatting objects**. Formatting objects are part of the XSL standard. Examples of formatting objects are blocks, and table-cells, and these have **formatting properties** such as border-color and font-size. In XSL, the approach to describe formatting objects have been through a formatting objects DTD (FO DTD). Each formatting object is given a concrete syntactical form. Formatting properties are described in attributes to these elements. The contents of a formatting objects is the text that is to be presented, or **rendered**, in the output according to the formatting object type and properties.

An XML document containing formatting objects is then exactly that: an XML document, conforming to the rules of the FO DTD. Of course, to include formatting details in XML goes against the principles of the standard. Normally, however, we would automatically generate the XML document containing FOs from the original XML document, e.g., by using XSLT (see previous and next sections).

Formatting objects create rectangular areas to hold the information presented. Most objects correspond to one rectangular area, but sometimes objects must be split into several areas, as in a long paragraph which is broken into several pages. The rectangular areas are placed adjacent to each other and create the **flow of the material** on the page. Areas may also contain other areas thereby forming a **formatting object tree**.

The root node of the formatting object tree must be an `root`[4] element. The children of `root` are a single `layout-master-set`, an optional `declarations`, and a sequence of one or more `page-sequence`s. The `layout-master-set` defines the page size and sequencing of the pages, while `declarations` contains objects that are used in the formatting process. The `page-sequence`s provide the content of the pages and are called **flows**. The descriptions of pages in the `layout-master-set` should naturally precede the actual pages contained in the `page-sequence`s as in the following specification.

[4] Note that all names of formatting object elements are preceded by `fo:`. We leave out this prefix to make the text more readable, but use the prefix in the examples.

```
<fo:root>
  <fo:layout-master-set>...</fo:layout-master-set>
  <fo:page-sequence>...</fo:page-sequence>
  <fo:page-sequence>...</fo:page-sequence>
</fo:root>
```

Page templates are defined in a `page-master` within the `layout-master-set` element. The `page-master` contains at least one `simple-page-master` element which defines a single page such as an initial page, a left-side page of a right-side page. Each `page-master` also has a name. We have then

```
<fo:root>
  <fo:layout-master-set>
    <fo:page-master>
      <fo:simple-page-master
        page-master-name="FirstMaster">
      ...
      </fo:simple-page-master>
    </fo:page-master>
  </fo:layout-master-set>
  <!-- page sequences -->
  <fo:page-sequence>...</fo:page-sequence>
  <fo:page-sequence>...</fo:page-sequence>
</fo:root>
```

Each `page-master` can have a writing direction such as left-to-right, top-to-bottom for languages such as English and Finnish (specified as `lr-tb`) but other languages may use other writing directions. A `page-master` also can also have a height and width attributes, margin attributes, and a reference orientation attribute if we want to change the default writing direction. The reference orientation attribute is specified in degrees from 0 to 360 degrees. A possible `page-master` is then

```
<fo:simple-page-master
  page-master-name="FirstMaster"
  margin = "1cm"
  size ="25cm 15cm"
  writing-mode = "lr-tb"
  reference-orientation = "0">
</fo:simple-page-master>
```

We can also use `margin-right`, `margin-left`, `margin-bottom` and `margin-top` for separately specifying the margins. Active regions on the page are defined with `region-...` elements. The only required such element is `region-body` which specifies the text body, but we can also specify the header, footer, left margin and right margin of the page as in

```
<fo:simple-page-master ...>
  <region-body margin-top="3cm" .../> <!-- main flow -->
  <region-before .../> <!-- header -->
  <region-after  .../> <!-- footer-->
  <region-start  .../> <!-- left margin -->
  <region-end    .../> <!-- right margin -->
</fo:simple-page-master>
```

All these regions can be given their own reference orientation and writing mode, as well as borders and padding. For example, text in a margin could be vertical. The `region-body` element also has attributes that define the number of columns, the width between columns, the background colour, and the width of any border around the area.

A `page-sequence` must specify the name of its `page-master`. A `page-sequence` also contains a `flow` element that contains the content of the pages. The `flow` element specifies where in the page the text should go, e.g., in the body or in the margin. Below, we have specified the text to appear in the page body.

```
</fo:root>
  <fo:page-sequence master-name="FirstMaster">
    <fo:flow flow-name="xsl-region-body">

      <!-- text content -->
      <fo:block>Roses are red,</fo:block>
      <fo:block>Violets are blue</fo:block>
      <fo:block>Violets are blue;</fo:block>
      <fo:block>Sugar is sweet,</fo:block>
      <fo:block>And I love you.</fo:block>
    </fo:flow>
  </fo:page-sequence>
```

The actual text is here specified in **blocks** which are used to enclose simple blocks of text, such as paragraphs or titles, and in this cases verses in a poem. At the beginning of the XML document, we should have an XML declaration and then specify what namespace the elements use (i.e. `fo:` in the element names refers to the url given below and makes the names unique.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
```

If we put these parts together, we have a simple XML document containing the above specifications and a short poem.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <fo:layout-master-set>
    <fo:simple-page-master
        margin-right="2.5cm"
        margin-left="2.5cm"
        margin-bottom="2cm"
        margin-top="1cm"
        page-width="29.7cm"
        page-height="21cm"
        master-name="FirstMaster">
        <fo:region-body margin-top="3cm"/>
    </fo:simple-page-master>
  </fo:layout-master-set>
```

```
  <fo:page-sequence master-name="FirstMaster">
    <fo:flow flow-name="xsl-region-body">

      <fo:block>Roses are red,</fo:block>
      <fo:block>Violets are blue</fo:block>
      <fo:block>Sugar is sweet,</fo:block>
      <fo:block>And I love you.</fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

The Apache project provides a Formatting Objects Processor (FOP) for translating such files into PDF format (We will learn how to use FOP in the exercises of this chapter.) In this case the result would be (this does not look exactly as the PDF result when printed, as LaTeX is not able to reproduce PDF.)

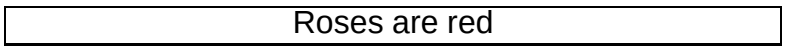
Roses are red,
Violets are blue
Sugar is sweet,
And I love you.

Unfortunately, not all formatting objects or properties in the standard have been implemented in FOP. For example, the `writing-mode` attribute has not been implemented, so we cannot see what happens if we decide to specify, e.g, the writing mode to right-to-left, bottom-up.

Let us instead take a look at some more formatting objects and properties. The `block` element specifies a block of text. It has several attributes. For example, we can specify padding, alignment, text colour, background colour, space after the block, line height, font family and font size as in the following block.

```
<fo:block
  padding-top="3pt"
  text-align="center"
  color="white"
  background-color="blue"
  space-after.optimum="15pt"
  line-height="24pt"
  font-family="sans-serif"
  font-size="18pt">Roses are Red</fo:block>
```

When included in the above poem as a title, we would produce the following result (with FOP into PDF).



Roses are red,
Violets are blue
Sugar is sweet,
And I love you.

The title would be printed in white on a blue background.

If the block contains more text than fits on a line, the text will be divided into several lines. We can specify that the text should be justified (no ragged margins!). The specification

```
<fo:block text-align="justify">Formatting objects are part of
the XSL standard. Examples of formatting objects are blocks, and
table-cells, and formatting properties such as border-color and
font-size. In XSL, to approach to describe formatting objects
have been ...
</fo:block>
```

would give a nice block of text as in

> Formatting objects are part of the XSL standard. Examples of formatting objects are blocks, and table-cells, and formatting properties such as border-color and font-size. In XSL, to approach to describe formatting objects have been through a formatting objects DTD (FO DTD). Each formatting object is given a concrete syntactical form. Formatting properties are described in attributes to these elements. The text of a formatting objects is the text that is to be presented, or rendered, in the output according to the formatting object type and properties.

where the width of the text is set in the `page-master`.

There are several **inline objects**. An inline objects is not separated in a block from the surrounding text. Typical inline objects would be emphasised words, etc. Single characters can have their properties set (e.g., font, size, color, etc) in a `character` element. The specification

```
H<fo:character
  character="2"
  color="green"
  vertical-align="sub"/>O
```

would make a green subindex ”2” between an H and an O letter. (Unfortunately FOP, again, does not implement the subindex property.)

Other inline objects can be specified in `inline` elements such as font weight and style (bold, italics, normal, etc.), and font size. Page numbers can be referred to in the text. A specification as below

```
<fo:block text-align="justify">
Here is <fo:inline font-weight="bold">some
text in bold font</fo:inline>.
</fo:block>

<fo:block text-align="justify">
<fo:inline font-style="italic">The current
page number is <fo:page-number font-size="200%"/></fo:inline>.
</fo:block>
```

will produce a page where we have the text

Here is **some text in bold font**.
The current page number is 1

Lists and tables have their own formatting objects. The following specification

```
<fo:list-block>
  <fo:list-item>
    <fo:list-item-label>
      <fo:block>a.
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>List item 1.
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>

  <fo:list-item>
    <fo:list-item-label>
      <fo:block>b.
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>List item 2.
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</fo:list-block>
```

will produce a simple list like

a. List item 1.
b. List item 2.

A table specification

```
<fo:table width="12cm" >
  <fo:table-column column-number="1" column-width="72pt">
  </fo:table-column>
  <fo:table-column column-number="2" column-width="72pt">
  </fo:table-column>
  <fo:table-column column-number="3" column-width="72pt">
  </fo:table-column>

  <fo:table-body>
    <fo:table-row>
      <fo:table-cell column-number="1">
        <fo:block>Cell 1
        </fo:block>
      </fo:table-cell>
      <fo:table-cell column-number="2">
        <fo:block>Cell 2
        </fo:block>
      </fo:table-cell>
      <fo:table-cell column-number="3">
        <fo:block>Cell 3
        </fo:block>
      </fo:table-cell>
    </fo:table-row>
    <fo:table-row>
      <fo:table-cell column-number="1">
        <fo:block>Cell 4
        </fo:block>
      </fo:table-cell>
      <fo:table-cell column-number="2">
        <fo:block>Cell 5
        </fo:block>
      </fo:table-cell>
      <fo:table-cell column-number="3">
        <fo:block>Cell 6
        </fo:block>
      </fo:table-cell>
    </fo:table-row>
  </fo:table-body>
</fo:table>
```

will produce a simple table

Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

There are many other formatting objects and properties in the standard. Applications, will at this stage, probably only implement a few of these objects.

8.3 Combining XSLT and Formatting Objects

Replacing XML document elements with appropriate formatting objects and properties is a rather tedious task to do manually. Instead, we can define a suitable XSL transformation performing the required replacements. We start with some declarations and a reference to XSL and FO.

```
<?xml version="1.0"?>
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="1.0"
        xmlns:fo="http://www.w3.org/1999/XSL/Format">
```

For element in the original XML document, there is an XSLT template specifying how the element should be replaced with the corresponding format object. Consider, e.g., a short document containing a root `doc`, some `para` elements and `title` elements (sections and subsections) as well as a `header` element. In the following, we specify an XSL stylesheet for transforming the document.

The root of the document is matched and replaced with information in the following template

```
<xsl:template match="doc">
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="single"
                  page-height="21cm"
                  page-width="29.7cm"
                  margin-top="1cm"
                  margin-bottom="2cm"
                  margin-left="2.5cm"
                  margin-right="2.5cm">
      <fo:region-body margin-top="3cm"/>
      <fo:region-before extent="3cm"/>
      <fo:region-after extent="1.5cm"/>
    </fo:simple-page-master>
  </fo:layout-master-set>
  <fo:page-sequence master-name="single">
    <fo:static-content flow-name="xsl-region-before">
                <xsl:apply-templates select="header"/>
    </fo:static-content>
    <fo:flow flow-name="xsl-region-body">
                <xsl:apply-templates select="body"/>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Note that the element names starts with `fo`. This template adds some information about the page, its height and with, margin sizes, etc. (in the `layout-master-set`). It also specifies some constant text that should appear on everey page in the `static-content` element.

A `para` element is matched by the following template.

```
<xsl:template match="para">
        <fo:block font-size="12pt"
                font-family="sans-serif"
                line-height="15pt"
                space-after.optimum="3pt"
                text-align="start">
                <xsl:apply-templates/>
        </fo:block>
</xsl:template>
```

The template will replace the `para` element with a `block` element, where a certain font size, family, etc. are used.

`title` element with `level` attribute value of `1` is also made into a block element as follows.

```
<xsl:template match="title[@level='1']">
        <fo:block font-size="18pt"
                font-family="sans-serif"
                line-height="24pt"
                space-after.optimum="15pt"
                background-color="blue"
                color="white"
                text-align="center"
                padding-top="3pt">
                <xsl:apply-templates/>
        </fo:block>
</xsl:template>
```

A title of level 2 could be made into a similar block element.

A header is also changed into a block element

```
<xsl:template match="header">
        <fo:block font-size="10pt">
        <xsl:apply-templates/>
        </fo:block>
</xsl:template>
```

Note that in all cases, we need the call for recursive processing (`apply_templates`) as these elements may include other elements (and text). When text content is found in a element it copied to the formatting objects file.

A complete XSL stylesheet for changing an XML document into one with formatting objects can be found in the appendix.

Literature and references

The XSL standard is described at

`http://www.w3.org/Style/XSL/` and

`http://www.xslinfo.com/`.

There is more information about using Xalan and FOP at

`http://www.cs.helsinki.fi/group/xmltools/`

and documentation also at

`http://xml.apache.org/`

Especially, note in the documentation on FOP what has been implmented of the XSL standard and what has not.

Exercises

E-8-1 Try out the Apache Xalan formatter (`http://xml.apache.org/`). You find Xalan in the directory

`/home/group/xmltools/formatters/xalan/`

Copy the file `setup_fop` from the directory

`/home/group/xmltools/formatters/exercise/`

and run the command

```
source setup_fop
```

in the directory. The command will update your CLASSPATH variable with the correct path to the software libraries.

Copy the files `SimpleTransform.java`, `birds.xml` and `birds.xsl` from the directory

`/home/group/xmltools/formatters/exercise/`

to your own test directory. Compile the file with the command

```
javac SimpleTransform.java
```

Study the output in `birds.out` of Xalan with the command

```
java SimpleTransform
```

Note: The program assumes the input to be in files `birds.xml` and `birds.xsl` (no command line input).

Make small changes in the Java file, recompile and study the result.

E-8-2 Try out the Apache FOP formatter (`http://xml.apache.org/`). FOP translates XML documents in the FO DTD format into PDF files.

You find FOP in the directory

`/home/group/xmltools/formatters/fop/`

Copy the file `setup_fop` from the directory

`/home/group/xmltools/formatters/exercise/`

and run the command

```
source setup_fop
```

in the directory. The command will update your CLASSPATH variable with the correct path to the software libraries.

Copy the files `SimpleTransform2.java`, `normal.xml` and `normal.xsl` from the directory

`/home/group/xmltools/formatters/exercise/`

to your own test directory. Compile the file with the command

```
javac SimpleTransform2.java
```

Study the output in `normal.fo` of Xalan with the command

```
java SimpleTransform2
```

Note: The program assumes the input to be in files `normal.xml` and `normal.xsl` (no command line input).

To use FOP as in

```
java org.apache.fop.apps.CommandLine normal.fo normal.pdf
```

Instead of these two steps, you can combine them as in

`java org.apache.fop.apps.XalanCommandLine normal.xml normal.xsl normal.pdf`.

Make small changes in the XSL file, run again and study the result.

Make small changes in the FO file, run again and study the result.

E-8-3 Change the elements in the file `personal.xml` to formatting objects and use FOP to turn it into PDF. You will have to look at the documentation of FOP at

`http://www.cs.helsinki.fi/group/xmltools/`

E-8-4 Change the elements in the file `catalog.xml` to formatting objects and use FOP to turn it into PDF. You will have to look at the documentation of FOP at

`http://www.cs.helsinki.fi/group/xmltools/`

E-8-5 Write an XSL file for changing the file `personal.xml` into PDF. The XSL file will add formatting objects and the result can be applied to FOP. You will have to look at the documentation of XSL and FOP at

`http://www.cs.helsinki.fi/group/xmltools/`

9 Other related standards

In the previous chapters we have taken a look at the following standards

- The **Extensible Markup Language** (**XML**) Version 1.0 (Second edition, W3C recommendation from October 2000)
- The **Simple API for XML** (**SAX**) Version 2.0 (de facto standard from May 2000)
- The **Document Object Model** (**DOM**) Level 2 Core Specification Version 1.0 (W3C recommendation from November 2000)
- The **Extensible Stylesheet Language** (**XSL**) Version 1.0 (W3C candidate recommendation from November 2000) including
  - the **Formatting Objects** Language for specifying formatting semantics (**FO DTD**), and
  - the **XSL Transformations** (**XSLT**) Version 1.0 (W3C recommendation from October 2000).

The development of these standards continues and new updated versions are released frequently.

The XML standard has several other related standards that we have not considered in much detail in this course. The "parent" of XML is the **Standard Generalized Markup Language** (**SGML**) standard, a standard that is more complex than XML and which was released in 1986. The **Hypertext Markup Language** (**HTML**) was created as an SGML application, i.e., a certain HTML DTD was used to define HTML; however, the SGML standard was not strictly followed, there has been several versions of HTML and especially vendors have made their own specific extensions to HTML.

XML was created as a hybrid between SGML and HTML, to combine the best features from the two markup languages. One of the design goals has been to keep the XML standard rather simple. This also means that there has been a need for adding extensions to the standard: descriptions of how to process XML (SAX and DOM), how to specify formatting information (XSL), etc.

In this chapter, we will take a look at some further extensions, namely,

- **XML Namespaces**, which provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references (W3C recommendation from January 1999)
- **XML Linking**. There are several standards concerned with linking in XML: The **XML Path Language** (**XPath**) and the **XML Pointer Language** (**XPointer**) for locating any fragment of a document, and the **XML Link Language** (**XLink**) for including hypertext links.
- The **Extensible HyperText Markup Language** (**XHTML**) Version 1.0: A Reformulation of HTML 4 in XML 1.0 (W3C Recommendation from January 2000)

- **XML Schema Part 1: Structures** and **Part 2: Datatypes** (W3C Candidate Recommendation from October 2000). This standard provides means for specifying more constraints on document instances than what DTDs can do. For example, datatypes, complex data structures, etc.

9.1 Namespaces

A single XML document may have fragments that are defined in different DTDs. For example, sometimes it might be useful to include HTML fragments such as tables which are powerful and well supported by web browsers. Another example of documents that mix several DTDs is the XSLT standard that mixes both formatting instructions and target document tags in the same document.

There are two problems with mixing tags from different DTDs. First, we should be able to identify to which DTD a particular element belongs. XML processors need perhaps to treat the elements differently depending on this information. Also, we must avoid name collisions: two different DTDs may define elements or attributes with the same names. Then, of course, there is also a third problem with validating documents that contain elements defined in different DTDs.

A **namespace** is an environment where all elements are unique. A single DTD is considered to own such a namespace. All references to either element or attribute names in this namespace are unique. A document that contain references to several DTDs is said to have **multiple namespaces**.

In order to make names unique, a prefix is added to names such as

```
prefix:name
```

The prefix and the name are separated by a colon. This object is called a **qualified name**. The prefix could in principle be any string that makes the qualified name unique. Most standards can be identified through their location on the web, they all have a unique uniform resource locator (URL). We can then use the URL referring to a certain standard as the prefix. This is helpful in two aspects, the URL is usually know to anyone interested in the certain DTD or it also helps to locate more information about the DTD of the URL was not known the person who wants to use the DTD.

However, URLs are usually very long. Repeating them in every element and attribute name would be tedious and make the document very long. They also contain characters that are not allowed in element or attribute names. Fortunately, there is a mechanism for defining short legal prefixes based on the URLs.

It should be noted that applications that understand namespaces do not have to be connected to the Internet in any way! The URL is useful purely to distinguish between names. The application will compare the URL against a list of standards that it can process, but does not have to go and look up the standard at the specified web address.

Namespaces are defined using attributes. The attribute `xmlns` is used for this as in

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

The `xmlns` attribute is used to define the namespace at the same time as the prefix to be used. The prefix is separated with a colon from the string `xmlns` in the definition. Namespaces are typically defined in the root of the document, but they can be defined in any element. Note that several namespaces can be defined in the same element.

The prefix is then used, for example as in the following XSL specification.

```
<xsl:template match="doc">
  <fo:root>
    <fo:layout-master-set>
      <fo:simple-page-master fo:master-name="single">
...
```

Also attributes from one namespace can be defined in elements from another, as long as we specify the prefix for the other namespace.

We can also define a default namespace; then we do not have to specify the prefix in element names. The default namespace can be changed at any point in the document. For example, we define the default namespace to be **mybook.dtd** as in

```
<book xmlns="mybook.dtd"
        xmlns:other="http://www.w3.org/TR/REC-html40">
        <p>This element comes from the mybook.dtd</p>
        <other:p>And this element from the other</other:p>
```

Attributes automatically default to the namespace of the element they are specified in. In the XSL example above, we could have had

```
      <fo:simple-page-master master-name="single">
```

instead of the attribute name `fo:master-name`.

Now to the problem with parsing. If a document contains multiple namespaces, its DTD must reflect this as well. We will then have to define a new DTD based on the ”original” DTDs, if we want to be able to validate the document. We could have a rule such as

```
<!ELEMENT book (p | other:p)>
```

to allow the example above to be validated. In the future, it is expected that XML parsers will be able to validate documents against the original DTDs so that we do not have to write a new DTD containing the prefixes. (On the other hand, how will we specify where an element from namespace can appear in an element from another namespace?)

9.2 Linking in XML

For locating documents and document fragments we use uniform resource locates (URLs) and the XML Path standard.

9.2.1 URLs

The **uniform resource locator** (**URL**) is used to locate documents on the Internet. It can also be used to identify documents on the intranet or on a local machine. URLs are specified as

```
protocol://host/location/resource
```

When a file is stored on a local storage, there is no protocol involved and instead we use the name `file:` such as

```
file:///home/fs/linden/test.xml
```

or

```
file:/c:/usr/linden/test.xml
```

Remote sources usually use the **HTTP** protocol or the **FTP** protocol. We can have

```
http://www.cs.helsinki.fi/u/linden/opetus/xml/index.html
```

URLs can be relative as in

```
../exercises/exercise1.html
```

and we can also include queries and fragments as in

```
http://www.cs.helsinki.fi/cgi/get-personnel?find=Address&name=Pekka"
```

and

```
files/document#part6
```

These URLs can be used, e.g, in entities for locating files used in the document.

9.2.2 XPath

The **XML Path standard** (**XPath**) is used for exploiting the inside of XML documents. Consider the following document.

```
<book>
  <title>A new book</title>
  <body>
    <chapter>
        <title>Introduction</title>
    </chapter>
    <appendix>
        <title>Appendix title</appendix>
    </appendix>
...
```

The book title would probably be processed rather differently from the chapter titles. How does a processor know to distinguish between the elements?

The XPath standard uses expressions to refer to document elements. The simple expression

```
book/title
```

identifies `title` elements are are direct children of `book` elements. Expressions identify elements by their location or 'path' in the document tree. Such a path can be relative or absolute.

A **relative path** starts from a currently selected element (e.g, by parser). For example

```
chapter
```

refers to chapters that are children of the current element. This expression is actually an abbreviation for

```
child::chapter
```

Note that this really means all children of the current element that are labeled `chapter`, not children of chapter elements! A path can include multiple steps separated by a slash as in

```
chapter/title
```

or

```
child::chapter/child::title
```

Wild cards may be used to cover more possibilities. We can use a wild card to replace an element as in

```
book/*/title
```

This expression would replace the expressions

```
book/chapter/title
book/appendix/title
```

but if the `title` element was located on different levels in the documetn tree (and not always two levels below the `book` element), this approach would not find all titles.

An expressions such as `node()` would refer to all nodes, including elements, in a document tree. Combined with the expression `descendant-or-self::` as in

```
book/descendant-or-self::node()/title
```

we would find all `title` elements located under the `book` element.

We have other expressions, such as `self::` referring to the current node itself and `parent::` referring to the parent of the current node. An expression `text()` would refer to the data content of a node, just as `processing-instruction()` and `comment()` represent any of these types.

An **absolute path** does not rely on the current context. We refer to the root of the tree with a slash `/`. The following expression

```
/book/title
```

would refer to `title` elements as children of `book` elements as children directly of the root. We can also use a double slash to refer to all elements of a certain name. The expression

```
//title
```

would refer to all `title` elements whatever the context.

We can refer to previous or next siblings with the expressions `previous-sibling::` and `next-sibling::`, and to ancestors (up to the root) or descendants (all nodes in the subtree rooted at the current element) with the expressions `ancestor::` and `descendant::`. We can also refer to a certain position within an element by `position()` as in

```
chapter[position()=2]
```

selecting the second `chapter` element. The above expression is equivalent to the expression

```
chapter[2]
```

The expression `last()` selects the last element in a list, the expression `count()` computes the number of nodes in a list.

XPath also allows for testing attribute values, using boolean tests, and for string testing and some simple arithmetics making a quite powerful way to locate document fragments anywhere in a document tree.

9.2.3 XML links

The XML standard provides a very simple linking mechanism for linking in a single document. Two special attribute types, `ID` and `IDREF` are used for this purpose. A simple example defines a chapter and a possible reference as follows.

```
<!ELEMENT chapter (...)>
<!ATTLIST chapter target ID #REQUIRED>
...
<!ELEMENT Xref (...)>
<!ATTLIST Xref link IDREF #REQUIRED>
```

We use the link as follows

```
In Chapter <Xref link="other_standards"/> we will discuss ...
...
<chapter target="other_standards">
```

The `ID` name must be unique within a document, all chapters must have a `target` attribute and if we use the `Xref` element, we must specify the `link` attribute.

The **XML Linking Language** (**XLink**) standard extends this linking mechanism by allowing for references to other documents, for using typed linkes, for locating multiple targets and for pointing to and from read-only documents.

9.2.4 XPointer

The **XML Pointer Language** (**XPointer**) completes the XLink language by allowing to point to document fragments that do not have a unique identifier, but that still have a significant location in the document. The XPointer language relies heavily on the XPath standard and uses expressions to refer to document fragments. For example, we have the following XPointer expression

```
http://www.cs.helsinki.fi/books/mybook.xml#xptr(chapter/title[3])
```

referring to the third chapter within a book located at the specified URL address; XPointer uses the expression `xptr()` and the argument is an XPath expression.

9.3 XHTML

HTML is an application of SGML. However, since XML has become so popular, the desire to combine HTML and XML somehow has arisen. For example, it would sometimes be convenient to be able to embed HTML elements in XML and vice versa. There is belief that the next version of HTML will be an XML application and there is now a recommendation for the **XML Hypertext Markup Language** (**XHTML**) to fill this function.

Some additional constraints have been set on XHTML so that XML specifications are followed. For example, as XML is case-sensitive, the XHTML DTD has been defined using only lower-case names and only such names are allowed in XHTML documents. Empty elements, such as the `img` element, must end with `/>`. If an element has not been declared empty, both start and end tags must be used (the end tag may not be left out as often is done in HTML, as browsers are still mostly able to interpret the HTML in the correct way).

9.4 XML Schema

A schema is a model for describing the structure of information. The term is borrowed from the database world where it is correspondingly used to describe the structure of

information in relational tables. A schema describes more about information than a DTD. While DTDs contain content models describing the order and sequence of elements, schemas also specify datatypes of elements.

XML schemas are XML documents. The vocabulary of XML schemas contain about thirty elements and attributes for describing information structure. An element type could be described as

```
<elementType name="person_name">
  <sequence>
    <elementTypeRef name="fname" minOccur="1" maxOccur="3"/>
    <elementTypeRef name="lname" minOccur="1" maxOccur="1"/>
  </sequence>
</elementType>
```

This description says that `person_name` elements must contain one to three `fname` subelements, and exactly one `lname` subelement, in this order.

An `fname` element can be declared to contained only text as in

```
<elementType name="fname">
  <mixed/>
</elementType>
```

XML Schema also allows definitions of datatypes. Here is a schema describing a datatype.

```
<datatype name="currency">
  <basetype name="decimal"/>
  <precision>8</precision>
  <scale>2</scale>
</datatype>
```

Currencies in this type are given as decimal numbers, they are allowed to contain eight digits including two digits after the decimal point. An element using this type is described as

```
<elementType name="unitcost">
  <datatypeRef name="currency"/>
</elementType>
```

To check for ”validity”, we need an XML processor that understands schemas. Schemas could be especially useful when exchanging data between databases or when used in e-commerce.

9.5 Other standards

XML Query is a standard describing a query langauge for XML. XML Signature describes how to make electronic singnatures in XML. XML Protocol is used for data communication. We refer to the web page of the XML standard at `http://www.w3c.org/XML/` for further information about related standards.

Literature and references

Please see `http://www.w3c.org/XML/` for further information about related standards. Henri Ruini has made a list with explanations in Finnish on XML related standards at

`http://www.cs.Helsinki.FI/u/ruini/structure/xml/intro/maarittelyt.html`

Exercises

E-9-1 Take a look at `http://www.w3c.org/XML/`. What related standards can you find? Also look at `http://www.xmlinfo.com` and `http://www.cs.Helsinki.FI/u/ruini/structure/xml/intro/maarittelyt.html`.

10 Synthesis

In this last chapter, we are going to make a synthesis of what we have learned in this course. Figure 2 shows a schematic overview of **one possible way** producing and processing XML. The figure includes also mentions of the tools we have used in the course.

Our purpose for XML processing is to standardise the production of good quality output from, in principle, any kind of input data. As a means, we use XML as a canonical representation to which we change our input and from which we render any output.

According to Figure 2 we start with input data in any format, such as plain text, HTML pages, Word or WP documents, etc. The input can be available in files or extracted from databases. In the first phase, we add markup to our data. We (probably) also have to construct a XML DTD and define a style sheet (or several) for the data. In this course, we have tested only one XML editor, but also used (non-XML) ordinary editors, such as emacs, for adding markup manually. For automatically adding markup to existing documents, there are several conversion programs (not mentioned in this course). In principle, we can use any (procedural or other) markup available in the input data for making this phase easier. A good help in DTD design may also be possible in-house document styles or database schemas, etc. A useful help would also be to use a DTD designer tool. The result of the markup phase is XML data, i.e., XML DTDs, document instances and XSLT style sheets.

The XML processing phase will change our XML data into any specified output format. XML processing includes parsing, transforming and formatting the data. An XML parser may check for the well-formedness and/or validity of the XML data. If the XML parser is combined with either the SAX or DOM interfaces, processing of the XML data can be event-driven or tree-manipulated. An XSLT processor will read, in addition to the XML document instances and XML DTDs, also an XSLT style sheet. It may produce any kind of output (as the XSLT language is very powerful), but in the normal case it produce an XML document containing formatting information (formatting objects and properties). This information will be read by an XSL processor which produces formatted output. The result of this phase could be data in any format, e.g., we could print out data as plain text or have an XSL processor produce PDF. In this course, we have used the Apahce Xerces XML parser, together with the SAX and DOM interfaces for making XML transformations. We have also used the Apache Xalan XSLT processor (which uses Xerces for parsing) for processing XSLT style sheets and producing either XSL documents or some other kind of output. We have also used the Apache FOP (Formatting Objects Processor) for producing PDF out of an XML document containing formatting objects or from an XML document and a corresponding XSLT stylesheet.

Note that **this is not the only way to process XML data**, but only an example of a possible dataflow that could be used to produce formatted output.

Processing data and phase	**Any format**, "input 1"	**Markup, design**	**XML data**, "input 2"	**XML processing** *Parsing*	*Transforming*	*Formatting*	**Any format**, "output"
Data flow	documents, files, databases, web pages, ...	$\Longrightarrow$	XML DTDs XML instances XSLT style sheets	$\Rightarrow$ *well-formed?* $\Rightarrow$ *valid?* $\Rightarrow$ $\Rightarrow$ $\Rightarrow$ $\Rightarrow$	*SAX event-driven* *DOM tree manipulation* *XSLT processor* *XSLT processor* $\rightarrow$ XSL $\rightarrow$ *XSL processor*	$\Rightarrow$ $\Rightarrow$ $\Rightarrow$ $\Rightarrow$ $\Rightarrow$ $\Rightarrow$	(formatted) documents, files, databases, web pages, ...
Formats	text, LaTeX, HTML, paper, ...		XML XSLT	XSL			text, HTML, XML, PDF, ...
Tools used in this course		*XMLPro editors*		*Xerces* *(Xerces+)*	*+SAX* *+DOM* *Xalan*	 *FOP*	

Figure 2: Schematic overview of XML processing.

References

[AM00] Helena Ahonen-Myka. Electronic commerce and internet - lecture course. `http://www.cs.helsinki.fi/u/summanen/opetus/2000/VEKA/index.en.html`, 2000.

[Bra00] Neil Bradley. *The XML Companion*. Addison-Wesley, 2nd edition, 2000.

[GP00] Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice Hall, 2000.

[MEA96] Eve Maler and Jeanne El Andaloussi. *Developing SGML DTDs. From Text to Model to Markup.* Prentice-hall, 1996.

[Tec00] Morphon Technologies. Morphon XML editor. http://www.morphon.com/, 2000.

A Short trilingual dictionary

English	Finnish	Swedish
attribute	attribuutti tai määrite	attribut
content	sisältö	innehåll
declaration	määrittely	deklaration
definition	määrittely?	definition
delimiter	erotinmerkki	avgränsningstecken
document	dokumentti (myös asiakirja)	dokument
document instance	dokumentin ilmentymä	dokumentinstans
document type definition	dokumenttityypin (rakenne)määrittely	dokumenttypsdefinition
DTD	DTD	DTD
element	elementti	element
end tag	lopputunniste	sluttagg
entity	entiteetti	entity
markup	merkkaus	uppmärkning
parser	jäsentäjä, jäsennin	parser, strukturerare
start tag	alkutunniste	starttagg
structure	rakenne	struktur
stylesheet	tyylimäärittely	stildefinition (?)
tag	tunniste	tagg, märkord
valid	validi	valid
well-formed	hyvinmuodostettu	välformad

Literature and references

Möller, Anders: *SGML — en introduktion till Standard Generalized Markup Language*. Studentlitteratur, 1994.

Ruini, Henri. *Johdatus XML-tekniikkaan*.
`http://www.cs.Helsinki.FI/u/ruini/structure/xml/`

B ISO 639:1988 language codes

Note! There is a newer version of this standard, ISO 639-2, which uses three letter abbreviations for languages. (Will this standard be used in XML?)

aa	Afar
ab	Abkhazian
af	Afrikaans
am	Amharic
ar	Arabic
as	Assamese
ay	Aymara
az	Azerbaijani
ba	Bashkir
be	Byelorussian
bg	Bulgarian
bh	Bihari
bi	Bislama
bn	Bengali; Bangla
bo	Tibetan
br	Breton
ca	Catalan
co	Corsican
cs	Czech
cy	Welsh
da	Danish
de	German
dz	Bhutani
el	Greek
en	English
eo	Esperanto
es	Spanish
et	Estonian
eu	Basque
fa	Persian
fi	Finnish
fj	Fiji
fo	Faroese
fr	French
fy	Frisian
ga	Irish (Irish Gaelic)
gd	Scots Gaelic (Scottish Gaelic)
gl	Galician
gn	Guarani
gu	Gujarati
gv	Manx Gaelic
ha	Hausa
he	Hebrew (formerly iw)
hi	Hindi
hr	Croatian
hu	Hungarian
hy	Armenian
ia	Interlingua
id	Indonesian (formerly in)
ie	Interlingue
ik	Inupiak
is	Icelandic
it	Italian
iu	Inuktitut
ja	Japanese
jw	Javanese
ka	Georgian
kk	Kazakh
kl	Greenlandic
km	Cambodian
kn	Kannada
ko	Korean
ks	Kashmiri
ku	Kurdish
kw	Cornish
ky	Kirghiz
la	Latin
lb	Luxemburgish
ln	Lingala
lo	Laothian (recte Laotian)
lt	Lithuanian
lv	Latvian; Lettish
mg	Malagasy
mi	Maori
mk	Macedonian
ml	Malayalam
mn	Mongolian
mo	Moldavian
mr	Marathi
ms	Malay
mt	Maltese
my	Burmese
na	Nauru
ne	Nepali
nl	Dutch
no	Norwegian
oc	Occitan
om	(Afan) Oromo
or	Oriya
pa	Punjabi
pl	Polish
ps	Pashto, Pushto
pt	Portuguese
qu	Quechua
rm	Rhaeto-Romance
rn	Kirundi
ro	Romanian
ru	Russian
rw	Kinyarwanda
sa	Sanskrit
sd	Sindhi
se	Northern Sámi
sg	Sangho
sh	Serbo-Croatian
si	Singhalese
sk	Slovak
sl	Slovenian
sm	Samoan
sn	Shona
so	Somali
sq	Albanian
sr	Serbian
ss	Siswati
st	Sesotho
su	Sundanese
sv	Swedish
sw	Swahili
ta	Tamil
te	Telugu
tg	Tajik
th	Thai
ti	Tigrinya
tk	Turkmen
tl	Tagalog
tn	Setswana
to	Tonga
tr	Turkish
ts	Tsonga
tt	Tatar
tw	Twi
ug	Uigur
uk	Ukrainian
ur	Urdu
uz	Uzbek
vi	Vietnamese
vo	Volapük
wo	Wolof
xh	Xhosa
yi	Yiddish (formerly ji)
yo	Yoruba
za	Zhuang
zh	Chinese
zu	Zulu

C ISO 3166 country codes

Afghanistan	AF	Costa Rica	CR
Albania	AL	Cote d'Ivoire	CI
Algeria	DZ	Croatia (local name: Hrvatska)	HR
American Samoa	AS	Cuba	CU
Andorra	AD	Cyprus	CY
Angola	AO	Czech Republic	CZ
Anguilla§	AI	Denmark	DK
Antarctica	AQ	Djibouti	DJ
Antigua And Barbuda	AG	Dominica	DM
Argentina	AR	Dominican Republic	DO
Armenia	AM	East Timor	TP
Aruba	AW	Ecuador	EC
Australia	AU	Egypt	EG
Austria	AT	El Salvador	SV
Azerbaijan	AZ	Equatorial Guinea	GQ
Bahamas	BS	Eritrea	ER
Bahrain	BH	Estonia	EE
Bangladesh	BD	Ethiopia	ET
Barbados	BB	Falkland Islands (Malvinas)	FK
Belarus	BY	Faroe Islands	FO
Belgium	BE	Fiji	FJ
Belize	BZ	Finland	FI
Benin	BJ	France	FR
Bermuda	BM	France, Metropolitan	FX
Bhutan	BT	French Guiana	GF
Bolivia	BO	French Polynesia	PF
Bosnia And Herzegowina	BA	French Southern Territories	TF
Botswana	BW	Gabon	GA
Bouvet Island	BV	Gambia	GM
Brazil	BR	Georgia	GE
British Indian Ocean Territory	IO	Germany	DE
Brunei Darussalam	BN	Ghana	GH
Bulgaria	BG	Gibraltar	GI
Burkina Faso	BF	Greece	GR
Burundi	BI	Greenland	GL
Cambodia	KH	Grenada	GD
Cameroon	CM	Guadeloupe	GP
Canada	CA	Guam	GU
Cape Verde	CV	Guatemala	GT
Cayman Islands	KY	Guinea	GN
Central African Republic	CF	Guinea-Bissau	GW
Chad	TD	Guyana	GY
Chile	CL	Haiti	HT
China	CN	Heard And Mc Donald Islands	HM
Christmas Island	CX	Honduras	HN
Cocos (Keeling) Islands	CC	Hong Kong	HK
Colombia	CO	Hungary	HU
Comoros	KM	Iceland	IS
Congo	CG	India	IN
Cook Islands	CK	Indonesia	ID

Iran (Islamic Republic Of)	IR
Iraq	IQ
Ireland	IE
Israel	IL
Italy	IT
Jamaica	JM
Japan	JP
Jordan	JO
Kazakhstan	KZ
Kenya	KE
Kiribati	KI
Korea, Democratic People's Republic Of	KP
Korea, Republic Of	KR
Kuwait	KW
Kyrgyzstan	KG
Lao People's Democratic Republic	LA
Latvia	LV
Lebanon	LB
Lesotho	LS
Liberia	LR
Libyan Arab Jamahiriya	LY
Liechtenstein	LI
Lithuania	LT
Luxembourg	LU
Macau	MO
Macedonia, The Former Yugoslav Republic Of	MK
Madagascar	MG
Malawi	MW
Malaysia	MY
Maldives	MV
Mali	ML
Malta	MT
Marshall Islands	MH
Martinique	MQ
Mauritania	MR
Mauritius	MU
Mayotte	YT
Mexico	MX
Micronesia, Federated States Of	FM
Moldova, Republic Of	MD
Monaco	MC
Mongolia	MN
Montserrat	MS
Morocco	MA
Mozambique	MZ
Myanmar	MM
Namibia	NA
Nauru	NR
Nepal	NP
Netherlands	NL
Netherlands Antilles	AN
New Caledonia	NC
New Zealand	NZ
Nicaragua	NI
Niger	NE
Nigeria	NG
Niue	NU
Norfolk Island	NF
Northern Mariana Islands	MP
Norway	NO
Oman	OM
Pakistan	PK
Palau	PW
Panama	PA
Papua New Guinea	PG
Paraguay	PY
Peru	PE
Philippines	PH
Pitcairn	PN
Poland	PL
Portugal	PT
Puerto Rico	PR
Qatar	QA
Reunion	RE
Romania	RO
Russian Federation	RU
Rwanda	RW
Saint Kitts And Nevis	KN
Saint Lucia	LC
Saint Vincent And The Grenadines	VC
Samoa	WS
San Marino	SM
Sao Tome And Principe	ST
Saudi Arabia	SA
Senegal	SN
Seychelles	SC
Sierra Leone	SL
Singapore	SG
Slovakia (Slovak Republic)	SK
Slovenia	SI
Solomon Islands	SB
Somalia	SO
South Africa	ZA
South Georgia And The South Sandwich Islands	GS
Spain	ES
Sri Lanka	LK
St. Helena	SH
St. Pierre And Miquelon	PM
Sudan	SD
Suriname	SR
Svalbard And Jan Mayen Islands	SJ
Swaziland	SZ

Sweden	SE
Switzerland	CH
Syrian Arab Republic	SY
Taiwan, Province Of China	TW
Tajikistan	TJ
Tanzania, United Republic Of	TZ
Thailand	TH
Togo	TG
Tokelau	TK
Tonga	TO
Trinidad And Tobago	TT
Tunisia	TN
Turkey	TR
Turkmenistan	TM
Turks And Caicos Islands	TC
Tuvalu	TV
Uganda	UG
Ukraine	UA
United Arab Emirates	AE
United Kingdom GB (	UK
) United States	US
United States Minor Outlying Islands	UM
Uruguay	UY
Uzbekistan	UZ
Vanuatu	VU
Vatican City State (Holy See)	VA
Venezuela	VE
Viet Nam	VN
Virgin Islands (British)	VG
Virgin Islands (U.S.)	VI
Wallis And Futuna Islands	WF
Western Sahara	EH
Yemen	YE
Yugoslavia	YU
Zaire	ZR
Zambia	ZM
Zimbabwe	ZW

D XML examples

The following examples are included with Morphon XML editor examples [Tec00].

D.1 Example of an article DTD

```
<!ELEMENT br EMPTY>

<!ELEMENT article (meta,title,subtitle?, date?,author?,source?,body)>
<!ATTLIST article
          id ID #REQUIRED>
<!-- meta
     contains data one wouldn't expect to see in a printed rendition of
     the article.
-->
<!ELEMENT meta (topics)>
<!ELEMENT topics (topic+)>
<!ELEMENT topic (#PCDATA)>

<!ELEMENT title (#PCDATA)>
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ATTLIST author
          email CDATA "webmaster@morphon.com">
<!ELEMENT source (#PCDATA)>
<!ELEMENT body (intro?,(p?,h1?,h2?,ul?,ol?,output?, listing?, quote?,sidebar?)
<!ELEMENT sidebar (sidetitle?,sidesubtitle?,sideauthor?,intro?,
(p?,h1?,h2?,ul?,ol?,quote?)*)>
<!ELEMENT sidetitle (#PCDATA)>
<!ELEMENT sidesubtitle (#PCDATA)>
<!ELEMENT sideauthor (#PCDATA)>
<!ENTITY % inline "#PCDATA | em | st | link | code">
<!ELEMENT p (%inline;)*>
<!ELEMENT output (#PCDATA)>
<!ELEMENT listing (#PCDATA)>
<!ELEMENT ol (li,(li?,ol?,ul?,p?)*)>
<!ELEMENT ul (li,(li?,ol?,ul?,p?)*)>
<!ELEMENT li (#PCDATA)>
<!ELEMENT h1 (#PCDATA)>
<!ELEMENT h2 (#PCDATA)>
<!ELEMENT em (#PCDATA)>
<!ELEMENT st (#PCDATA)>
<!ELEMENT link (#PCDATA)>
<!ATTLIST link
          url CDATA "http://www.morphon.com/">
<!ELEMENT code (#PCDATA)>
<!ELEMENT intro (p+)>
<!ELEMENT quote (p+)>
```

D.2 Example of an article instance

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE article SYSTEM "article.dtd">
<article id="mass-editing">
  <meta>
    <topics>
      <topic>itnews</topic>
      <topic>perl</topic>
    </topics>
  </meta>
  <title>Changing strings in a bunch of files</title>
  <date>June 22, 1999</date>
  <author>Bart Schuller</author>
  <body>
    <intro>
<p>
A colleague reminded me that he again didn't write down the handy tricks one
can use for mass-editing files. So here they are. The examples in this article
use the GNU versions of common Unix tools, as well as perl5.
</p><p>
The recipes given here are incremental, which makes it easier to
understand, adapt and debug them.
</p>
    </intro>

<h1>Finding the files you want</h1>
<p>
First you need to ask yourself which files you want to operate on:
</p>
<h2>All files in a directory:</h2>
<listing>ls -1 *</listing>
<h2>All files in all directories:</h2>
<listing>find . -type f</listing>
<h2>All html files in all directories:</h2>
<listing>find . -name \*.html</listing>

<p>
More advanced file finding will follow after some other tricks.
Read the manual for <em>find(1)</em> for lots more criteria, including
selection by modification time.
</p>

<h1>Changing a file using perl</h1>
<p>
Perl has a number of command line options that make it easy to write
powerful oneliners:
</p>
<h2>Put a program on the command line:</h2>
<listing>perl -e 'print scalar(localtime), "\n"'</listing>
<h2>Implicitly loop through all lines of all files:</h2>
<listing>perl -n -e 'print if /^schuller/' /etc/passwd /etc/group</listing>
```

```
<h2>Loop and print:</h2>
<listing>perl -p -e 's/chuller/martass/' /etc/passwd /etc/group</listing>
<h2>Edit files in place with backups</h2>
<listing>perl -p -i.bak -e 's/Moron/Customer/g' *.txt</listing>

<p>
  If you're really sure of yourself you can tell perl to not generate
  backup copies:
</p>
<listing>perl -pi -e '&lt;statements&gt;'</listing>
<p>
  If an edit with backups didn't turn out quite right, you can move the
  backups back to their original names using the <em>rename</em> perl script:
</p>
<sidebar><p>
The <em>rename</em> script is probably not installed at your site, even
if the rest of perl is installed. It comes with the perl source
distribution but is also available as <link url="http://www.cpan.org/authors/i
<listing>rename 's/\.bak$//' *.bak</listing>

<h1>Putting two and two together</h1>
<p>
We've seen ways to generate lists of filenames. We've seen ways to
change files that are named on the command line. Here's how you can
combine the techniques using the <code>xargs</code> command. It runs the comma
you give it over all files that you feed it through the standard input.
You can experiment with xargs by putting for example <code>| xargs echo</code>
or <code>|xargs ls -l</code> after the file-finding commands from the first
section.
</p>
<h2>Finding files containing a certain string:</h2>
<listing>grep 'a string' *.html</listing>
<h2>Listing only the filenames</h2>
<listing>grep -l 'a string' *.html</listing>
<h2>Searching recursively, printing only filenames</h2>
<listing>find . -name \*.html | xargs grep -l 'a string'</listing>
<p>
  The result of this command is still a list of filenames, so you can
  feed it to another instance of <code>| xargs grep -l 'something'</code> if y
  want.
</p>
<h2>Changing "foo" to "bar" in all files under the current directory</h2>
<listing>find . -type f | xargs perl -pi -e 's/\bfoo\b/bar/g'</listing>


<h1>Developing your oneliner</h1>
<p>
As you can see, it's best to develop the "select the right files" and
the "change the files" parts independently. You can try out the editing
operations by naming just one example file on the commandline:
</p>
<listing>perl -pi -e 's/\b([a-z])/\u$1/g' example.txt</listing>
```

```
<p>
The same kind of expressions can also be useful with the "rename"
script:
</p>
<listing>
rename 's/\.htm$/.html/' *.htm

rename 'tr[A-Z][a-z]' *
</listing>

<h1>Advanced "find" and "xargs" usage</h1>
<p>
The examples given here don't work when your filenames contain spaces.
Under windows, that's where you'll lose all hope of using command line
tools. Under GNU systems, you can use the following:
</p>
<h2>Create an interesting testcase and verify the breakage:</h2>
<listing>
touch "Program Files!"
find . -type f | xargs ls -l
</listing>
<h2>Have "find" print a null character as the filename delimiter</h2>
<listing>find . -type f -print0</listing>
<h2>Have xargs act on null-separated filenames:</h2>
<listing>find . -type f -print0 | xargs --null ls -l</listing>
<h2>Remove the files the easy way:</h2>
<listing>rm Program&lt;tab&gt;</listing>
<h2>Alternatively, use your newfound knowledge:</h2>
<listing>find . -name '* *' -print0| xargs --null rm</listing>
</body>
</article>
```

D.3 Example of a play DTD

```
<!-- DTD for Shakespeare    J. Bosak    1994.03.01, 1997.01.02 -->
<!-- Revised for case sensitivity 1997.09.10 -->
<!-- Revised for XML 1.0 conformity 1998.01.27 (thanks to Eve Maler) -->

<!ENTITY amp "&#38;">
<!ELEMENT PLAY     (TITLE, FM, PERSONAE, SCNDESCR, PLAYSUBT, INDUCT?,
             PROLOGUE?, ACT+, EPILOGUE?)>
<!ELEMENT TITLE    (#PCDATA)>
<!ELEMENT FM       (P+)>
<!ELEMENT P        (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP   (PERSONA+, GRPDESCR)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
```

```
<!ELEMENT INDUCT   (TITLE, SUBTITLE*, (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT      (TITLE, SUBTITLE*, PROLOGUE?, SCENE+, EPILOGUE?)>
<!ELEMENT SCENE    (TITLE, SUBTITLE*, (SPEECH | STAGEDIR | SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
<!ELEMENT SPEAKER  (#PCDATA)>
<!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD  (#PCDATA)>
```

D.4 Example of a play instance

```
<?xml version="1.0"?>
<!DOCTYPE PLAY SYSTEM "play.dtd">

<PLAY>
<TITLE>The Tempest</TITLE>

<FM>
<P>Text placed in the public domain by Moby Lexical Tools, 1992.</P>
<P>SGML markup by Jon Bosak, 1992-1994.</P>
<P>XML version by Jon Bosak, 1996-1998.</P>
<P>This work may be freely copied and distributed worldwide.</P>
</FM>


<PERSONAE>
<TITLE>Dramatis Personae</TITLE>

<PERSONA>ALONSO, King of Naples.</PERSONA>
<PERSONA>SEBASTIAN, his brother.</PERSONA>
<PERSONA>PROSPERO, the right Duke of Milan.</PERSONA>
<PERSONA>ANTONIO, his brother, the usurping Duke of Milan.</PERSONA>
<PERSONA>FERDINAND, son to the King of Naples.</PERSONA>
<PERSONA>GONZALO, an honest old Counsellor.</PERSONA>

<PGROUP>
<PERSONA>ADRIAN</PERSONA>
<PERSONA>FRANCISCO</PERSONA>
<GRPDESCR>Lords.</GRPDESCR>
</PGROUP>

<PERSONA>CALIBAN, a savage and deformed Slave.</PERSONA>
<PERSONA>TRINCULO, a Jester.</PERSONA>
<PERSONA>STEPHANO, a drunken Butler.</PERSONA>
<PERSONA>Master of a Ship. </PERSONA>
<PERSONA>Boatswain. </PERSONA>
<PERSONA>Mariners. </PERSONA>
```

```
<PERSONA>MIRANDA, daughter to Prospero.</PERSONA>
<PERSONA>ARIEL, an airy Spirit.</PERSONA>

<PGROUP>
<PERSONA>IRIS</PERSONA>
<PERSONA>CERES</PERSONA>
<PERSONA>JUNO</PERSONA>
<PERSONA>Nymphs</PERSONA>
<PERSONA>Reapers</PERSONA>
<GRPDESCR>presented by Spirits.</GRPDESCR>
</PGROUP>

<PERSONA>Other Spirits attending on Prospero.</PERSONA>
</PERSONAE>

<SCNDESCR>SCENE  A ship at Sea: an island.</SCNDESCR>

<PLAYSUBT>THE TEMPEST</PLAYSUBT>

<ACT><TITLE>ACT I</TITLE>

<SCENE><TITLE>SCENE I.  On a ship at sea: a tempestuous noise of thunder and l

<STAGEDIR>Enter a Master and a Boatswain</STAGEDIR>

<SPEECH>
<SPEAKER>Master</SPEAKER>
<LINE>Boatswain!</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>Here, master: what cheer?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Master</SPEAKER>
<LINE>Good, speak to the mariners: fall to't, yarely,</LINE>
<LINE>or we run ourselves aground: bestir, bestir.</LINE>
</SPEECH>

<STAGEDIR>Exit</STAGEDIR>
<STAGEDIR>Enter Mariners</STAGEDIR>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>Heigh, my hearts! cheerly, cheerly, my hearts!</LINE>
<LINE>yare, yare! Take in the topsail. Tend to the</LINE>
<LINE>master's whistle. Blow, till thou burst thy wind,</LINE>
<LINE>if room enough!</LINE>
</SPEECH>
```

<STAGEDIR>Enter ALONSO, SEBASTIAN, ANTONIO, FERDINAND,
GONZALO, and others</STAGEDIR>

<SPEECH>
<SPEAKER>ALONSO</SPEAKER>
<LINE>Good boatswain, have care. Where’s the master?</LINE>
<LINE>Play the men.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>I pray now, keep below.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>ANTONIO</SPEAKER>
<LINE>Where is the master, boatswain?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>Do you not hear him? You mar our labour: keep your</LINE>
<LINE>cabins: you do assist the storm.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>GONZALO</SPEAKER>
<LINE>Nay, good, be patient.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>When the sea is. Hence! What cares these roarers</LINE>
<LINE>for the name of king? To cabin: silence! trouble us not.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>GONZALO</SPEAKER>
<LINE>Good, yet remember whom thou hast aboard.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>None that I more love than myself. You are a</LINE>
<LINE>counsellor; if you can command these elements to</LINE>
<LINE>silence, and work the peace of the present, we will</LINE>
<LINE>not hand a rope more; use your authority: if you</LINE>
<LINE>cannot, give thanks you have lived so long, and make</LINE>
<LINE>yourself ready in your cabin for the mischance of</LINE>
<LINE>the hour, if it so hap. Cheerly, good hearts! Out</LINE>
<LINE>of our way, I say.</LINE>

```
</SPEECH>


<STAGEDIR>Exit</STAGEDIR>

<SPEECH>
<SPEAKER>GONZALO</SPEAKER>
<LINE>I have great comfort from this fellow: methinks he</LINE>
<LINE>hath no drowning mark upon him; his complexion is</LINE>
<LINE>perfect gallows. Stand fast, good Fate, to his</LINE>
<LINE>hanging: make the rope of his destiny our cable,</LINE>
<LINE>for our own doth little advantage. If he be not</LINE>
<LINE>born to be hanged, our case is miserable.</LINE>
</SPEECH>

<STAGEDIR>Exeunt</STAGEDIR>
<STAGEDIR>Re-enter Boatswain</STAGEDIR>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>Down with the topmast! yare! lower, lower! Bring</LINE>
<LINE>her to try with main-course.</LINE>
<STAGEDIR>A cry within</STAGEDIR>
<LINE>A plague upon this howling! they are louder than</LINE>
<LINE>the weather or our office.</LINE>
<STAGEDIR>Re-enter SEBASTIAN, ANTONIO, and GONZALO</STAGEDIR>
<LINE>Yet again! what do you here? Shall we give o’er</LINE>
<LINE>and drown? Have you a mind to sink?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>SEBASTIAN</SPEAKER>
<LINE>A pox o’ your throat, you bawling, blasphemous,</LINE>
<LINE>incharitable dog!</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>Work you then.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>ANTONIO</SPEAKER>
<LINE>Hang, cur! hang, you whoreson, insolent noisemaker!</LINE>
<LINE>We are less afraid to be drowned than thou art.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>GONZALO</SPEAKER>
<LINE>I’ll warrant him for drowning; though the ship were</LINE>
<LINE>no stronger than a nutshell and as leaky as an</LINE>
<LINE>unstanched wench.</LINE>
```

```
</SPEECH>

<SPEECH>
<SPEAKER>Boatswain</SPEAKER>
<LINE>Lay her a-hold, a-hold! set her two courses off to</LINE>
<LINE>sea again; lay her off.</LINE>
</SPEECH>

...

</ACT>
</PLAY>
```

E Example SAX Applications

In the following we have two simple examples of SAX applications for parsing XML documents. Both application will print out the events they encounters. The first application uses a single `DefaultHandler` for processing all events. The second application separates the `ContentHandler` from the

`ErrorHandler`.

E.1 A simple SAX application

```
import java.io.FileReader;
import org.xml.sax.XMLReader;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;

public class MySAXApp extends DefaultHandler
{

    public static void main (String args[])
        throws Exception
    {
 XMLReader xr =
  XMLReaderFactory.createXMLReader(
          "org.apache.xerces.parsers.SAXParser");
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

                                // Parse each file provided on the
                                // command line.
        for (int i = 0; i < args.length; i++) {
            FileReader r = new FileReader(args[i]);
            xr.parse(new InputSource(r));
        }
    }


    public MySAXApp ()
    {
        super();
    }


    //////////////////////////////////////////////////////////
    // Event handlers.
    //////////////////////////////////////////////////////////
```

```
public void startDocument ()
{
    System.out.println("Start document");
}


public void endDocument ()
{
    System.out.println("End document");
}


public void startElement (String uri, String name,
                          String qName, Attributes atts)
{
    System.out.println("Start element: {" + uri + "}" + name);
}


public void endElement (String uri, String name, String qName)
{
    System.out.println("End element:   {" + uri + "}" + name);
}


public void characters (char ch[], int start, int length)
{
    System.out.print("Characters:    \"");
    for (int i = start; i < start + length; i++) {
        switch (ch[i]) {
        case '\\':
            System.out.print("\\\\");
            break;
        case '"':
            System.out.print("\\\"");
            break;
        case '\n':
            System.out.print("\\n");
            break;
        case '\r':
            System.out.print("\\r");
            break;
        case '\t':
            System.out.print("\\t");
            break;
        default:
            System.out.print(ch[i]);
            break;
        }
    }
    System.out.print("\"\n");
```

```
        }

    }
```

E.2 Another simple SAX application

```
/*--

 Copyright (C) 2000 Brett McLaughlin. All rights reserved.

 Redistribution and use in source and binary forms, with or without modifica-
 tion, are permitted provided that the following conditions are met:

 1. Redistributions of source code must retain the above copyright notice,
    this list of conditions, and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright notice,
    this list of conditions, the disclaimer that follows these conditions,
    and/or other materials provided with the distribution.

 3. Products derived from this software may not be called "Java and XML", nor
    may "Java and XML" appear in their name, without prior written permission
    from Brett McLaughlin (brett@newInstance.com).

 THIS SOFTWARE IS PROVIDED ''AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES,
 INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS  FOR A PARTICULAR  PURPOSE ARE  DISCLAIMED.  IN NO  EVENT SHALL  THE
 JDOM PROJECT  OR ITS CONTRIBUTORS  BE LIABLE FOR  ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL,  EXEMPLARY, OR CONSEQUENTIAL  DAMAGES (INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT  OF SUBSTITUTE GOODS OR SERVICES; LOSS
 OF USE, DATA, OR  PROFITS; OR BUSINESS  INTERRUPTION)  HOWEVER CAUSED AND ON
 ANY  THEORY OF LIABILITY,  WHETHER  IN CONTRACT,  STRICT LIABILITY,  OR TORT
 (INCLUDING  NEGLIGENCE OR  OTHERWISE) ARISING IN  ANY WAY OUT OF THE  USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

 This software was originally created by Brett McLaughlin
 <brett@newInstance.com>.  For more  information on "Java and XML", please
 see <http://www.oreilly.com/catalog/javaxml/> or <http://www.newInstance.com>

 */
import java.io.IOException;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
```

```
/**
 * <b><code>SAXParserDemo</code></b> will take an XML file and parse it using
 *   SAX, displaying the callbacks in the parsing lifecycle.
 *
 * @author Brett McLaughlin
 * @version 1.0
 */
public class SAXParserDemo {

    /**
     * <p>
     * This parses the file, using registered SAX handlers, and output
     *   the events in the parsing process cycle.
     * </p>
     *
     * @param uri <code>String</code> URI of file to parse.
     */
    public void performDemo(String uri) {
        System.out.println("Parsing XML File: " + uri + "\n\n");

        // Get instances of our handlers
        ContentHandler contentHandler = new MyContentHandler();
        ErrorHandler errorHandler = new MyErrorHandler();

        try {
            // Instantiate a parser
            XMLReader parser =
                XMLReaderFactory.createXMLReader(

//                    "com.jclark.xml.sax.Driver");
                  "org.apache.xerces.parsers.SAXParser");

            // Register the content handler
            parser.setContentHandler(contentHandler);

            //parser.setFeature("http://xml.org/sax/features/validation",
                true);

            // Register the error handler
            parser.setErrorHandler(errorHandler);

            // Parse the document
            parser.parse(uri);

        } catch (IOException e) {
            System.out.println("Error reading URI: " + e.getMessage());
        } catch (SAXException e) {
            System.out.println("Error in parsing: " + e.getMessage());
        }
    }
```

```
    /**
     * <p>
     * This provides a command line entry point for this demo.
     * </p>
     */
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java SAXParserDemo [XML URI]");
            System.exit(0);
        }

        String uri = args[0];

        SAXParserDemo parserDemo = new SAXParserDemo();
        parserDemo.performDemo(uri);
    }
}

/**
 * <b><code>MyContentHandler</code></b> implements the SAX
 *   <code>ContentHandler</code> interface and defines callback
 *   behavior for the SAX callbacks associated with an XML
 *   document's content.
 */
class MyContentHandler implements ContentHandler {

    /** Hold onto the locator for location information */
    private Locator locator;

    /**
     * <p>
     * Provide reference to <code>Locator</code> which provides
     *   information about where in a document callbacks occur.
     * </p>
     *
     * @param locator <code>Locator</code> object tied to callback
     *                process
     */
    public void setDocumentLocator(Locator locator) {
        System.out.println("    * setDocumentLocator() called");
        // We save this for later use if desired.
        this.locator = locator;
    }

    /**
     * <p>
     * This indicates the start of a Document parse - this precedes
     *   all callbacks in all SAX Handlers with the sole exception
     *   of <code>{@link #setDocumentLocator}</code>.
     * </p>
     *
     * @throws <code>SAXException</code> when things go wrong
```

```
 */
public void startDocument() throws SAXException {
    System.out.println("Parsing begins...");
}

/**
 * <p>
 * This indicates the end of a Document parse - this occurs after
 *   all callbacks in all SAX Handlers.</code>.
 * </p>
 *
 * @throws <code>SAXException</code> when things go wrong
 */
public void endDocument() throws SAXException {
    System.out.println("...Parsing ends.");
}

/**
 * <p>
 * This will indicate that a processing instruction (other than
 *   the XML declaration) has been encountered.
 * </p>
 *
 * @param target <code>String</code> target of PI
 * @param data <code>String</code containing all data sent to the PI.
 *             This typically looks like one or more attribute value
 *             pairs.
 * @throws <code>SAXException</code> when things go wrong
 */
public void processingInstruction(String target, String data)
    throws SAXException {

    System.out.println("PI: Target:" + target + " and Data:" + data);
}

/**
 * <p>
 * This will indicate the beginning of an XML Namespace prefix
 *   mapping.  Although this typically occur within the root element
 *   of an XML document, it can occur at any point within the
 *   document.  Note that a prefix mapping on an element triggers
 *   this callback <i>before</i> the callback for the actual element
 *   itself (<code>{@link #startElement}</code>) occurs.
 * </p>
 *
 * @param prefix <code>String</code> prefix used for the namespace
 *               being reported
 * @param uri <code>String</code> URI for the namespace
 *            being reported
 * @throws <code>SAXException</code> when things go wrong
 */
public void startPrefixMapping(String prefix, String uri) {
```

```
    System.out.println("Mapping starts for prefix " + prefix +
                       " mapped to URI " + uri);
}

/**
 * <p>
 * This indicates the end of a prefix mapping, when the namespace
 *   reported in a <code>{@link #startPrefixMapping}</code> callback
 *   is no longer available.
 * </p>
 *
 * @param prefix <code>String</code> of namespace being reported
 * @throws <code>SAXException</code> when things go wrong
 */
public void endPrefixMapping(String prefix) {
    System.out.println("Mapping ends for prefix " + prefix);
}

/**
 * <p>
 * This reports the occurrence of an actual element.  It will include
 *   the element's attributes, with the exception of XML vocabulary
 *   specific attributes, such as
 *   <code>xmlns:[namespace prefix]</code> and
 *   <code>xsi:schemaLocation</code>.
 * </p>
 *
 * @param namespaceURI <code>String</code> namespace URI this element
 *                     is associated with, or an empty
 *                     <code>String</code>
 * @param localName <code>String</code> name of element (with no
 *                  namespace prefix, if one is present)
 * @param rawName <code>String</code> XML 1.0 version of element name:
 *                [namespace prefix]:[localName]
 * @param atts <code>Attributes</code> list for this element
 * @throws <code>SAXException</code> when things go wrong
 */
public void startElement(String namespaceURI, String localName,
                         String rawName, Attributes atts)
    throws SAXException {

    System.out.print("startElement: " + localName);
    if (!namespaceURI.equals("")) {
        System.out.println(" in namespace " + namespaceURI +
                           " (" + rawName + ")");
    } else {
        System.out.println(" has no associated namespace");
    }

    for (int i=0; i<atts.getLength(); i++)
        System.out.println("  Attribute: " + atts.getLocalName(i) +
                           "=" + atts.getValue(i));
```

```
}

/**
 * <p>
 * Indicates the end of an element
 *   (<code>&lt;/[element name]&gt;</code>) is reached.  Note that
 *   the parser does not distinguish between empty
 *   elements and non-empty elements, so this will occur uniformly.
 * </p>
 *
 * @param namespaceURI <code>String</code> URI of namespace this
 *                     element is associated with
 * @param localName <code>String</code> name of element without prefix
 * @param rawName <code>String</code> name of element in XML 1.0 form
 * @throws <code>SAXException</code> when things go wrong
 */
public void endElement(String namespaceURI, String localName,
                       String rawName)
    throws SAXException {

    System.out.println("endElement: " + localName + "\n");
}

/**
 * <p>
 * This will report character data (within an element).
 * </p>
 *
 * @param ch <code>char[]</code> character array with character data
 * @param start <code>int</code> index in array where data starts.
 * @param end <code>int</code> index in array where data ends.
 * @throws <code>SAXException</code> when things go wrong
 */
public void characters(char[] ch, int start, int end)
    throws SAXException {

    String s = new String(ch, start, end);
    System.out.println("characters: " + s);
}

    /**
 * <p>
 * This will report whitespace that can be ignored in the
 *   originating document.  This is typically only invoked when
 *   validation is ocurring in the parsing process.
 * </p>
 *
 * @param ch <code>char[]</code> character array with character data
 * @param start <code>int</code> index in array where data starts.
 * @param end <code>int</code> index in array where data ends.
 * @throws <code>SAXException</code> when things go wrong
 */
```

```
    public void ignorableWhitespace(char[] ch, int start, int end)
        throws SAXException {

        String s = new String(ch, start, end);
        System.out.println("ignorableWhitespace: [" + s + "]");
    }

        /**
     * <p>
     * This will report an entity that is skipped by the parser.  This
     *   should only occur for non-validating parsers, and then is still
     *   implementation-dependent behavior.
     * </p>
     *
     * @param name <code>String</code> name of entity being skipped
     * @throws <code>SAXException</code> when things go wrong
     */
    public void skippedEntity(String name) throws SAXException {
        System.out.println("Skipping entity " + name);
    }

}

/**
 * <b><code>MyErrorHandler</code></b> implements the SAX
 *   <code>ErrorHandler</code> interface and defines callback
 *   behavior for the SAX callbacks associated with an XML
 *   document's errors.
 */
class MyErrorHandler implements ErrorHandler  {

    /**
     * <p>
     * This will report a warning that has occurred; this indicates
     *   that while no XML rules were "broken", something appears
     *   to be incorrect or missing.
     * </p>
     *
     * @param exception <code>SAXParseException</code> that occurred.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void warning(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Warning**\n" +
                           "  Line:    " +
                              exception.getLineNumber() + "\n" +
                           "  URI:     " +
                              exception.getSystemId() + "\n" +
                           "  Message: " +
                              exception.getMessage());
        throw new SAXException("Warning encountered");
```

```
    }

    /**
     * <p>
     * This will report an error that has occurred; this indicates
     *   that a rule was broken, typically in validation, but that
     *   parsing can reasonably continue.
     * </p>
     *
     * @param exception <code>SAXParseException</code> that occurred.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void error(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Error**\n" +
                           "  Line:    " +
                              exception.getLineNumber() + "\n" +
                           "  URI:     " +
                              exception.getSystemId() + "\n" +
                           "  Message: " +
                              exception.getMessage());
        throw new SAXException("Error encountered");
    }
    /**
     * <p>
     * This will report a fatal error that has occurred; this indicates
     *   that a rule has been broken that makes continued parsing either
     *   impossible or an almost certain waste of time.
     * </p>
     *
     * @param exception <code>SAXParseException</code> that occurred.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void fatalError(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Fatal Error**\n" +
                           "  Line:    " +
                              exception.getLineNumber() + "\n" +
                           "  URI:     " +
                              exception.getSystemId() + "\n" +
                           "  Message: " +
                              exception.getMessage());
        throw new SAXException("Fatal Error encountered");
    }
}
```

F Example DOM Application

In the following we have a simple example of a DOM applications for processing XML documents. The application will print out certain characteristics of all the nodes in the DOM tree.

```
import java.io.IOException;
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXException;


public class DOMParserDemo {

    public void printNode (Node node, String indent) {
        switch (node.getNodeType()) {
        case Node.DOCUMENT_NODE:
            System.out.println(indent + "<xml version=\"1.0\">");
            NodeList nodes = node.getChildNodes();
            if (nodes != null) {
                for (int i=0; i < nodes.getLength(); i++) {
                    printNode (nodes.item(i), "");
                }
            }
            break;
        case Node.ELEMENT_NODE:
            String name = node.getNodeName();
            System.out.print(indent + "<" + name);
            NamedNodeMap attributes = node.getAttributes();
            for (int i = 0; i < attributes.getLength(); i++) {
                Node current = attributes.item(i);
                System.out.print(" " + current.getNodeName() + "=\"" +
                        current.getNodeValue() + "\"");
            }
            System.out.println(">");
            NodeList children = node.getChildNodes ();
            if (children != null) {
                for (int i=0; i < children.getLength(); i++) {
                    printNode (children.item(i), indent + "  ");
                }
            }
            System.out.println (indent + "</" + name + ">");
            break;
        case Node.TEXT_NODE:
        case Node.CDATA_SECTION_NODE:
            System.out.println (indent+node.getNodeValue());
            break;
        case Node.PROCESSING_INSTRUCTION_NODE:
```

```
            System.out.println (indent + "<?" + node.getNodeName() + " " +
                node.getNodeValue () + " ?>");
            break;
        case Node.ENTITY_REFERENCE_NODE:
            System.out.println ("&" + node.getNodeName() + ";");
            break;
        case Node.DOCUMENT_TYPE_NODE:
            DocumentType docType = (DocumentType)node;
            System.out.print ("<!DOCTYPE " + docType.getName());
            if (docType.getPublicId() != null) {
                System.out.print ("PUBLIC \"" + docType.getPublicId () +
                "\"");
            } else {
                System.out.print (" SYSTEM ");
            }
            System.out.println ("\"" + docType.getSystemId () + "\"");
            break;
        }
    }

    public void performDemo (String uri) {
        System.out.println("Parsing XML File: " + uri + "\n");

        DOMParser parser = new DOMParser ();

        try {
            parser.parse (uri);
            Document doc = parser.getDocument ();
            printNode (doc, "");
        } catch (IOException e) {
            System.out.println ("Error in reading " + e.getMessage());
        } catch (SAXException e) {
            System.out.println ("Error in parsing " + e.getMessage());
        }
    }

    public static void main (String[] args) {
        if (args.length != 1) {
            System.out.println ("Usage: java DOMParserDemo [XML URI]");
            System.exit (0);
        }
        String uri = args[0];
        DOMParserDemo parserDemo = new DOMParserDemo();
        parserDemo.performDemo (uri);
    }
}
```

G Example XSL Specification and FO output

In the following a n example XSl specification with corresponding input and output when procssed with XSLT.

G.1 XSL file

```
<?xml version="1.0"?>
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
        xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="doc">

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <fo:layout-master-set>
    <fo:simple-page-master master-name="single"
                  page-height="21cm"
                  page-width="29.7cm"
                  margin-top="1cm"
                  margin-bottom="2cm"
                  margin-left="2.5cm"
                  margin-right="2.5cm">
      <fo:region-body margin-top="3cm"/>
      <fo:region-before extent="3cm"/>
      <fo:region-after extent="1.5cm"/>
    </fo:simple-page-master>

  </fo:layout-master-set>

  <fo:page-sequence master-name="single">

    <fo:static-content flow-name="xsl-region-before">
                <xsl:apply-templates select="header"/>
    </fo:static-content>

    <fo:flow flow-name="xsl-region-body">
                <xsl:apply-templates select="body"/>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
</xsl:template>

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

<xsl:template match="para">
        <fo:block font-size="12pt"
```

```
                font-family="sans-serif"
                line-height="15pt"
                space-after.optimum="3pt"
                text-align="start">
                <xsl:apply-templates/>
        </fo:block>
</xsl:template>

<xsl:template match="title[@level='1']">
        <fo:block font-size="18pt"
                font-family="sans-serif"
                line-height="24pt"
                space-after.optimum="15pt"
                background-color="blue"
                color="white"
                text-align="center"
                padding-top="3pt">
                <xsl:apply-templates/>
        </fo:block>
</xsl:template>

<xsl:template match="title[@level='2']">
        <fo:block font-size="16pt"
                font-family="sans-serif"
                line-height="20pt"
                space-before.optimum="10pt"
                space-after.optimum="10pt"
                text-align="start"
                padding-top="3pt">
        <xsl:apply-templates/>
        </fo:block>
</xsl:template>

<xsl:template match="header">
        <fo:block font-size="10pt">
        <xsl:apply-templates/>
        </fo:block>
</xsl:template>

<xsl:template match="page-number">
        <fo:page-number/>
</xsl:template>

</xsl:stylesheet>
```

G.2 XML document: input

```
<doc>
<header>XML Recommendation - p. <page-number/></header>
<body>
```

```
<title level="1">Extensible Markup Language (XML) 1.0</title>
<title level="2">Abstract</title>
<para>
The Extensible Markup Language (XML) is a subset of SGML that is
completely described in this document. Its goal is to enable
generic SGML to be served, received, and processed on the Web in
the way that is now possible with HTML. XML has been designed for
ease of implementation and for interoperability with both SGML and HTML.
</para>
<para>
This document has been reviewed by W3C Members and other interested
parties and has been endorsed by the Director as a W3C Recommendation.
It is a stable document and may be used as reference material or cited
as a normative reference from another document. W3C's role in making
the Recommendation is to draw attention to the specification and to
promote its widespread deployment. This enhances the functionality
and interoperability of the Web.
</para>
<title level="2">Status of this Document</title>
<para>
This document specifies a syntax created by subsetting an existing,
widely used international text processing standard (Standard Generalized
Markup Language, ISO 8879:1986(E) as amended and corrected) for use
on the World Wide Web. It is a product of the W3C XML Activity, details
of which can be found at http://www.w3.org/XML. A list of current W3C
Recommendations and other technical documents can be found at
http://www.w3.org/TR.
</para>
</body>
</doc>
```

G.3 FO Output

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
<fo:simple-page-master margin-right="2.5cm" margin-left="2.5cm"
 margin-bottom="2cm" margin-top="1cm" page-width="29.7cm"
 page-height="21cm" master-name="single">
<fo:region-body margin-top="3cm"/>
<fo:region-before extent="3cm"/>
<fo:region-after extent="1.5cm"/>
</fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-name="single">
<fo:static-content flow-name="xsl-region-before">
<fo:block font-size="10pt">XML Recommendation - p.<fo:page-number/>
</fo:block></fo:static-content><fo:flow flow-name="xsl-region-body">

<fo:block padding-top="3pt" text-align="center" color="white"
```

```
background-color="blue" space-after.optimum="15pt" line-height="24pt"
font-family="sans-serif"
font-size="18pt">Extensible Markup Language (XML) 1.0</fo:block>
<fo:block padding-top="3pt" text-align="start" space-after.optimum="10pt"
space-before.optimum="10pt" line-height="20pt" font-family="sans-serif"
font-size="16pt">Abstract</fo:block>
<fo:block text-align="start" space-after.optimum="3pt" line-height="15pt"
font-family="sans-serif" font-size="12pt">
The Extensible Markup Language (XML) is a subset of SGML that is c
ompletely described in this document. Its goal is to enable generic
SGML to be served, received, and processed on the Web in the way that
is now possible with HTML. XML has been designed for ease of
implementation and for interoperability with both SGML and HTML.
</fo:block>
<fo:block text-align="start" space-after.optimum="3pt"
line-height="15pt" font-family="sans-serif" font-size="12pt">
This document has been reviewed by W3C Members and other interested
parties and has been endorsed by the Director as a W3C Recommendation.
It is a stable document and may be used as reference material or cited
as a normative reference from another document. W3C's role in making
the Recommendation is to draw attention to the specification and to
promote its widespread deployment. This enhances the functionality
and interoperability of the Web.
</fo:block>
<fo:block padding-top="3pt" text-align="start" space-after.optimum="10pt"
space-before.optimum="10pt" line-height="20pt" font-family="sans-serif"
font-size="16pt">Status of this Document</fo:block>
<fo:block text-align="start" space-after.optimum="3pt" line-height="15pt"
font-family="sans-serif" font-size="12pt">This document specifies a
syntax created by subsetting an existing, widely used international
text processing standard (Standard Generalized Markup Language,
ISO 8879:1986(E) as amended and corrected) for use on the World Wide
Web. It is a product of the W3C XML Activity, details of which can
be found at http://www.w3.org/XML. A list of current W3C Recommendations
and other technical documents can be found at http://www.w3.org/TR.
</fo:block>
</fo:flow></fo:page-sequence></fo:root>
```