Part I. Introduction

1

Getting Started with SGML/XML

$Revision: 1.2 $
$Date: 2002/01/06 16:25:42 $

This chapter is intended to provide a quick introduction to structured markup (SGML and XML). If you're already familiar with SGML or XML, you only need to skim this chapter.

To work with DocBook, you need to understand a few basic concepts of structured editing in general, and DocBook, in particular. That's covered here. You also need some concrete experience with the way a DocBook document is structured. That's covered in the next chapter.

HTML and SGML vs. XML

This chapter doesn't assume that you know what HTML is, but if you do, you have a starting point for understanding structured markup. HTML (Hypertext Markup Language) is a way of marking up text and graphics so that the most popular web browsers can interpret them. HTML consists of a set of markup tags with specific meanings. Moreover, HTML is a very basic type of SGML markup that is easy to learn and easy for computer applications to generate. But the simplicity of HTML is both its virtue and its weakness. Because of HTML's limitations, web users and programmers have had to extend and enhance it by a series of customizations and revisions that still fall short of accommodating current, to say nothing of future, needs.

SGML, on the other hand, is an international standard that describes how markup languages are defined. SGML does not consist of particular tags or the rules for their usage. HTML is an example of a markup language defined in SGML.

XML promises an intelligent improvement over HTML, and compatibility with it is already being built into the most popular web browsers. XML is not a new markup language designed to compete with HTML, and it's not designed to create conversion headaches for people with tons of HTML documents. XML is intended to alleviate compatibility problems with browser software; it's a new, easier version of the standard rules that govern the markup itself, or, in other words, a new version of SGML. The rules of XML are designed to make it easier to write both applications that interpret its type of markup and applications that generate its markup. XML was developed by a team of SGML experts who understood and sought to correct the problems of learning and implementing SGML. XML is also *extensible* markup, which means that it is customizable. A browser or word processor that is XML-capable will be able to read any XML-based markup language that an individual user defines.

In this book, we tend to describe things in terms of SGML, but where there are differences between SGML and XML (and there are only a few), we point them out. For our purposes, it doesn't really matter whether you use SGML or XML.

During the coming months, we anticipate that XML-aware web browsers and other tools will become available. Nevertheless, it's not unreasonable to do your authoring in SGML and your online publishing in XML or HTML. By the same token, it's not unreasonable to do your authoring in XML.

Basic SGML/XML Concepts

Here are the basic SGML/XML concepts you need to grasp:

- structured, semantic markup
- elements
- attributes
- entities

Structured and Semantic Markup

An essential characteristic of structured markup is that it explicitly distinguishes (and accordingly “marks up” within a document) the structure and semantic content of a document. It does not mark up the way in which the document will appear to the reader, in print or otherwise.

In the days before word processors it was common for a typed manuscript to be submitted to a publisher. The manuscript identified the logical structures of the documents (chapters, section titles, and so on), but said nothing about its appearance. Working independently of the author, a designer then developed a specification for the appearance of the document, and a typesetter marked up and applied the designer's format to the document.

Because presentation or appearance is usually based on structure and content, SGML markup logically precedes and generally determines the way a document will look to a reader. If you are familiar with strict, simple HTML markup, you know that a given document that is structurally the same can also look different on different computers. That's because the markup does not specify many aspects of a document's appearance, although it does specify many aspects of a document's structure.

Many writers type their text into a word processor, line-by-line and word-for-word, italicizing technical terms, underlining words for emphasis, or setting section headers in a font complementary to the body text, and finally, setting the headers off with a few carriage returns fore and aft. The format such a writer imposes on the words on the screen imparts structure to the document by changing its appearance in ways that a reader can more or less reliably decode. The reliability depends on how consistently and unambiguously the changes in type and layout are made. By contrast, an SGML/XML markup of a section header explicitly specifies that a specific piece of text is a section header. This assertion does not specify the presentation or appearance of the section header, but it makes the fact that the text is a section header completely unambiguous.

SGML and XML use named elements, delimited by angle brackets (“<” and “>”) to identify the markup in a document. In DocBook, a top-level section is `<sect1>`, so the title of a top-level section named *My First-Level Header* would be identified like this:

```
<sect1><title>My First-Level Header</title>
```

Note the following features of this markup:

Clarity

> A title begins with `< title>` and ends with `</title>`. The `sect1` also has an ending `</sect1>`, but we haven't shown the whole section so it's not visible.

Hierarchy

> “My First-Level Header” is the title of a top-level section because it occurs inside a title in a `sect1`. A `title` element occurring somewhere else, say in a `Chapter` element, would be the title of the chapter.

Plain text

> SGML documents can have varying character sets, but most are ASCII. XML documents use the Unicode character set. This makes SGML and XML documents highly portable across systems and tools.

In an SGML document, there is no obligatory difference between the size or face of the type in a first-level section header and the title of a book in a footnote or the first sentence of a body paragraph. All SGML files are simple text files without font changes or special characters.[1] Similarly, an SGML document does not specify the words in a text that are to be set in italic, bold, or roman type. Instead, SGML marks certain kinds of texts for their semantic content. For example, if a particular word is the name of a file, then the tags around it should specify that it is a filename:

```
Many mail programs read configuration information from the
users <filename>.mailrc</filename> file.
```

If the meaning of a phrase is particularly audacious, it might get tagged for boldness of thought instead of appearance. An SGML document contains all the information that a typesetter needs to lay out and typeset a printed page in the most effective and consistent way, but it does not specify the layout or the type.[2]

Not only is the structure of an SGML/XML document explicit, but it is also carefully controlled. An SGML document makes reference to a set of declarations—a document type definition (DTD)—that contains an inventory of tag names and specifies the combination rules for the various structural and semantic features that make up a document. What the distinctive features are and how they should be combined is “arbitrary” in the sense that almost any selection of features and rules of composition is theoretically possible. The DocBook DTD chooses a particular set of features and rules for its users.

Here is a specific example of how the DocBook DTD works. DocBook specifies that a third-level section can follow a second-level section but cannot follow a first-level section without an intervening second-level section.

This is valid:

```
<sect1><title>...</title>
  <sect2><title>...</title>
    <sect3><title>...</title>
      ...
    </sect3>
  </sect2>
</sect1>
```

This is not:

```
<sect1><title>...</title>
  <sect3><title>...</title>
    ...
  </sect3>
</sect1>
```

Because an SGML/XML document has an associated DTD that describes the valid, logical structures of the document, you can test the logical structure of any particular document against the DTD. This process is performed by a parser. An SGML processor must begin by parsing the document and determining if it is valid, that is, if it conforms to the rules specified in the DTD. XML processors are not required to check for validity, but it's always a good idea to check for validity when authoring. Because you can test and validate the structure of an SGML/XML document with software, a DocBook document containing a first-level section followed immediately by a third-level section will be identified as invalid, meaning that it's not a valid instance or example of a document defined by the DocBook DTD. Presumably,

---

[1] Some structured editors apply style to the document while it's being edited, using fonts and color to make the editing task easier, but this stylistic information is not stored in the actual SGML/XML document. Instead, it is provided by the editing application.

[2] The distinction between appearance or presentation and structure or content is essential to SGML, but there is a way to specify the appearance of an SGML document: attach a stylesheet to it. There are several standards for such stylesheets: CSS, XSL, FOSIs, and DSSSL. See Chapter 4.

a document with a logical structure won't normally jump from a first- to a third-level section, so the rule is a safeguard—but not a guarantee—of good writing, or at the very least, reasonable structure. A parser also verifies that the names of the tags are correct and that tags requiring an ending tag have them. This means that a valid document is also one that should format correctly, without runs of paragraphs incorrectly appearing in bold type or similar monstrosities that everyone has seen in print at one time or another. For more information about SGML/XML parsers, see Chapter 3.

In general, adherence to the explicit rules of structure and markup in a DTD is a useful and reassuring guarantee of consistency and reliability within documents, across document sets, and over time. This makes SGML/XML markup particularly desirable to corporations or governments that have large sets of documents to manage, but it is a boon to the individual writer as well.

How can this markup help you?

Semantic markup makes your documents more amenable to interpretation by software, especially publishing software. You can publish a white paper, authored as a DocBook `Article`, in the following formats:

- On the Web in HTML
- As a standalone document on 8½×11 paper
- As part of a quarterly journal, in a 6×9 format
- In Braille
- In audio

You can produce each of these publications from exactly the same source document using the presentational techniques best suited to both the content of the document and the presentation medium. This versatility also frees the author to concentrate on the document content. For example, as we write this book, we don't know exactly how O'Reilly will choose to present chapter headings, bulleted lists, SGML terms, or any of the other semantic features. And we don't care. It's irrelevant; whatever presentation is chosen, the SGML sources will be transformed automatically into that style.

Semantic markup can relieve the author of other, more significant burdens as well (after all, careful use of paragraph and character styles in a word processor document theoretically allows us to change the presentation independently from the document). Using semantic markup opens up your documents to a world of possibilities. Documents become, in a loose sense, databases of information. Programs can compile, retrieve, and otherwise manipulate the documents in predictable, useful ways.

Consider the online version of this book: almost every element name (`Article`, `Book`, and so on) is a hyperlink to the reference page that describes that element. Maintaining these links by hand would be tedious and might be unreliable, as well. Instead, every element name is marked as an element using `SGMLTag`: a `Book` is a `<sgmltag>Book</sgmltag>`.

Because each element name in this book is tagged semantically, the program that produces the online version can determine which occurrences of the word “book” in the text are actually references to the `Book` element. The program can then automatically generate the appropriate hyperlink when it should.

There's one last point to make about the versatility of SGML documents: how much you have depends on the DTD. If you take a good photo with a high resolution lens, you can print it and copy it and scan it and put it on the Web, and it will look good. If you start with a low-resolution picture it will not survive those transformations so well. DocBook SGML/XML has this advantage over, say, HTML: DocBook has specific and unambiguous semantic and structural markup, because you can convert its documents with ease into other presentational forms, and search them more precisely.

If you start with HTML, whose markup is at a lower resolution than DocBook's, your versatility and searchability is substantially restricted and cannot be improved.

What are the shortcomings to structural authoring?

There are a few significant shortcomings to structured authoring:

- It requires a significant change in the authoring process. Writing structured documents is very different from writing with a typical word processor, and change is difficult. In particular, authors don't like giving up control over the appearance of their words especially now that they have acquired it with the advent of word processors. But many publishing companies need authors to relinquish that control, because book design and production remains their job, not their authors'.
- Because semantics are separate from appearance, in order to publish an SGML/XML document, a stylesheet or other tool must create the presentational form from the structural form. Writing stylesheets is a skill in its own right, and though not every author among a group of authors has to learn how to write them, someone has to.
- Authoring tools for SGML documents can generally be pretty expensive. While it's not entirely unreasonable to edit SGML/XML documents with a simple text editor, it's a bit tedious to do so. However, there are a few free tools that are SGML-aware. The widespread interest in XML may well produce new, clever, and less expensive XML editing tools.

Elements and Attributes

SGML/XML markup consists primarily of elements, attributes, and entities. Elements are the terms we have been speaking about most, like `sect1`, that describe a document's content and structure. Most elements are represented by pairs of tags and mark the start and end of the construct they surround—for example, the SGML source for this particular paragraph begins with a `<para>` tag and ends with a `</para>` tag. Some elements are “empty” (such as DocBook's cross-reference element, `<xref>`) and require no end tag.[3]

Elements can, but don't necessarily, include one or more attributes, which are additional terms that extend the function or refine the content of a given element. For instance, in DocBook a `<sect1>` start tag can contain an identifier—an `id` attribute—that will ultimately allow the writer to cross-reference it or enable a reader to retrieve it. End tags cannot contain attributes. A `<sect1>` element with an `id` attribute looks like this:

```
<sect1 id="idvalue">
```

In SGML, the catalog of attributes that can occur on an element is predefined. You cannot add arbitrary attribute names to an element. Similarly, the values allowed for each attribute are predefined. In XML, the use of namespaces[i] may allow you to add additional attributes to an element, but as of this writing, there's no way to perform validation on those attributes.

The `id` attribute is one half of a cross reference. An `idref` attribute on another element, for example `<xref linkend=”idvalue” >`, provides the other half. These attributes provide whatever application might process the SGML source with the data needed either to make a hypertext link or to substitute a named and/or numbered cross reference in place of the `< xref>`. Another use for attributes is to specify subclasses of certain elements. For instance, you can subdivide DocBook's `<systemitem>` into URLs and email addresses by making the content of the `role` attribute the distinction between them, as in `<systemitem role="URL">` versus `<systemitem role="emailaddr">`.

---

[3] In XML, this is written as `<xref/>`, as we'll see in the section the section called “Typing an SGML Document”.

[i] http://www.w3.org/TR/REC-xml-names/

Entities

Entities are a fundamental concept in SGML and XML, and can be somewhat daunting at first. They serve a number of related, but slightly different functions, and this makes them a little bit complicated.

In the most general terms, entities allow you to assign a name to some chunk of data, and use that name to refer to that data. The complexity arises because there are two different contexts in which you can use entities (in the DTD and in your documents), two types of entities (parsed and unparsed), and two or three different ways in which the entities can point to the chunk of data that they name.

In the rest of this section, we'll describe each of the commonly encountered entity types. If you find the material in this section confusing, feel free to skip over it now and come back to it later. We'll refer to the different types of entities as the need arises in our discussion of DocBook. Come back to this section when you're looking for more detail.

Entities can be divided into two broad categories, general entities and parameter entities. Parameter entities are most often used in the DTD, not in documents, so we'll describe them last. Before you can use any type of entity, it must be formally declared. This is typically done in the document prologue, as we'll explain in Chapter 2, but we will show you how to declare each of the entities discussed here.

General Entities

In use, general entities are introduced with an ampersand (&) and end with a semicolon (;). Within the category of general entities, there are two types: internal general entities and external general entities.

Internal general entities

With internal entities, you can associate an essentially arbitrary piece of text (which may have other markup, including references to other entities) with a name. You can then include that text by referring to its name. For example, if your document frequently refers to, say, “O'Reilly & Associates,” you might declare it as an entity:

```
<!ENTITY ora "O'Reilly &amp; Associates">
```

Then, instead of typing it out each time, you can insert it as needed in your document with the entity reference `&ora;`, simply to save time. Note that this entity declaration includes another entity reference within it. That's perfectly valid as long as the reference isn't directly or indirectly recursive.

If you find that you use a number of entities across many documents, you can add them directly to the DTD and avoid having to include the declarations in each document. See the discussion of `dbgenent.mod` in Chapter 5.

External general entities

With external entities, you can reference other documents from within your document. If these entities contain document text (SGML or XML), then references to them cause the parser to insert the text of the external file directly into your document (these are called parsed entities). In this way, you can use entities to divide your single, logical document into physically distinct chunks. For example, you might break your document into four chapters and store them in separate files. At the top of your document, you would include entity declarations to reference the four files:

```
<!ENTITY ch01 SYSTEM "ch01.sgm">
<!ENTITY ch02 SYSTEM "ch02.sgm">
<!ENTITY ch03 SYSTEM "ch03.sgm">
```

```
<!ENTITY ch04 SYSTEM "ch04.sgm">
```

Your `Book` now consists simply of references to the entities:

```
<book>
&ch01;
&ch02;
&ch03;
&ch04;
</book>
```

Sometimes it's useful to reference external files that don't contain document text. For example, you might want to reference an external graphic. You can do this with entities by declaring the type of data that's in the entity using a notation (these are called unparsed entities). For example, the following declaration declares the entity `tree` as an encapsulated PostScript image:

```
<!ENTITY tree SYSTEM "tree.eps" NDATA EPS>
```

Entities declared this way cannot be inserted directly into your document. Instead, they must be used as entity attributes to elements:

```
<graphic entityref="tree"></graphic>
```

Conversely, you cannot use entities declared without a notation as the value of an entity attribute.

Special characters

In order for the parser to recognize markup in your document, it must be able to distinguish markup from content. It does this with two special characters: "<," which identifies the beginning of a start or end tag, and "&," which identifies the beginning of an entity reference.[4] If you want these characters to have their literal value, they must be encoded as entity references in your document. The entity reference `&lt;` produces a left angle bracket; `&amp;` produces the ampersand.[5]

If you do not encode each of these as their respective entity references, then an SGML parser or application is likely to interpret them as characters introducing elements or entities (an XML parser will always interpret them this way); consequently, they won't appear as you intended. If you wish to cite text that contains literal ampersands and less-than signs, you need to transform these two characters into entity references before they are included in a DocBook document. The only other alternative is to incorporate text that includes them in your document through some process that avoids the parser.

In SGML, character entities are frequently declared using a third entity category (one that we deliberately chose to overlook), called data entities. In XML, these are declared using numeric character references. Numeric character ref-

---

[4] In XML, these characters are fixed. In SGML, it is possible to change the markup start characters, but we won't consider that case here. If you change the markup start characters, you know what you're doing. While we're on the subject, in SGML, these characters only have their special meaning if they are followed by a name character. It is, in fact, valid in an *SGML* (but not an XML) document to write "O'Reilly & Associates" because the ampersand is not followed by a name character. Don't do this, however.

[5] The sequence of characters that end a marked section (see the section called "Marked sections"), such as]]> must also be encoded with at least one entity reference if it is not being used to end a marked section. For this purpose, you can use the entity reference `&gt;` for the final right angle bracket.

erences resemble entity references, but technically aren't the same. They have the form `&#999;`, in which “999” is the numeric character number.

In XML, the numeric character number is always the Unicode character number. In addition, XML allows hexadecimal numeric character references of the form `&#xhhhh;`. In SGML, the numeric character number is a number from the document character set that's declared in the SGML declaration.

Character entities are also used to give a name to special characters that can't otherwise be typed or are not portable across applications and operating systems. You can then include these characters in your document by refering to their entity name. Instead of using the often obscure and inconsistent key combinations of your particular word processor to type, say, an uppercase letter U with an umlaut (Ü), you type in an entity for it instead. For instance, the entity for an uppercase letter U with an umlaut has been defined as the entity `Uuml`, so you would type in `&Uuml;` to reference it instead of the actual character. The SGML application that eventually processes your document for presentation will match the entity to your platform's handling of special characters in order to render it appropriately.

Parameter Entities

Parameter entities are only recognized in markup declarations (in the DTD, for example). Instead of beginning with an ampersand, they begin with a percent sign. Parameter entities are most frequently used to customize the DTD. For a detailed discussion of this topic, see Chapter 5. Following are some other uses for them.

Marked sections

You might use a parameter entity reference in an SGML document in a marked section. Marking sections is a mechanism for indicating that special processing should apply to a particular block of text. Marked sections are introduced by the special sequence `<![keyword[` and end with `]]>`. In SGML, marked sections can appear in both DTDs and document instances. In XML, they're only allowed in the DTD.[6]

The most common keywords are `INCLUDE`, which indicates that the text in the marked section should be included in the document; `IGNORE`, which indicates that the text in the marked section should be ignored (it completely disappears from the parsed document); and `CDATA`, which indicates that all markup characters within that section should be ignored except for the closing characters `]]>`.

In SGML, these keywords can be parameter entities. For example, you might declare the following parameter entity in your document:

```
<!ENTITY % draft "INCLUDE">
```

Then you could put the sections of the document that are only applicable in a draft within marked sections:

```
<![%draft;[
<para>
This paragraph only appears in the draft version.
</para>
]]>
```

When you're ready to print the final version, simply change the `draft` parameter entity declaration:

---

[6] Actually, CDATA marked sections are allowed in an XML document, but the keyword cannot be a parameter entity, and it must be typed literally. See the examples on this page.

```
<!ENTITY % draft "IGNORE">
```

and publish the document. None of the draft sections will appear.

How Does DocBook Fit In?

DocBook is a very popular set of tags for describing books, articles, and other prose documents, particularly technical documentation. DocBook is defined using the native DTD syntax of SGML and XML. Like HTML, DocBook is an example of a markup language defined in SGML/XML.

A Short DocBook History

DocBook is almost 10 years old. It began in 1991 as a joint project of HaL Computer Systems and O'Reilly. Its popularity grew, and eventually it spawned its own maintainance organization, the Davenport Group. In mid-1998, it became a Technical Committee (TC) of the Organization for the Advancement of Structured Information Standards (OASIS).

The HaL and O'Reilly era

The DocBook DTD was originally designed and implemented by HaL Computer Systems and O'Reilly & Associates around 1991. It was developed primarily to facilitate the exchange of UNIX documentation originally marked up in **troff**. Its design appears to have been based partly on input from SGML interchange projects conducted by the Unix International and Open Software Foundation consortia.

When DocBook V1.1 was published, discussion about its revision and maintenance began in earnest in the Davenport Group, a forum created by O'Reilly for computer documentation producers. Version 1.2 was influenced strongly by Novell and Digital.

In 1994, the Davenport Group became an officially chartered entity responsible for DocBook's maintenance. DocBook V1.2.2 was published simultaneously. The founding sponsors of this incarnation of Davenport include the following people:

- Jon Bosak, Novell
- Dale Dougherty, O'Reilly & Associates
- Ralph Ferris, Fujitsu OSSI
- Dave Hollander, Hewlett-Packard
- Eve Maler, Digital Equipment Corporation
- Murray Maloney, SCO
- Conleth O'Connell, HaL Computer Systems
- Nancy Paisner, Hitachi Computer Products
- Mike Rogers, SunSoft
- Jean Tappan, Unisys

The Davenport era

Under the auspices of the Davenport Group, the DocBook DTD began to widen its scope. It was now being used by a much wider audience, and for new purposes, such as direct authoring with SGML-aware tools, and publishing directly to paper. As the largest users of DocBook, Novell and Sun had a heavy influence on its design.

In order to help users manage change, the new Davenport charter established the following rules for DocBook releases:

- Minor versions ("point releases" such as V2.2) could add to the markup model, but could not change it in a backward-incompatible way. For example, a new kind of list element could be added, but it would not be acceptable for the existing itemized-list model to start requiring two list items inside it instead of only one. Thus, any document conforming to version *n*.0 would also conform to *n*.*m*.
- Major versions (such as V3.0) could both add to the markup model and make backward-incompatible changes. However, the changes would have to be announced in the last major release.
- Major-version introductions must be separated by at least a year.

V3.0 was released in January 1997. After that time, although DocBook's audience continued to grow, many of the Davenport Group stalwarts became involved in the XML effort, and development slowed dramatically. The idea of creating an official XML-compliant version of DocBook was discussed, but not implemented. (For more detailed information about DocBook V3.0 and plans for subsequent versions, see Appendix C.)

The sponsors wanted to close out Davenport in an orderly way to ensure that DocBook users would be supported. It was suggested that OASIS become DocBook's new home. An OASIS DocBook Technical Committee was formed in July, 1998, with Eduardo Gutentag of Sun Microsystems as chair.

The OASIS era

The DocBook Technical Commitee is continuing the work started by the Davenport Group. The transition from Davenport to OASIS has been very smooth, in part because the core design team consists of essentially the same individuals (we all just changed hats).

DocBook V3.1, published in February 1999, was the first OASIS release. It integrated a number of changes that had been "in the wings" for some time.

The committee is undertaking new DocBook development to ensure that the DTD continues to meet the needs of its users.

In February, OASIS made DocBook SGML V4.1 and DocBook XML V4.1.2 official OASIS Specifications[ii].

Development continues, with:

- A V5.0 DTD projected for release sometime in 2001.
- Experimental XML Schema[iii] versions available[iv] in January 2001.
- Experimental RELAX[v] schemas available[vi] in January 2001.

[ii] http://lists.oasis-open.org/archives/members/200102/msg00000.html

[iii] http://www.w3.org/XML/Schema

[iv] http://www.oasis-open.org/docbook/xmlschema/

[v] http://www.xml.gr.jp/relax/

[vi] http://www.oasis-open.org/docbook/relax/

- Experimental TREX[vii] schemas available[viii] in January 2001.

[vii] http://www.thaiopensource.com/trex/

[viii] http://www.oasis-open.org/docbook/trex/

2

Creating DocBook Documents

$Revision: 1.1 $
$Date: 2001/08/02 10:22:22 $

This chapter explains in concrete, practical terms how to make DocBook documents. It's an overview of all the kinds of markup that are possible in DocBook documents. It explains how to create several kinds of DocBook documents: books, sets of books, chapters, articles, and reference manual entries. The idea is to give you enough basic information to actually start writing. The information here is intentionally skeletal; you can find “the details” in the reference section of this book.

Before we can examine DocBook markup, we have to take a look at what an SGML or XML system requires.

Making an SGML Document

SGML requires that your document have a specific prologue. The following sections describe the features of the prologue.

An SGML Declaration

SGML documents begin with an optional SGML Declaration. The declaration can precede the document instance, but generally it is stored in a separate file that is associated with the DTD. The SGML Declaration is a grab bag of SGML defaults. DocBook includes an SGML Declaration that is appropriate for most DocBook documents, so we won't go into a lot of detail here about the SGML Declaration.

In brief, the SGML Declaration describes, among other things, what characters are markup delimiters (the default is angle brackets), what characters can compose tag and attribute names (usually the alphabetical and numeric characters plus the dash and the period), what characters can legally occur within your document, how long SGML “names” and “numbers” can be, what sort of minimizations (abbreviation of markup) are allowed, and so on. Changing the SGML Declaration is rarely necessary, and because many tools only partially support changes to the declaration, changing it is best avoided, if possible.

Wayne Wholer has written an excellent tutorial on the SGML Declaration; if you're interested in more details, see http://www.oasis-open.org/cover/wlw11.html.

A Document Type Declaration

All SGML documents must begin with a document type declaration. This identifies the DTD that will be used by the document and what the root element of the document will be. A typical doctype declaration for a DocBook document looks like this:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
```

This declaration indicates that the root element, which is the first element in the hierarchical structure of the document, will be `<book>` and that the DTD used will be the one identified by the public identifier `-//OASIS//DTD DocBook V3.1//EN`. See the section called “Public Identifiers”” later in this chapter.

An Internal Subset

It's also possible to provide additional declarations in a document by placing them in the document type declaration:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN" [
<!ENTITY nwalsh "Norman Walsh">
<!ENTITY chap1 SYSTEM "chap1.sgm">
<!ENTITY chap2 SYSTEM "chap2.sgm">
]>
```

These declarations form what is known as the internal subset. The declarations stored in the file referenced by the public or system identifier in the `DOCTYPE` declaration is called the external subset and it is technically optional. It is legal to put the DTD in the internal subset and to have no external subset, but for a DTD as large as DocBook that wouldn't make much sense.

Note

The internal subset is parsed *first* and, if multiple declarations for an entity occur, the first declaration is used. Declarations in the internal subset override declarations in the external subset.

The Document (or Root) Element

Although comments and processing instructions may occur between the document type declaration and the root element, the root element usually immediately follows the document type declaration:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN" [
<!ENTITY nwalsh "Norman Walsh">
<!ENTITY chap1 SYSTEM "chap1.sgm">
<!ENTITY chap2 SYSTEM "chap2.sgm">
]>
<book>
&chap1;
&chap2;
</book>
```

You cannot place the root element of the document in an external entity.

Typing an SGML Document

If you are entering SGML using a text editor such as Emacs or vi, there are a few things to keep in mind.[7] Using a structured text editor designed for SGML hides most of these issues.

- DocBook element and attribute names are not case-sensitive. There's no difference between `<Para>` and `<pArA>`. Entity names are case-sensitive, however.

  If you are interested in future XML compatibility, input all element and attribute names strictly in lowercase.

- If attribute values contain spaces or punctuation characters, you must quote them. You are not required to quote attribute values if they consist of a single word or number, although it is not wrong to do so.

---

[7] Many of these things are influenced by the SGML declaration in use. For the purpose of this discussion, we assume you are using the standard DocBook declaration.

When quoting attribute values, you can use either a straight single quote ('), or a straight double quote ("). Don't use the “curly” quotes (“ and ”) in your editing tool.

If you are interested in future XML compatibility, always quote all attribute values.

- Several forms of markup minimization are allowed, including empty tags. Instead of typing the entire end tag for an element, you can type simply `</>`. For example:

```
<para>
This is <emphasis>important</>: never stick the tines of a fork
in an electrical outlet.
</para>
```

  You can use this technique for any and every tag, but it will make your documents very hard to understand and difficult to debug if you introduce errors. It is best to use this technique only for inline elements containing a short string of text.

  Empty start tags are also possible, but may be even more confusing. For the record, if you encounter an empty start tag, the SGML parser uses the element that ended last:

```
<para>
This is <emphasis>important</emphasis>.  So is <>this</para>.
</para>
```

  Both “important” and “this” are emphasized.

  If you are interested in future XML compatibility, don't use any of these tricks.

- The null end tag (net) minimization feature allows constructions like this:

```
<para>
This is <emphasis/important/: never stick the tines of a fork
in an electrical outlet.
</para>
```

  If, instead of ending a start tag with `>`, you end it with a slash, then the next occurrence of a slash ends the element.

  If you are interested in future XML compatibility, don't use net tag minimization either.

If you are willing to modify both the declaration and the DTD, even more dramatic minimizations are possible, including completely omitted tags and “shortcut” markup.

Removing Minimizations

Although we've made a point of reminding you about which of these minimization features are not valid in XML, that's not really a sufficient reason to avoid using them. (The fact that many of the minimization features can lead to confusing, difficult-to-author documents might be.)

> If you want to convert one of these documents to XML at some point in the future, you can run it through a program like **sgmlnorm**, which will remove all the minimizations and insert the correct, verbose markup. The **sgmlnorm** program is part of the SP and Jade distributions[ii], which are on the CD-ROM.

Making an XML Document

In order to create DocBook documents in XML, you'll need an XML version of DocBook. We've included one on the CD, but it hasn't been officially adopted by the OASIS DocBook Technical Committee yet. If you're interested in the technical details, Appendix B, describes the specific differences between SGML and XML versions of DocBook.

XML, like SGML, requires a specific prologue in your document. The following sections describe the features of the XML prologue.

An XML Declaration

XML documents should begin with an XML declaration. Unlike the SGML declaration, which is a grab bag of features, the XML declaration identifies a few simple aspects of the document:

```
<?xml version="1.0" standalone="no"?>
```

Identifying the version of XML ensures that future changes to the XML specification will not alter the semantics of this document. The standalone declaration simply makes explicit the fact that this document cannot “stand alone,” and that it relies on an external DTD. The complete details of the XML declaration are described in the XML specification[iii].

A Document Type Declaration

Strictly speaking, XML documents don't require a DTD. Realistically, DocBook XML documents will have one.

The document type declaration identifies the DTD that will be used by the document and what the root element of the document will be. A typical doctype declaration for a DocBook document looks like this:

```
<?xml version='1.0'?>
<!DOCTYPE book PUBLIC "-//Norman Walsh//DTD DocBk XML V3.1.4//EN"
                         "http://nwalsh.com/docbook/xml/3.1.4/db3xml.dtd">
```

This declaration indicates that the root element will be `<book>` and that the DTD used will be the one indentified by the public identifier `-//Norman Walsh//DTD DocBk XML V3.1.4//EN`. External declarations in XML must include a system identifier (the public identifier is optional). In this example, the DTD is stored on a web server.

System identifiers in XML must be URIs. Many systems may accept filenames and interpret them locally as `file:` URLs, but it's always correct to fully qualify them.

An Internal Subset

It's also possible to provide additional declarations in a document by placing them in the document type declaration:

```
<?xml version='1.0'?>
<!DOCTYPE book PUBLIC "-//Norman Walsh//DTD DocBk XML V3.1.4/EN"
                         "http://nwalsh.com/docbook/xml/3.1.4/db3xml.dtd" [
<!ENTITY nwalsh "Norman Walsh">
<!ENTITY chap1 SYSTEM "chap1.sgm">
```

[ii] http://www.jclark.com/

[iii] http://www.w3.org/TR/REC-xml

```
<!ENTITY chap2 SYSTEM "chap2.sgm">
]>
```

These declarations form what is known as the internal subset. The declarations stored in the file referenced by the public or system identifier in the `DOCTYPE` declaration is called the external subset, which is technically optional. It is legal to put the DTD in the internal subset and to have no external subset, but for a DTD as large as DocBook, that would make very little sense.

Note

The internal subset is parsed *first* in XML and, if multiple declarations for an entity occur, the first declaration is used. Declarations in the internal subset override declarations in the external subset.

The Document (or Root) Element

Although comments and processing instructions may occur between the document type declaration and the root element, the root element usually immediately follows the document type declaration:

```
<?xml version='1.0'?>
<!DOCTYPE book PUBLIC "-//Norman Walsh//DTD DocBk XML V3.1.4//EN"
                         "http://nwalsh.com/docbook/xml/3.1.4/db3xml.dtd" [
<!ENTITY nwalsh "Norman Walsh">
<!ENTITY chap1 SYSTEM "chap1.sgm">
<!ENTITY chap2 SYSTEM "chap2.sgm">
]>
<book>...</book>
```

The important point is that the root element must be physically present immediately after the document type declaration. You cannot place the root element of the document in an external entity.

Typing an XML Document

If you are entering SGML using a text editor such as Emacs or vi, there are a few things to keep in mind. Using a structured text editor designed for XML hides most of these issues.

- In XML, all markup is case-sensitive. In the XML version of DocBook, you must always type all element, attribute, and entity names in lowercase.
- You are required to quote all attribute values in XML.

  When quoting attribute values, you can use either a straight single quote ('), or a straight double quote ("). Don't use the “curly” quotes (“ and ”) in your editing tool.
- Empty elements in XML are marked with a distinctive syntax: `<xref/>`.
- Processing instructions in XML begin and end with a question mark: `<?pitarget data?>`.
- XML was designed to be served, received, and processed over the Web. Two of its most important design principles are ease of implementation and interoperability with both SGML and HTML.

  The markup minimization features in SGML documents make it more difficult to process, and harder to write a parser to interpret it; these minimization features also run counter to the XML design principles named above. As a result, XML does not support them.

> Luckily, a good authoring environment can offer all of the features of markup minimization without interfering with the interoperability of documents. And because XML tools are easier to write, it's likely that good, inexpensive XML authoring environments will be available eventually.

XML and SGML Markup Considerations in This Book

Conceptually, almost everything in this book applies equally to SGML and XML. But because DocBook V3.1 is an SGML DTD, we naturally tend to use SGML conventions in our writing. If you're primarily interested in XML, there are just a few small details to keep in mind.

- XML is case-sensitive, while the SGML version of DocBook is not. In this book, we've chosen to present the element names using mixed case (`Book`, `indexterm`, `XRef`, and so on), but in the DocBook XML DTD, all element, attribute, and entity names are strictly lowercase.
- Empty element start tags in XML are marked with a distinctive syntax: `<xref/>`. In SGML, the trailing slash is not present, so some of our examples need slight revisions to be valid XML elements.
- Processing instructions in XML begin and end with a question mark: `<?pitarget data?>`. In SGML, the trailing question mark is not present, so some of our examples need slight revisions to be valid XML elements.
- Generally we use public identifiers in examples, but whenever system identifiers are used, don't forget that XML system identifiers must be Uniform Resource Indicators (URIs), in which SGML system identifiers are usually simple filenames.

For a more detailed discussion of DocBook and XML, see Appendix B.

Public Identifiers, System Identifiers, and Catalog Files

When a DTD or other external file is referenced from a document, the reference can be specified in three ways: using a public identifier, a system identifier, or both. In XML, the system identifier is *generally* required and the public identifier is optional. In SGML, neither is required, but at least one must be present.[8]

A public identifier is a globally unique, abstract name, such as the following, which is the official public identifier for DocBook V3.1:

```
-//OASIS//DTD DocBook V3.1//EN
```

The introduction of XML has added some small complications to system identifiers. In SGML, a system identifier generally points to a single, local version of a file using local system conventions. In XML, it must point with a Uniform Resource Indicator (URI). The most common URI today is the Uniform Resource Locator (URL), which is familiar to anyone who browses the Web. URLs are a lot like SGML system identifiers, because they generally point to a single version of a file on a particular machine. In the future, Uniform Resource Names (URN), another form of URI, will allow XML system identifiers to have the abstract characteristics of public identifiers.

The following filename is an example of an SGML system identifier:

```
/usr/local/sgml/docbook/3.1/docbook.dtd
```

An equivalent XML system identifier might be:

```
file:///usr/local/sgml/docbook/3.1/docbook.dtd
```

---

[8] This is not absolutely true. SGML allows for the possibility that the reference could be implied by the application, but this is very rarely the case.

The advantage of using the public identifier is that it makes your documents more portable. For any system on which DocBook is installed, the public identifier will resolve to the appropriate local version of the DTD (if public identifiers can be resolved at all).

Public identifiers have two disadvantages:

- Because XML does not require them, and because system identifiers are required, developing XML tools may not provide adequate support for public identifiers. To work with these systems you must use system identifiers.
- Public identifiers aren't magical. They're simply a method of indirection. For them to work, there must be a resolution mechanism for public identifiers. Luckily, several years ago, SGML Open (now OASIS[iv]) described a standard mechanism for mapping public identifiers to system identifers using catalog files.

  See OASIS Technical Resolution 9401:1997 (Amendment 2 to TR 9401).[v]

Public Identifiers

An important characteristic of public identifiers is that they are *globally unique*. Referring to a document with a public identifier should mean that the identifier will resolve to the same actual document on any system even though the location of that document on each system may vary. As a rule, you should never reuse public identifiers, and a published revision should have a new public identifier. Not following these rules defeats one purpose of the public identifier.

A public identifier can be any string of upper- and lowercase letters, digits, any of the following symbols: “'”, “(“, “)”, “+”, “,”, “-”, “.”, “/”, “:”, “=”, “?”, and white space, including line breaks.

Formal public identifiers

Most public identifiers conform to the ISO 8879 standard that defines formal public identifiers. Formal public identifiers, frequently referred to as FPI, have a prescribed format that can ensure uniqueness:[9]

```
prefix//owner-identifier//
text-class text-description//
language//display-version
```

Here are descriptions of the identifiers in this string:

`prefix`

> The *`prefix`* is either a “`+`” or a “`-`” Registered public identifiers begin with “`+`”; unregistered identifiers begin with “`-`”.
>
> (ISO standards sometimes use a third form beginning with `ISO` and the standard number, but this form is only available to ISO.)
>
> The purpose of registration is to guarantee a unique owner-identifier. There are few authorities with the power to issue registered public identifiers, so in practice unregistered identifiers are more common.

[iv] http://www.oasis-open.org/

[v] http://www.oasis-open.org/html/a401.htm

[9] Essentially, it can ensure that two different owners won't accidentally tread on each other. Nothing can prevent a given owner from reusing public identifiers, except maybe common sense.

The Graphics Communication Association[vi] (GCA) can assign registered public identifiers. They do this by issuing the applicant a unique string and declaring the format of the owner identifier. For example, the Davenport Group was issued the string “A00002” and could have published DocBook using an FPI of the following form:

```
+//ISO/IEC 9070/RA::A00002//...
```

Another way to use a registered public identifier is to use the format reserved for internet domain names. For example, O'Reilly can issue documents using an FPI of the following form:

```
+//IDN oreilly.com//...
```

As of DocBook V3.1, the OASIS Technical Committee responsible for DocBook has elected to use the unregistered owner identifier, `OASIS`, thus its prefix is `-`.

```
-//OASIS//...
```

`owner-identifier`

Identifies the person or organization that owns the identifier. Registration guarantees a unique owner identifier. Short of registration, some effort should be made to ensure that the owner identifier is globally unique. A company name, for example, is a reasonable choice as are Internet domain names. It's also not uncommon to see the names of individuals used as the owner-identifier, although clearly this may introduce collisions over time.

The owner-identifier for DocBook V3.1 is `OASIS`. Earlier versions used the owner-identifier `Davenport`.

`text-class`

The text class identifies the kind of document that is associated with this public identifier. Common text classes are

DOCUMENT

An SGML or XML document.

DTD

A DTD or part of a DTD.

ELEMENTS

A collection of element declarations.

ENTITIES

A collection of entity declarations.

NONSGML

Data that is not in SGML or XML.

DocBook is a DTD, thus its text class is DTD.

`text-description`

[vi] http://www.gca.org/

This field provides a description of the document. The text description is free-form, but cannot include the string //.

The text description of DocBook is `DocBook V3.1`.

In the uncommon case of unavailable public texts (FPIs for proprietary DTDs, for example), there are a few other options available (technically in front of or in place of the text description), but they're rarely used. [10]

`language`

Indicates the language in which the document is written. It is recommended that the ISO standard two-letter language codes be used if possible.

DocBook is an English-language DTD, thus its language is `EN`.

`display-version`

This field, which is not frequently used, distinguishes between public texts that are the same except for the display device or system to which they apply.

For example, the FPI for the ISO Latin 1 character set is:

```
-//ISO 8879-1986//ENTITIES Added Latin 1//EN
```

A reasonable FPI for an XML version of this character set is:

```
-//ISO 8879-1986//ENTITIES Added Latin 1//EN//XML
```

System Identifiers

System identifiers are usually filenames on the local system. In SGML, there's no constraint on what they can be. Anything that your SGML processing system recognizes is allowed. In XML, system identifiers must be URIs (Uniform Resource Identifiers).

The use of URIs as system identifiers introduces the possibility that a system identifier can be a URN. This allows the system identifier to benefit from the same global uniqueness benefit as the public identifier. It seems likely that XML system identifiers will eventually move in this direction.

Catalog Files

Catalog files are the standard mechanism for resolving public identifiers into system identifiers. Some resolution mechanism is necessary because DocBook refers to its component modules with public identifiers, and those must be mapped to actual files on the system before any piece of software can actually load them.

The catalog file format was defined in 1994 by SGML Open (now OASIS). The formal specification is contained in OASIS Technical Resolution 9401:1997.

Informally, a catalog is a text file that contains a number of keyword/value pairs. The most frequently used keywords are `PUBLIC`, `SYSTEM`, `SGMLDECL`, `DTDDECL`, `CATALOG`, `OVERRIDE`, `DELEGATE`, and `DOCTYPE`.

`PUBLIC`

The `PUBLIC` keyword maps public identifiers to system identifiers:

---

[10] See Appendix A of [maler96], for more details.

```
PUBLIC "-//OASIS//DTD DocBook V3.1//EN" "docbook/3.1/docbook.dtd"
```

`SYSTEM`

The `SYSTEM` keyword maps system identifiers to system identifiers:

```
SYSTEM "http://nwalsh.com/docbook/xml/1.3/db3xml.dtd"
    "docbook/xml/1.3/db3xml.dtd"
```

`SGMLDECL`

The `SGMLDECL` keyword identifies the system identifier of the SGML Declaration that should be used:

```
SGMLDECL "docbook/3.1/docbook.dcl"
```

`DTDDECL`

Like `SGMLDECL`, `DTDDECL` identifies the SGML Declaration that should be used. `DTDDECL` associates a declaration with a particular public identifier for a DTD:

```
DTDDECL "-//OASIS//DTD DocBook V3.1//EN" "docbook/3.1/docbook.dcl"
```

Unfortunately, it is not supported by the free tools that are available. The practical benefit of `DTDDECL` can usually be achieved, albeit in a slightly cumbersome way, with multiple catalog files.

`CATALOG`

The `CATALOG` keyword allows one catalog to include the content of another. This can make maintenance somewhat easier and allows a system to directly use the catalog files included in DTD distributions. For example, the DocBook distribution includes a catalog file. Rather than copying each of the declarations in that catalog into your system catalog, you can simply include the contents of the DocBook catalog:

```
CATALOG "docbook/3.1/catalog"
```

`OVERRIDE`

The `OVERRIDE` keyword indicates whether or not public identifiers override system identifiers. If a given declaration includes both a system identifer and a public identifier, most systems attempt to process the document referenced by the system identifier, and consequently ignore the public identifier. Specifying

```
OVERRIDE YES
```

in the catalog informs the processing system that resolution should be attempted first with the public identifier.

`DELEGATE`

The `DELEGATE` keyword allows you to specify that some set of public identifiers should be resolved by another catalog. Unlike the `CATALOG` keyword, which loads the referenced catalog, `DELEGATE` does nothing until an attempt is made to resolve a public identifier.

The `DELEGATE` entry specifies a partial public identifier and an alternate catalog:

```
DELEGATE "-//OASIS" "/usr/sgml/oasis/catalog"
```

Partial public identifers are simply initial substring matches. Given the preceding entry, if an attempt is made to match any public identifier that begins with the string `-//OASIS`, the alternate catalog `/usr/sgml/oasis/catalog` will be used instead of the current catalog.

`DOCTYPE`

The `DOCTYPE` keyword allows you to specify a default system identifier. If an SGML document begins with a `DOCTYPE` declaration that specifies neither a public identifier nor a system identifier (or is missing a `DOCTYPE` declaration altogether), the `DOCTYPE` declaration may provide a default:

```
DOCTYPE BOOK n:/share/sgml/docbook/3.1/docbook.dtd
```

A small fragment of an actual catalog file is shown in Example 2.1.

Example 2.1. A Sample Catalog

```
                                                          ①
-- Comments are delimited by pairs of double-hyphens,
   as in SGML and XML comments. --
                                                          ②
OVERRIDE YES
                                                          ③
SGMLDECL "n:/share/sgml/docbook/3.1/docbook.dcl"
                                                          ④
DOCTYPE  BOOK  n:/share/sgml/docbook/3.1/docbook.dtd
                                                          ⑤
PUBLIC "-//OASIS//DTD DocBook V3.1//EN"
  n:/share/sgml/docbook/3.1/docbook.dtd
                                                          ⑥
SYSTEM "http://nwalsh.com/docbook/xml/1.3/db3xml.dtd"
  n:/share/sgml/Norman_Walsh/db3xml/db3xml.dtd
```

① Catalog files may also include comments.

② This catalog specifies that public identifiers should be used in favor of system identifiers, if both are present.

③ The default declaration specified by this catalog is the DocBook declaration.

④ Given an explicit (or implied) SGML `DOCTYPE` of

```
<!DOCTYPE BOOK SYSTEM>
```

use `n:/share/sgml/docbook/3.1/docbook.dtd` as the default system identifier. Note that this can only apply to SGML documents because the DOCTYPE declaration above is not a valid XML element.

⑤ Map the OASIS public identifer to the local copy of the DocBook V3.1 DTD.

❻ Map a system identifer for the XML version of DocBook to a local version.

A few notes:

- It's not uncommon to have several catalog files. See below, the section called “Locating catalog files””.
- Like attributes on elements you can quote, the public identifier and system identifier are surrounded by either single or double quotes.
- White space in the catalog file is generally irrelevant. You can use spaces, tabs, or new lines between keywords and their arguments.
- When a relative system identifier is used, it is considered to be relative to the location of the catalog file, not the document being processed.

Locating catalog files

Catalog files go a long way towards making documents more portable by introducing a level of indirection. A problem still remains, however: how does a processor locate the appropriate catalog file(s)? OASIS outlines a complete interchange packaging scheme, but for most applications the answer is simply that the processor looks for a file called `catalog` or `CATALOG`.

Some applications allow you to specify a list of directories that should be examined for catalog files. Other tools allow you to specify the actual files.

Note that even if a list of directories or catalog files is provided, applications may still load catalog files that occur in directories in which other documents are found. For example, SP and Jade always load the catalog file that occurs in the directory in which a DTD or document resides, even if that directory is not on the catalog file list.

Physical Divisions: Breaking a Document into Physical Chunks

The rest of this chapter describes how you can break documents into logical chunks, such as books, chapters, sections, and so on. Before we begin, and while the subject of the internal subset is fresh in your mind, let's take a quick look at how to break documents into separate physical chunks.

Actually, we've already told you how to do it. If you recall, in the preceding sections we had declarations of the form:

```
<!ENTITY name SYSTEM "filename">
```

If you refer to the entity *`name`* in your document after this declaration, the system will insert the contents of the file *`filename`* into your document at that point. So, if you've got a book that consists of three chapters and two appendixes, you might create a file called `book.sgm`, which looks like this:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN" [
<!ENTITY chap1 SYSTEM "chap1.sgm">
<!ENTITY chap2 SYSTEM "chap2.sgm">
<!ENTITY chap3 SYSTEM "chap3.sgm">
<!ENTITY appa SYSTEM "appa.sgm">
<!ENTITY appb SYSTEM "appb.sgm">
]>
<book><title>My First Book</title>
&chap1;
&chap2;
&chap3;
&appa;
```

```
&appb;
</book>
```

You can then write the chapters and appendixes conveniently in separate files. Note that these files do not and must not have document type declarations.

For example, Chapter 1 might begin like this:

```
<chapter id="ch1"><title>My First Chapter</title>
<para>My first paragraph.</para>
...
```

But it should not begin with its own document type declaration:

```
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<chapter id="ch1"><title>My First Chapter</title>
<para>My first paragraph.</para>
...
```

Logical Divisions: The Categories of Elements in DocBook

DocBook elements can be divided broadly into these categories:
Sets
Books
Divisions, which divide books into parts
Components, which divide books or divisions into chapters
Sections, which subdivide components
Meta-information elements
Block elements
Inline elements

In the rest of this section, we'll describe briefly the elements that make up these categories. This section is designed to give you an overview. It is not an exhaustive list of every element in DocBook.

For more information about any specific element and the elements that it may contain, consult the reference page for the element in question.

Sets

A `Set` contains two or more `Book`s. It's the hierarchical top of DocBook. You use the `Set` tag, for example, for a series of books on a single subject that you want to access and maintain as a single unit, such as the manuals for an airplane engine or the documentation for a programming language.

Books

A `Book` is probably the most common top-level element in a document. The DocBook definition of a book is very loose and general. Given the variety of books authored with DocBook and the number of different conventions for book organization used in countries around the world, attempting to impose a strict ordering of elements can make the content model extremely complex. But DocBook gives you free reign. It's very reasonable to use a local customization layer to impose a more strict ordering for your applications.

`Book`s consist of a mixture of the following elements:

Dedication

> `Dedication` pages almost always occur at the front of a book.

Navigational Components

> There are a few component-level elements designed for navigation: `ToC`, for Tables of Contents; `LoT`, for Lists of Titles (for lists of figures, tables, examples, and so on); and `Index`, for indexes.

Divisions

> Divisions are the first hierarchical level below `Book`. They contain `Part`s and `Reference`s. `Part`s, in turn, contain components. `Reference`s contain `RefEntry`s. These are discussed more thoroughly in the section called “Making a Reference Page””.
>
> Books can contain components directly and are not required to contain divisions.

Components

> These are the chapter-like elements of a `Book`.

Components

Components are the chapter-like elements of a `Book` or `Part`: `Preface`, `Chapter`, `Appendix`, `Glossary`, and `Bibliography`. `Article`s can also occur at the component level. We describe `Article`s in more detail in the section titled the section called “Making an Article””. Components generally contain block elements and/or sections, and some can contain navigational components and `RefEntry`s.

Sections

There are several flavors of sectioning elements in DocBook:

`Sect1`…`Sect5` elements

> The `Sect1`…`Sect5` elements are the most common sectioning elements. They can occur in most component-level elements. These numbered section elements must be properly nested (`Sect2`s can only occur inside `Sect1`s, `Sect3`s can only occur inside `Sect2`s, and so on). There are five levels of numbered sections.

`Section` element

> The `Section` element, introduced in DocBook V3.1, is an alternative to numbered sections. `Section`s are recursive, meaning that you can nest them to any depth desired.

`SimpleSect` element

> In addition to numbered sections, there's the `SimpleSect` element. It is a terminal section that can occur at any level, but it cannot have any other sectioning element nested within it.

`BridgeHead`

A `BridgeHead` provides a section title without any containing section.

`RefSect1`…`RefSect3` elements

These elements, which occur only in `RefEntry`s, are analogous to the numbered section elements in components. There are only three levels of numbered section elements in a `RefEntry`.

`GlossDiv`, `BiblioDiv`, and `IndexDiv`

`Glossary`s, `Bibliography`s, and `Index`es can be broken into top-level divisions, but not sections. Unlike sections, these elements do not nest.

Meta-Information

All of the elements at the section level and above include a wrapper for meta-information about the content. See, for example, `BookInfo`.

The meta-information wrapper is designed to contain bibliographic information about the content (`Author`, `Title`, `Publisher`, and so on) as well as other meta-information such as revision histories, keyword sets, and index terms.

Block Elements

The block elements occur immediately below the component and sectioning elements. These are the (roughly) paragraph-level elements in DocBook. They can be divided into a number of categories: lists, admonitions, line-specific environments, synopses of several sorts, tables, figures, examples, and a dozen or more miscellaneous elements.

Block vs. Inline Elements

At the paragraph-level, it's convenient to divide elements into two classes, block and inline. From a structural point of view, this distinction is based loosely on their relative size, but it's easiest to describe the difference in terms of their presentation.

Block elements are usually presented with a paragraph (or larger) break before and after them. Most can contain other block elements, and many can contain character data and inline elements. Paragraphs, lists, sidebars, tables, and block quotations are all common examples of block elements.

Inline elements are generally represented without any obvious breaks. The most common distinguishing mark of inline elements is a font change, but inline elements may present no visual distinction at all. Inline elements contain character data and possibly other inline elements, but they never contain block elements. Inline elements are used to mark up data such as cross references, filenames, commands, options, subscripts and superscripts, and glossary terms.

Lists

There are seven list elements in DocBook:

`CalloutList`

A list of `CallOut`s and their descriptions. `CallOut`s are marks, frequently numbered and typically on a graphic or verbatim environment, that are described in a `CalloutList`, outside the element in which they occur.

`GlossList`

A list of glossary terms and their definitions.

`ItemizedList`

An unordered (bulleted) list. There are attributes to control the marks used.

`OrderedList`

A numbered list. There are attributes to control the type of enumeration.

`SegmentedList`

A repeating set of named items. For example, a list of states and their capitals might be represented as a `SegmentedList`.

`SimpleList`

An unadorned list of items. `SimpleList`s can be inline or arranged in columns.

`VariableList`

A list of terms and definitions or descriptions. (This list of list types is a `VariableList`.)

Admonitions

There are five types of admonitions in DocBook: `Caution`, `Important`, `Note`, `Tip`, and `Warning`.

All of the admonitions have the same structure: an optional `Title` followed by paragraph-level elements. The DocBook DTD does not impose any specific semantics on the individual admonitions. For example, DocBook does not mandate that `Warning`s be reserved for cases where bodily harm can result.

Line-specific environments

These environments preserve whitespace and line breaks in the source text. DocBook does not provide the equivalent of HTML's `BR` tag, so there's no way to interject a line break into normal running text.

`Address`

The `Address` element is intended for postal addresses. In addition to being line-specific, `Address` contains additional elements suitable for marking up names and addresses.

`LiteralLayout`

A `LiteralLayout` does not have any semantic association beyond the preservation of whitespace and line breaks. In particular, while `ProgramListing` and `Screen` are frequently presented in a fixed-width font, a change of fonts is not necessarily implied by `LiteralLayout`.

`ProgramListing`

A `ProgramListing` is a verbatim environment, usually presented in Courier or some other fixed-width font, for program sources, code fragments, and similar listings.

`Screen`

A `Screen` is a verbatim or literal environment for text screen-captures, other fragments of an ASCII display, and similar things. `Screen` is also a frequent catch-all for any verbatim text.

`ScreenShot`

`ScreenShot` is actually a wrapper for a `Graphic` intended for screen shots of a GUI for example.

`Synopsis`

A `Synopsis` is a verbatim environment for command and function synopsis.

Examples, figures, and tables

Examples, Figures, and Tables are common block-level elements: `Example`, `InformalExample`, `Figure`, `InformalFigure`, `Table`, and `InformalTable`.

The distinction between formal and informal elements is that formal elements have titles while informal ones do not. The `InformalFigure` element was introduced in DocBook V3.1. In prior versions of DocBook, you could only achieve the effect of an informal figure by placing its content, unwrapped, at the location where the informal figure was desired.

Paragraphs

There are three paragraph elements: `Para`, `SimPara` (simple paragraphs may not contain other block-level elements), and `FormalPara` (formal paragraphs have titles).

Equations

There are two block-equation elements, `Equation` and `InformalEquation` (for inline equations, use `InlineEquation`).

Informal equations don't have titles. For reasons of backward-compatibility, `Equation`s are not required to have titles. However, it may be more difficult for some stylesheet languages to properly enumerate `Equation`s if they lack titles.

Graphics

Graphics occur most frequently in `Figure`s and `ScreenShot`s, but they can also occur without a wrapper. DocBook considers a `Graphic` a block element, even if it appears to occur inline. For graphics that you want to be represented inline, use `InlineGraphic`.

DocBook V3.1 introduced a new element to contain graphics and other media types: `MediaObject` and its inline cousin, `InlineMediaObject`. These elements may contain video, audio, image, and text data. A single media object can contain several alternative forms from which the presentation system can select the most appropriate object.

Questions and answers

DocBook V3.1 introduced the `QandASet` element, which is suitable for FAQs (Frequently Asked Questions) and other similar collections of `Question`s and `Answer`s.

Miscellaneous block elements

The following block elements are also available:

`BlockQuote`

A block quotation. Block quotations may have `Attribution`s.

`CmdSynopsis`

An environment for marking up all the parameters and options of a command.

`Epigraph`

A short introduction, typically a quotation, at the beginning of a document. `Epigraph`s may have `Attribution`s.

`FuncSynopsis`

An environment for marking up the return value and arguments of a function.

`Highlights`

A summary of the main points discussed in a book component (chapter, section, and so on).

`MsgSet`

A set of related error messages.

`Procedure`

A procedure. Procedures contain `Step`s, which may contain `SubStep`s.

`Sidebar`

A sidebar.

Inline Elements

Users of DocBook are provided with a surfeit of inline elements. Inline elements are used to mark up running text. In published documents, inline elements often cause a font change or other small change, but they do not cause line or paragraph breaks.

In practice, writers generally settle on the tagging of inline elements that suits their time and subject matter. This may be a large number of elements or only a handful. What is important is that you choose to mark up not every possible item, but only those for which distinctive tagging will be useful in the production of the finished document for the readers who will search through it.

The following comprehensive list may be a useful tool for the process of narrowing down the elements that you will choose to mark up; it is not intended to overwhelm you by its sheer length. For convenience, we've divided the inlines into several subcategories.

The classification used here is not meant to be authoritative, only helpful in providing a feel for the nature of the inlines. Several elements appear in more than one category, and arguments could be made to support the placement of additional elements in other categories or entirely new categories.

Traditional publishing inlines

These inlines identify things that commonly occur in general writing:

`Abbrev`

An abbreviation, especially one followed by a period.

`Acronym`

An often pronounceable word made from the initial (or selected) letters of a name or phrase.

`Emphasis`

Emphasized text.

`Footnote`

A footnote. The location of the `Footnote` element identifies the location of the first reference to the footnote. Additional references to the same footnote can be inserted with `FootnoteRef`.

`Phrase`

A span of text.

`Quote`

An inline quotation.

`Trademark`

A trademark.

Cross references

The cross reference inlines identify both explicit cross references, such as `Link`, and implicit cross references like `GlossTerm`. You can make the most of the implicit references explicit with a `LinkEnd` attribute.

`Anchor`

A spot in the document.

`Citation`

An inline bibliographic reference to another published work.

`CiteRefEntry`

A citation to a reference page.

`CiteTitle`

The title of a cited work.

`FirstTerm`

The first occurrence of a term.

`GlossTerm`

A glossary term.

`Link`

A hypertext link.

`OLink`

A link that addresses its target indirectly, through an entity.

`ULink`

A link that addresses its target by means of a URL (Uniform Resource Locator).

`XRef`

A cross reference to another part of the document.

Markup

These inlines are used to mark up text for special presentation:

`ForeignPhrase`

A word or phrase in a language other than the primary language of the document.

`WordAsWord`

A word meant specifically as a word and not representing anything else.

`ComputerOutput`

Data, generally text, displayed or presented by a computer.

`Literal`

Inline text that is some literal value.

`Markup`

A string of formatting markup in text that is to be represented literally.

`Prompt`

A character or string indicating the start of an input field in a computer display.

`Replaceable`

Content that may or must be replaced by the user.

`SGMLTag`

A component of SGML markup.

`UserInput`

Data entered by the user.

Mathematics

DocBook does not define a complete set of elements for representing equations. No one has ever pressed the DocBook maintainers to add this functionality, and the prevailing opinion is that incorporating MathML[vii] using a mechanism like namespaces[viii] is probably the best long-term solution.

`InlineEquation`

A mathematical equation or expression occurring inline.

`Subscript`

A subscript (as in H_2O, the molecular formula for water).

`Superscript`

A superscript (as in x^2, the mathematical notation for x multiplied by itself).

User interfaces

These elements describe aspects of a user interface:

`Accel`

A graphical user interface (GUI) keyboard shortcut.

`GUIButton`

The text on a button in a GUI.

`GUIIcon`

Graphic and/or text appearing as a icon in a GUI.

`GUILabel`

The text of a label in a GUI.

`GUIMenu`

The name of a menu in a GUI.

`GUIMenuItem`

The name of a terminal menu item in a GUI.

`GUISubmenu`

The name of a submenu in a GUI.

`KeyCap`

The text printed on a key on a keyboard.

[vii] http://www.w3.org/TR/REC-MathML/

[viii] http://www.w3.org/TR/REC-xml-names/

`KeyCode`

The internal, frequently numeric, identifier for a key on a keyboard.

`KeyCombo`

A combination of input actions.

`KeySym`

The symbolic name of a key on a keyboard.

`MenuChoice`

A selection or series of selections from a menu.

`MouseButton`

The conventional name of a mouse button.

`Shortcut`

A key combination for an action that is also accessible through a menu.

Programming languages and constructs

Many of the technical inlines in DocBook are related to programming.

`Action`

A response to a user event.

`ClassName`

The name of a class, in the object-oriented programming sense.

`Constant`

A programming or system constant.

`ErrorCode`

An error code.

`ErrorName`

An error name.

`ErrorType`

The classification of an error message.

`Function`

The name of a function or subroutine, as in a programming language.

`Interface`

An element of a GUI.

`InterfaceDefinition`

The name of a formal specification of a GUI.

`Literal`

Inline text that is some literal value.

`MsgText`

The actual text of a message component in a message set.

`Parameter`

A value or a symbolic reference to a value.

`Property`

A unit of data associated with some part of a computer system.

`Replaceable`

Content that may or must be replaced by the user.

`ReturnValue`

The value returned by a function.

`StructField`

A field in a structure (in the programming language sense).

`StructName`

The name of a structure (in the programming language sense).

`Symbol`

A name that is replaced by a value before processing.

`Token`

A unit of information.

`Type`

The classification of a value.

`VarName`

The name of a variable.

Operating systems

These inlines identify parts of an operating system, or an operating environment:

`Application`

The name of a software program.

`Command`

The name of an executable program or other software command.

`EnVar`

A software environment variable.

`Filename`

The name of a file.

`MediaLabel`

A name that identifies the physical medium on which some information resides.

`MsgText`

The actual text of a message component in a message set.

`Option`

An option for a software command.

`Parameter`

A value or a symbolic reference to a value.

`Prompt`

A character or string indicating the start of an input field in a computer display.

`SystemItem`

A system-related item or term.

General purpose

There are also a number of general-purpose technical inlines.

`Application`

The name of a software program.

`Database`

The name of a database, or part of a database.

`Email`

An email address.

`Filename`

The name of a file.

`Hardware`

A physical part of a computer system.

`InlineGraphic`

An object containing or pointing to graphical data that will be rendered inline.

`Literal`

Inline text that is some literal value.

`MediaLabel`

A name that identifies the physical medium on which some information resides.

`Option`

An option for a software command.

`Optional`

Optional information.

`Replaceable`

Content that may or must be replaced by the user.

`Symbol`

A name that is replaced by a value before processing.

`Token`

A unit of information.

`Type`

The classification of a value.

Making a DocBook Book

A typical `Book`, in English at least, consists of some meta-information in a `BookInfo` (`Title`, `Author`, `Copyright`, and so on), one or more `Preface`s, several `Chapter`s, and perhaps a few `Appendix`es. A `Book` may also contain `Bibliography`s, `Glossary`s, `Index`es and a `Colophon`.

Example 2.2 shows the structure of a typical book. Additional content is required where the ellipses occur.

Example 2.2. A Typical Book

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<book>
<bookinfo>
  <title>My First Book</title>
  <author><firstname>Jane</firstname><surname>Doe</surname></author>
  <copyright><year>1998</year><holder>Jane Doe</holder></copyright>
</bookinfo>
<preface><title>Foreword</title> ... </preface>
<chapter> ... </chapter>
<chapter> ... </chapter>
<chapter> ... </chapter>
<appendix> ... </appendix>
<appendix> ... </appendix>
<index> ... </index>
</book>
```

Making a Chapter

`Chapter`s, `Preface`s, and `Appendix`es all have a similar structure. They consist of a `Title`, possibly some additional meta-information, and any number of block-level elements followed by any number of top-level sections. Each section may in turn contain any number of block-level elements followed by any number from the next section level, as shown in Example 2.3.

Example 2.3. A Typical Chapter

```
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<chapter><title>My Chapter</title>
<para> ... </para>
<sect1><title>First Section</title>
<para> ... </para>
<example> ... </example>
</sect1>
</chapter>
```

Making an Article

For documents smaller than a book, such as: journal articles, white papers, or technical notes, `Article` is frequently the most logical starting point. The body of an `Article` is essentially the same as the body of a `Chapter` or any other component-level element, as shown in Example 2.4

`Article`s may include `Appendix`es, `Bibliography`s, `Index`es and `Glossary`s.

Example 2.4. A Typical Article

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<article>
<artheader>
  <title>My Article</title>
  <author><honorific>Dr</honorific><firstname>Emilio</firstname>
          <surname>Lizardo</surname></author>
</artheader>
<para> ... </para>
<sect1><title>On the Possibility of Going Home</title>
<para> ... </para>
</sect1>
<bibliography> ... </bibliography>
</article>
```

Making a Reference Page

The reference page or manual page in DocBook was inspired by, and in fact designed to reproduce, the common UNIX “manpage” concept. (We use the word "page" loosely here to mean a document of variable length containing reference material on a specific topic.) DocBook is rich in markup tailored for such documents, which often vary greatly in content, however well-structured they may be. To reflect both the structure and the variability of such texts, DocBook specifies that reference pages have a strict sequence of parts, even though several of them are actually optional.

Of the following sequence of elements that may appear in a `RefEntry`, only two are obligatory: `RefNameDiv` and `RefSect1`.

`DocInfo`

The `DocInfo` element contains meta-information about the reference page (which should not be confused with `RefMeta`, which it precedes). It marks up information about the author of the document, or the product to which it pertains, or the document's revision history, or other such information.

`RefMeta`

`RefMeta` contains a title for the reference page (which may be inferred if the `RefMeta` element is not present) and an indication of the volume number in which this reference page occurs. The `ManVolNum` is a very UNIX-centric concept. In traditional UNIX documentation, the subject of a reference page is typically identified by name and volume number; this allows you to distinguish between the **uname** command, “uname(1)” in volume 1 of the documentation and the `uname` function, “uname(3)” in volume 3.

Additional information of this sort such as conformance or vendor information specific to the particular environment you are working in, may be stored in `RefMiscInfo`.

`RefNameDiv`

The first obligatory element is `RefNameDiv`, which is a wrapper for information about whatever you're documenting, rather than the document itself. It can begin with a `RefDescriptor` if several items are being documented as a group and the group has a name. The `RefNameDiv` must contain at least one `RefName`, that is, the name of whatever you're documenting, and a single short statement that sums up the use or function of the item(s) at a glance: their `RefPurpose`. Also available is the `RefClass`, intended to detail the operating system configurations that the software element in question supports.

If no `RefEntryTitle` is given in the `RefMeta`, the title of the reference page is the `RefDescriptor`, if present, or the first `RefName`.

`RefSynopsisDiv`

A `RefSynopsisDiv` is intended to provide a quick synopsis of the topic covered by the reference page. For commands, this is generally a syntax summary of the command, and for functions, the function prototype, but other options are possible. A `Title` is allowed, but not required, presumably because the application that processes reference pages will generate the appropriate title if it is not given. In traditional UNIX documentation, its title is always “Synopsis”.

`RefSect1`…`RefSect3`

Within `RefEntry`s, there are only three levels of sectioning elements: `RefSect1`, `RefSect2`, and `RefSect3`.

Example 2.5 shows the beginning of a `RefEntry` that illustrates one possible reference page:

Example 2.5. A Sample Reference Page

```
<refentry id="printf">

<refmeta>
<refentrytitle>printf</refentrytitle>
<manvolnum>3S</manvolnum>
</refmeta>

<refnamediv>
<refname>printf</refname>
<refname>fprintf</refname>
<refname>sprintf</refname>
<refpurpose>print formatted output</refpurpose>
</refnamediv>

<refsynopsisdiv>

<funcsynopsis>
<funcsynopsisinfo>
#include &lt;stdio.h&gt;
</funcsynopsisinfo>
<funcprototype>
  <funcdef>int <function>printf</function></funcdef>
  <paramdef>const char *<parameter>format</parameter></paramdef>
  <paramdef>...</paramdef>
</funcprototype>

<funcprototype>
  <funcdef>int <function>fprintf</function></funcdef>
  <paramdef>FILE *<parameter>strm</parameter></paramdef>
  <paramdef>const char *<parameter>format</parameter></paramdef>
  <paramdef>...</paramdef>
</funcprototype>

<funcprototype>
  <funcdef>int <function>sprintf</function></funcdef>
  <paramdef>char *<parameter>s</parameter></paramdef>
  <paramdef>const char *<parameter>format</parameter></paramdef>
  <paramdef>...</paramdef>
</funcprototype>
</funcsynopsis>

</refsynopsisdiv>

<refsect1><title>Description</title>
<para>
<indexterm><primary>functions</primary>
  <secondary>printf</secondary></indexterm>
<indexterm><primary>printing function</primary></indexterm>

<function>printf</function> places output on the standard
output stream stdout.
&hellip;
</para>
```

Making Front- and Backmatter

DocBook contains markup for the usual variety of front- and backmatter necessary for books and articles: indexes, glossaries, bibliographies, and tables of contents. In many cases, these components are generated automatically, at least in part, from your document by an external processor, but you can create them by hand, and in either case, store them in DocBook.

Some forms of backmatter, like indexes and glossaries, usually require additional markup *in the document* to make generation by an application possible. Bibliographies are usually composed by hand like the rest of your text, unless you are automatically selecting bibliographic entries out of some larger database. Our principal concern here is to acquaint you with the kind of markup you need to include in your documents if you want to construct these components.

Frontmatter, like the table of contents, is almost always generated automatically from the text of a document by the processing application. If you need information about how to mark up a table of contents in DocBook, please consult the reference page for `ToC`.

Making an Index

In some highly-structured documents such as reference manuals, you can automate the whole process of generating an index successfully without altering or adding to the original source. You can design a processing application to select the information and compile it into an adequate index. But this is rare.

In most cases—and even in the case of some reference manuals—a useful index still requires human intervention to mark occurrences of words or concepts that will appear in the text of the index.

Marking index terms

Docbook distinguishes two kinds of index markers: those that are singular and result in a single page entry in the index itself, and those that are multiple and refer to a range of pages.

You put a singular index marker where the subject it refers to actually occurs in your text:

```
<para>
The tiger<indexterm>
<primary>Big Cats</primary>
<secondary>Tigers</secondary></indexterm>
is a very large cat indeed.
</para>
```

This index term has two levels, `primary` and `secondary`. They correspond to an increasing amount of indented text in the resultant index. DocBook allows for three levels of index terms, with the third labeled `tertiary`.

There are two ways that you can index a range of text. The first is to put index marks at both the beginning and end of the discussion. The mark at the beginning asserts that it is the start of a range, and the mark at the end refers back to the beginning. In this way, the processing application can determine what range of text is indexed. Here's the previous tiger example recast as starting and ending index terms:

```
<para>
The tiger<indexterm id="tiger-desc" class="startofrange">
<primary>Big Cats</primary>
<secondary>Tigers</secondary></indexterm>
is a very large cat indeed…
</para>
```

```
<para>
So much for tigers<indexterm startref="tiger-desc" class="endofrange">. Let's talk about
leopards.
</para>
```

Note that the mark at the start of the range identifies itself as the start of a range with the `Class` attribute, and provides an `ID`. The mark at the end of the range points back to the start.

Another way to mark up a range of text is to specify that the entire content of an element, such as a chapter or section, is the complete range. In this case, all you need is for the index term to point to the `ID` of the element that contains the content in question. The `Zone` attribute of `indexterm` provides this functionality.

One of the interesting features of this method is that the actual index marks do not have to occur anywhere near the text being indexed. It is possible to collect all of them together, for example, in one file, but it is not invalid to have the index marker occur near the element it indexes.

Suppose the discussion of tigers in your document comprises a whole text object (like a `Sect1` or a `Chapter`) with an `ID` value of `tiger-desc`. You can put the following tag anywhere in your document to index that range of text:

```
<indexterm zone="tiger-desc">
<primary>Big Cats</primary>
<secondary>Tigers</secondary></indexterm>
```

DocBook also contains markup for index hits that point to other index hits (of the same type such as "See Cats, big" or "See also Lions"). See the reference pages for `See` and `SeeAlso`.

Printing an index

After you have added the appropriate markup to your document, an external application can use this information to build an index. The resulting index must have information about the page numbers on which the concepts appear. It's usually the document formatter that builds the index. In this case, it may never be instantiated in DocBook.

However, there are applications that can produce an index marked up in DocBook. The following example includes some one- and two-level `IndexEntry` elements (which correspond to the primary and secondary levels in the `indexterm`s themselves) that begin with the letter D:

```
<!DOCTYPE index PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<index><title>Index</title>
<indexdiv><title>D</title>
<indexentry>
  <primaryie>database (bibliographic), 253, 255</primaryie>
     <secondaryie>structure, 255</secondaryie>
     <secondaryie>tools, 259</secondaryie>
</indexentry>
<indexentry>
  <primaryie>dates (language specific), 179</primaryie>
</indexentry>
<indexentry>
  <primaryie>DC fonts, <emphasis>172</emphasis>, 177</primaryie>
     <secondaryie>Math fonts, 177</secondaryie>
</indexentry>
</indexdiv>
</index>
```

Making a Glossary

`Glossary`s, like `Bibliography`s, are often constructed by hand. However, some applications are capable of building a skeletal index from glossary term markup in the document. If all of your terms are defined in some glossary database, it may even be possible to construct the complete glossary automatically.

To enable automatic glossary generation, or simply automatic linking from glossary terms in the text to glossary entries, you must add markup to your documents. In the text, you markup a term for compilation later with the inline `GlossTerm` tag. This tag can have a `LinkEnd` attribute whose value is the ID of the actual entry in the glossary.[11]

For instance, if you have this markup in your document:

```
<glossterm linkend="xml">Extensible Markup Language</glossterm> is a new standard…
```

your glossary might look like this:

```
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<glossary><title>Example Glossary</title>

<glossdiv><title>E</title>

<glossentry id="xml"><glossterm>Extensible Markup Language</glossterm>
  <acronym>XML</acronym>
<glossdef>
  <para>Some reasonable definition here.</para>
  <glossseealso otherterm="sgml">
</glossdef>
</glossentry>

</glossdiv>
```

Note that the `GlossTerm` tag reappears in the glossary to mark up the term and distinguish it from its definition within the `GlossEntry`. The `ID` that the `GlossEntry` referenced in the text is the ID of the `GlossEntry` in the `Glossary` itself. You can use the link between source and glossary to create a link in the online form of your document, as we have done with the online form of the glossary in this book.

Making a Bibliography

There are two ways to set up a bibliography in DocBook: you can have the data *raw* or *cooked*. Here's an example of a raw bibliographical item, wrapped in the `Biblioentry` element:

```
<biblioentry xreflabel="Kites75">
  <authorgroup>
    <author><firstname>Andrea</firstname><surname>Bahadur</surname></author>
    <author><firstname>Mark</><surname>Shwarek</></author>
  </authorgroup>
  <copyright><year>1974</year><year>1975</year>
     <holder>Product Development International Holding N. V.</holder>
     </copyright>
  <isbn>0-88459-021-6</isbn>
  <publisher>
    <publishername>Plenary Publications International, Inc.</publishername>
```

---

[11] Some sophisticated formatters might even be able to establish the link simply by examining the content of the terms and the glossary. In that case, the author is not required to make explicit links.

```
  </publisher>
  <title>Kites</title>
  <subtitle>Ancient Craft to Modern Sport</subtitle>
  <pagenums>988-999</pagenums>
  <seriesinfo>
    <title>The Family Creative Workshop</title>
    <seriesvolnums>1-22</seriesvolnums>
    <editor>
      <firstname>Allen</firstname>
      <othername role=middle>Davenport</othername>
      <surname>Bragdon</surname>
      <contrib>Editor in Chief</contrib>
    </editor>
  </seriesinfo>
</biblioentry>
```

The “raw” data in a `Biblioentry` is comprehensive to a fault—there are enough fields to suit a host of different bibliographical styles, and that is the point. An abundance of data requires processing applications to select, punctuate, order, and format the bibliographical data, and it is unlikely that all the information provided will actually be output.

All the “cooked” data in a `Bibliomixed` entry in a bibliography, on the other hand, is intended to be presented to the reader in the form and sequence in which it is provided. It even includes punctuation between the fields of data:

```
<bibliomixed>
  <bibliomset relation=article>
    <surname>Walsh</surname>, <firstname>Norman</firstname>.
    <title role=article>Introduction to Cascading Style Sheets</title>.
  </bibliomset>
  <bibliomset relation=journal>
    <title>The World Wide Web Journal</title>
    <volumenum>2</volumenum><issuenum>1</issuenum>.
    <publishername>O'Reilly & Associates, Inc.</publishername> and
    <corpname>The World Wide Web Consortium</corpname>.
    <pubdate>Winter, 1996</pubdate></bibliomset>.
</bibliomixed>
```

Clearly, these two ways of marking up bibliographical entries are suited to different circumstances. You should use one or the other for your bibliography, not both. Strictly speaking, mingling the raw and the cooked may be “kosher” as far as the DTD is concerned, but it will almost certainly cause problems for most processing applications.

3

Parsing DocBook Documents

$Revision: 1.2 $
$Date: 2002/03/23 20:57:55 $

A key feature of SGML and XML markup is that you *validate* it. The DocBook DTD is a precise description of valid nesting, the order of elements, and their content. All DocBook documents must conform to this description or they are not DocBook documents (by definition).

A validating parser is a program that can read the DTD and a particular document and determine whether the exact nesting and order of elements in the document is valid according to the DTD.

If you are not using a structured editor that can enforce the markup as you type, validation with an external parser is a particularly important step in the document creation process. You cannot expect to get rational results from subsequent processing (such as document publishing) if your documents are not valid.

The most popular free SGML parser is SP by James Clark, available at http://www.jclark.com/.

SP includes **nsgmls**, a fast command-line parser. In the world of free validating XML parsers, James Clark's **xp** is a popular choice.

Note

Not all XML parsers are validating, and although a non-validating parser may have many uses, it cannot ensure that your documents are valid according to the DTD.

Validating Your Documents

The exact way in which the parser is executed varies according to the parser in use, naturally. For information about your particular parser, consult the documentation that came with it.

Using nsgmls

The **nsgmls** command from SP is a validating SGML parser. The options used in the example below suppress the normal output (`-s`), except for error messages, print the version number (`-v`), and specify the catalog file that should be used to map public identifiers to system identifiers. Printing the version number guarantees that you always get *some* output, so that you know the command ran successfully:

```
[n:\dbtdg] nsgmls -sv -c \share\sgml\catalog test.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
```

Because no error messages were printed, we know our document is valid. If you're working with a document that you discover has many errors, the `-f` option offers a handy way to direct the errors to a file so they don't all scroll off your screen.

If you want to validate an XML document with SP, you must make sure that SP uses the correct declaration. An XML declaration called `xml.dcl` is included with SP.

The easiest way to make sure that SP uses `xml.dcl` is to include the declaration explicitly on the command line when you run **nsgmls** (or Jade, or other SP tools):

```
[n:\dbtdg] nsgmls -sv -c \share\sgml\catalog m:\jade\xml.dcl test.xml
m:\jade\nsgmls.exe:I: SP version "1.3.2"
```

Using xp

The xp distribution includes several sample programs. One of these programs, **Time**, performs a validating parse of the document and prints the amount of time required to parse the DTD and the document. This program makes an excellent validity checker:

```
java com.jclark.xml.apps.Time examples\simple.xml
6.639
```

The result states that it took 6.639 seconds to parse the DTD and the document. This indicates that the document is valid. If the document is invalid, additional error messages are displayed.

Understanding Parse Errors

Every parser produces slightly different error messages, but most indicate exactly (at least technically)[12] what is wrong and where the error occurred. With a little experience, this information is all you'll need to quickly identify what's wrong.

In the rest of this section, we'll look at a number of common errors and the messages they produce in SP. We've chosen SP for the rest of these examples because that is the same parser used by Jade, which we'll be discussing further in Chapter 4.

DTD Cannot Be Found

The telltale sign that SP could not find the DTD, or some module of the DTD, is the error message: "cannot generate system identifier for public text …". Generally, the errors that occur after this are spurious; if SP couldn't find some part of the DTD, it's likely to think that *everything* is wrong.

Careful examination of the following document will show that we've introduced a simple typographic error into the public identifier (the word “DocBook” is misspelled with a lowercase “b”):

```
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD Docbook XML V4.1.2//EN"
                  "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
```

---

[12] It is often the case that you can correct an error in the document in several ways. The parser suggests one possible fix, but this is not always the right fix. For example, the parser may suggest that you can correct out of context data by adding another element, when in fact it's “obvious” to human eyes that the problem is a missing end tag.

```
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para>
<emphasis role="bold">This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

SP responds dramatically to this error:

```
hermes:/documents/books/tdg/examples/errs$ nsgmls -sv -c cat1 /usr/lib/sgml/declaration/xml.dcl nodtd.sgm
nsgmls:I: SP version "1.3.4"
nsgmls:nodtd.sgm:2:76:E: could not resolve host "www.oasis-open.org" (try again later)
nsgmls:nodtd.sgm:2:76:E: DTD did not contain element declaration for document type name
nsgmls:nodtd.sgm:3:8:E: element "chapter" undefined
nsgmls:nodtd.sgm:3:15:E: element "title" undefined
nsgmls:nodtd.sgm:4:5:E: element "para" undefined
nsgmls:nodtd.sgm:10:5:E: element "para" undefined
nsgmls:nodtd.sgm:11:15:E: there is no attribute "role"
nsgmls:nodtd.sgm:11:21:E: element "emphasis" undefined
nsgmls:nodtd.sgm:12:9:E: element "emphasis" undefined
nsgmls:nodtd.sgm:12:24:E: element "emphasis" undefined
nsgmls:nodtd.sgm:13:18:E: element "superscript" undefined
nsgmls:nodtd.sgm:14:16:E: element "subscript" undefined
nsgmls:nodtd.sgm:16:5:E: element "para" undefined
```

Other things to look for, if you haven't misspelled the public identifier, are typos in the catalog or failure to specify a catalog that resolves the public identifier that can't be found.

ISO Entity Set Missing

A missing entity set is another example of either a misspelled public identifier, or a missing catalog or catalog entry.

In this case, there's nothing wrong with the document, but the catalog that's been specified is missing the public identifiers for the ISO entity sets:

```
[n:\dbtdg]nsgmls -sv -c examples\errs\cat2 examples\simple.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:53:65:W: cannot generate system identifier for public text "ISO 8879:1986//ENTITIES Added Math Symbols: Arrow Relations//EN"
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:54:8:E: reference to entity "ISOamsa" for which no system identifier could be generated
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:52:0: entity was defined here
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:60:66:W: cannot generate system identifier for public text "ISO 8879:1986//ENTITIES Added Math Symbols: Binary Operators//EN"
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:61:8:E: reference to entity "ISOamsb" for which no system identifier could be generated
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:59:0: entity was defined here
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:67:60:W: cannot generate system identifier for public text "ISO 8879:1986//ENTITIES Added Math Symbols: Delimiters//EN"
```

```
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:68:8:E: reference to entity "ISOamsc" for which no system identifier could be generated
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:66:0: entity was defined here
m:\jade\nsgmls.exe:n:/share/sgml/docbook/3.1/dbcent.mod:74:67:W: cannot generate system identifier for public text "ISO 8879:1986//ENTITIES Added Math Symbols: Negated Relations//EN"
...
```

The ISO entity sets are required by the DocBook DTD, but they are not distributed with it. That's because they aren't maintained by OASIS.[13]

Character Data Not Allowed Here

Out of context character data is frequently caused by a missing start tag, but sometimes it's just the result of typing in the wrong place!

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
You can't put character data here.
<para>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\dbtdg] nsgmls -sv -c \share\sgml\catalog examples\errs\badpcdata.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
m:\jade\nsgmls.exe:examples\errs\badpcdata.sgm:9:0:E: character data is not allowed here
```

`Chapter`s aren't allowed to contain character data directly. Here, a wrapper element, such as `Para`, is missing around the sentence between the first two paragraphs.

Misspelled Start Tag

If you spell it wrong, the parser gets confused.

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
```

---

[13] If you need to locate the entity sets, consult http://www.oasis-open.org/cover/topics.html#entities.

```
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<paar>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\documents\books\dbtdg]nsgmls -sv -c \share\sgml\catalog examples\errs\misspe
ll.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
m:\jade\nsgmls.exe:examples\errs\misspell.sgm:9:5:E: element "PAAR" undefined
m:\jade\nsgmls.exe:examples\errs\misspell.sgm:14:6:E: end tag for element "PARA" which is not open
m:\jade\nsgmls.exe:examples\errs\misspell.sgm:21:9:E: end tag for "PAAR" omitted, but OMITTAG NO was specified
m:\jade\nsgmls.exe:examples\errs\misspell.sgm:9:0: start tag was here
```

Luckily, these are pretty easy to spot, unless you accidentally spell the name of another element. In that case, your error might appear to be out of context.

Misspelled End Tag

Spelling the end tag wrong is just as confusing.

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</titel>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\dbtdg]nsgmls -sv -c \share\sgml\catalog examples\errs\misspell2.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:2:35:E: end tag for element "TITEL" which is not open
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:3:5:E: document type does not allow element "PARA" here; missing one of "FOOTNOTE", "MSGTEXT" start-tag
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:9:5:E: document type does not allow element "PARA" here; missing one of "FOOTNOTE", "MSGTEXT" start-tag
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:15:5:E: document type does not allow element "PARA" here; missing one of "FOOTNOTE", "MSGTEXT" start-tag
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:21:9:E: end tag for "TITLE" omitted, but OMITTAG NO was specified
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:2:9: start tag was here
m:\jade\nsgmls.exe:examples\errs\misspell2.sgm:21:9:E: end tag for "CHAPTER" which is not finished
```

These are pretty easy to spot as well, but look at how confused the parser became. From the parser's point of view, failure to close the open `Title` element means that all the following elements appear out of context.

Out of Context Start Tag

Sometimes the problem isn't spelling, but placing a tag in the wrong context. When this happens, the parser tries to figure out what it can add to your document to make it valid. Then it proceeds as if it had seen what was added in order to recover from the error seen, which can cause future errors.

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para><title>Paragraph With Inlines</title>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\dbtdg]nsgmls -sv -c \share\sgml\catalog examples\errs\badstarttag.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
[illegible]
```

In this example, we probably wanted a `FormalPara`, so that we could have a title on the paragraph. But note that the parser didn't suggest this alternative. The parser only tries to add additional elements, rather than rename elements that it's already seen.

Missing End Tag

Leaving out an end tag is a lot like an out of context start tag. In fact, they're really the same error. The problem is never caused by the missing end tag per se, rather it's caused by the fact that something following it is now out of context.

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\dbtdg]nsgmls -sv -c \share\sgml\catalog examples\errs\noendtag.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
m:\jade\nsgmls.exe:examples\errs\noendtag.sgm:14:5:E: document type does not allow element "PARA" here; missing one of "FOOTNOTE", "MSGTEXT", "CAUTION", "IMPORTANT", "NOTE", "TIP", "WARNING", "BLOCKQUOTE", "INFORMALEXAMPLE" start-tag
m:\jade\nsgmls.exe:examples\errs\noendtag.sgm:20:9:E: end tag for "PARA" omitted, but OMITTAG NO was specified
m:\jade\nsgmls.exe:examples\errs\noendtag.sgm:9:0: start tag was here
```

In this case, the parser figured out that the best thing it could do is end the paragraph.

Bad Entity Reference

If you spell an entity name wrong, the parser will catch it.

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para>
There's no entity called &xyzzy; defined in this document.
</para>
<para>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
```

```
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\dbtdg]nsgmls -sv -c \share\sgml\catalog examples\errs\badent.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
m:\jade\nsgmls.exe:examples\errs\badent.sgm:10:26:E: general entity "xyzzy" not defined and no default entity
```

More often than not, you'll see this when you misspell a character entity name. For example, this happens when you type `&ldqou;` instead of `&ldquo;`.

Invalid 8-Bit Character

In XML, the entire range of Unicode characters is available to you, but in SGML, the declaration indicates what characters are valid. The distributed DocBook declaration doesn't allow a bunch of fairly common 8-bit characters.

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para>
The DocBook declaration in use doesn't allow 8 bit characters
like  this .
</para>
<para>
<emphasis role=bold>This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

```
[n:\dbtdg]nsgmls -sv -c \share\sgml\catalog examples\errs\badchar.sgm
m:\jade\nsgmls.exe:I: SP version "1.3.2"
```

```
m:\jade\nsgmls.exe:examples\errs\badchar.sgm:11:0:E: non SGML character number 147
m:\jade\nsgmls.exe:examples\errs\badchar.sgm:11:5:E: non SGML character number 148
```

In this example, the Windows code page values for curly left and right quotes have been used, but they aren't in the declared character set. Fix this by converting them to character entities.

You can also fix them by changing the declaration, but if you do that, make sure all your interchange partners are aware of, and have a copy of, the modified declaration. See Appendix F.

Considering Other Schema Languages

Historically, DTDs were the only way to describe the valid stricture of SGML and XML documents, but that is no longer the case. At the time of this writing (January, 2001), DocBook is experimentally available in three other schema languages:

XML Schema[iii]

> The schema language being defined by the W3C[iv] as the successor to DTDs for describing the structure of XML. XML Schema are likely to become a W3C[v] Recommendation in 2001.

RELAX[vi]

> RELAX, the Regular Language description for XML) is a less complex alternative to XML Schemas. The RELAX Core module is defined by ISO in *ISO/IEC DTR 22250-1, Document Description and Processing Languages -- Regular Language Description for XML (RELAX) -- Part 1: RELAX Core, 2000*. The RELAX Namespaces module is currently under development.

TREX[vii]

> TREX, Tree Regular Expressions for XML, is another less complex alternative to XML Schemas. It is concise, powerful, and datatype neutral.

Parsing and Validation

Before we look closer at these new schema languages, there's one significant difference between DTDs and all of them that we should get out of the way: XML parsers (which may understand DTDs) build an XML information set out of a stream of characters, all of these other schema languages begin with an information set and perform validation on it.

What I mean by that is that an XML parser reads a stream of bytes:

```
"<" "?" "x" "m" "l" " " "v" "e" ...
"<" "!" "D" "O" "C" "T" "Y" "P" "E" " " "b" "o" "o" "k" ...
"<" "b" "o" "o" "k" " " "i" "d" "=" "'" "f" "o" "o" "'" ">"
...
"<" "/" "b" "o" "o" "k" ">"
```

interprets them as a stream of characters (which may change the interpretation of some sequences of bytes) and constructs some representation of the XML document. This representation is the set of all the XML information items encountered:

[iii] http://www.w3.org/XML/Schema

[iv] http://www.w3.org/

[v] http://www.w3.org/

[vi] http://www.xml.gr.jp/relax/

[vii] http://www.thaiopensource.com/trex/

the information set of the document. The W3C[viii] XML Core Working Group[ix] is in the process of defining what an XML Information Set[x] contains.

The other schema languages are defined not in terms of the sequence of characters in the file but in terms of the information set of the XML document. They have to work this way because the XML Recommendation[xi] says what an XML document is and they all want to work on top of XML.

So what, you might ask? Well, it turns out that this has at least one very significant implication: there's no way for these languages to provide support for entity declarations.

An entity, like “`&ora;`” as a shortcut for “O'Reilly & Associates” or “`&eacute;`” as a mnemonic for “é”, is a feature of the character stream seen by the XML parser, it doesn't exist in the information set of valid XML documents. More importantly, this means that even if the schema language had a syntax for declaring entities, it wouldn't help the XML parser that needs to know the definitions long before the schema language processor comes into play.

There are a couple of other XML features that are impacted, though not necessarily as significantly: notations and default attribute values. One use for notations is on external entity declarations, and as we've already seen, the schema language is too late to be useful for anything entity related. Default attribute values are also problematic since you would like them to be in the information set produced by the parser so that the schema language sees them.

A Coarse Comparison of Three XML Schema Languages

FIXME: write a short synopsis of how these languages compare.

[viii] http://www.w3.org/

[ix] http://www.w3.org/XML/

[x] http://www.w3.org/TR/xml-infoset

[xi] http://www.w3.org/TR/REC-xml

4

Publishing DocBook Documents

$Revision: 1.3 $
$Date: 2002/04/18 22:06:23 $

Creating and editing SGML/XML documents is usually only half the battle. After you've composed your document, you'll want to publish it. Publishing, for our purposes, means either print or web publishing. For SGML and XML documents, this is usually accomplished with some kind of stylesheet. In the (not too distant) future, you may be able to publish an XML document on the Web by simply putting it online with a stylesheet, but for now you'll probably have to translate your document into HTML.

There are many ways, using both free and commercial tools, to publish SGML documents. In this chapter, we're going to survey a number of possibilities, and then look at just one solution in detail: Jade[i] and the Modular DocBook Stylesheets.[ii] We used jade to produce this book and to produce the online versions on the CD-ROM; it is also being deployed in other projects such as `<SGML>&tools;`,[iii] which originated with the Linux Documentation Project.

For a brief survey of other tools, see Appendix D.

A Survey of Stylesheet Languages

Over the years, a number of attempts have been made to produce a standard stylesheet language and, failing that, a large number of proprietary languages have been developed.

FOSIs

> First, the U.S. Department of Defense, in an attempt to standardize stylesheets across military branches, created the *Output Specification*, which is defined in MIL-PRF-28001C, *Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text*.[14]
>
> Commonly called FOSIs (for Formatting Output Specification Instances), they are supported by a few products including ADEPT Publisher by Arbortext[v] and DL Composer by Datalogics[vi].

DSSSL

---

[i] http://www.jclark.com/jade/

[ii] http://nwalsh.com/docbook/dsssl/

[iii] http://www.sgmltools.org/

[14] See *Formally Published CALS Standards [http://www-cals.itsi.disa.mil/core/formal/fps.htm]* for more information.

[v] http://www.arbortext.com/

[vi] http://www.datalogics.com/

Next, the International Organization for Standardization (ISO) created DSSSL, the Document Style Semantics and Specification Language. Subsets of DSSSL are supported by Jade and a few other tools, but it never achieved widespread support.

CSS

The W3C CSS Working Group created CSS as a style attachment language for HTML, and, more recently, XML.

XSL

Most recently, the XML effort has identified a standard Extensible Style Language (XSL) as a requirement. The W3C XSL Working Group is currently pursuing that effort.

Stylesheet Examples

By way of comparison, here's an example of each of the standard style languages. In each case, the stylesheet fragment shown contains the rules that reasonably formatted the following paragraph:

```
<para>
This is an example paragraph. It should be presented in a
reasonable body font. <emphasis>Emphasized</emphasis> words
should be printed in italics. A single level of
<emphasis>Nested <emphasis>emphasis</emphasis> should also
be supported.</emphasis>
</para>
```

FOSI stylesheet

FOSIs are SGML documents. The element in the FOSI that controls the presentation of specific elements is the `e-i-c` (element in context) element. A sample FOSI fragment is shown in Example 4.1.

Example 4.1. A Fragment of a FOSI Stylesheet

```
<e-i-c gi="para">
  <charlist>
    <textbrk startln="1" endln="1">
  </charlist>
</e-i-c>

<e-i-c gi="emphasis">
  <charlist inherit="1">
    <font posture="italic">
  </charlist>
</e-i-c>

<e-i-c gi="emphasis" context="emphasis">
  <charlist inherit="1">
    <font posture="upright">
  </charlist>
</e-i-c>
```

DSSSL stylesheet

DSSSL stylesheets are written in a Scheme-like language (see “Scheme” later in this chapter). It is the `element` function that controls the presentation of individual elements. See the example in Example 4.2.

Example 4.2. A Fragment of a DSSSL Stylesheet

```
(element para
  (make paragraph
    (process-children)))

(element emphasis
  (make sequence
    font-posture: 'italic
    (process-children)))

(element (emphasis emphasis)
  (make sequence
    font-posture: 'upright
    (process-children)))
```

CSS stylesheet

CSS stylesheets consist of selectors and formatting properties, as shown in Example 4.3.

Example 4.3. A Fragment of a CSS Stylesheet

```
para              { display: block }
emphasis          { display: inline;
                    font-style: italic; }
emphasis emphasis { display: inline;
                    font-style: upright; }
```

XSL stylesheet

XSL stylesheets are XML documents, as shown in Example 4.4. The element in the XSL stylesheet that controls the presentation of specific elements is the `xsl:template` element.

Example 4.4. A Fragment of an XSL Stylesheet

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

<xsl:template match="para">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="emphasis">
  <fo:sequence font-style="italic">
    <xsl:apply-templates/>
  </fo:sequence>
</xsl:template>

<xsl:template match="emphasis/emphasis">
  <fo:sequence font-style="upright">
    <xsl:apply-templates/>
  </fo:sequence>
</xsl:template>

</xsl:stylesheet>
```

Using Jade and DSSSL to Publish DocBook Documents

Jade is a free tool that applies DSSSL[vii] stylesheets to SGML and XML documents. As distributed, Jade can output RTF, TeX, MIF, and SGML. The SGML backend can be used for SGML to SGML transformations (for example, DocBook to HTML).

A complete set of DSSSL stylesheets for creating print and HTML output from DocBook is included on the CD-ROM. More information about obtaining and installing Jade appears in Appendix A.

A Brief Introduction to DSSSL

DSSSL is a stylesheet language for both print and online rendering. The acronym stands for *Document Style Semantics and Specification Language*. It is defined by ISO/IEC 10179:1996. For more general information about DSSSL, see the DSSSL Page[viii].

Scheme

The DSSSL expression language is Scheme, a variant of Lisp. Lisp is a functional programming language with a remarkably regular syntax. Every expression looks like this:

```
(operator [arg1] [arg2] ... [argn] )
```

This is called “prefix” syntax because the operator comes before its arguments.

In Scheme, the expression that subtracts 2 from 3, is `(- 3 2)`. And `(+ (- 3 2) (* 2 4))` is 9. While the prefix syntax and the parentheses may take a bit of getting used to, Scheme is not hard to learn, in part because there are no exceptions to the syntax.

DSSSL Stylesheets

A complete DSSSL stylesheet is shown in Example 4.5. After only a brief examination of the stylesheet, you'll probably begin to have a feel for how it works. For each element in the document, there is an element rule that describes how you should format that element. The goal of the rest of this chapter is to make it possible for you to read, understand, and even write stylesheets at this level of complexity.

Example 4.5. A Complete DSSSL Stylesheet

```
<!DOCTYPE style-sheet PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN">

<style-sheet>
<style-specification>
<style-specification-body>

(element chapter
  (make simple-page-sequence
    top-margin: 1in
    bottom-margin: 1in
    left-margin: 1in
    right-margin: 1in
    font-size: 12pt
    line-spacing: 14pt
```

[vii] http://www.jclark.com/dsssl/

[viii] http://www.jclark.com/dsssl/

```
    min-leading: 0pt
    (process-children)))

(element title
  (make paragraph
    font-weight: 'bold
    font-size: 18pt
    (process-children)))

(element para
  (make paragraph
    space-before: 8pt
    (process-children)))

(element emphasis
  (if (equal? (attribute-string "role") "strong")
      (make sequence
 font-weight: 'bold
 (process-children))
      (make sequence
 font-posture: 'italic
 (process-children))))

(element (emphasis emphasis)
  (make sequence
    font-posture: 'upright
    (process-children)))

(define (super-sub-script plus-or-minus
               #!optional (sosofo (process-children)))
  (make sequence
    font-size: (* (inherited-font-size) 0.8)
    position-point-shift: (plus-or-minus (* (inherited-font-size) 0.4))
    sosofo))

(element superscript (super-sub-script +))
(element subscript (super-sub-script -))

</style-specification-body>
</style-specification>
</style-sheet>
```

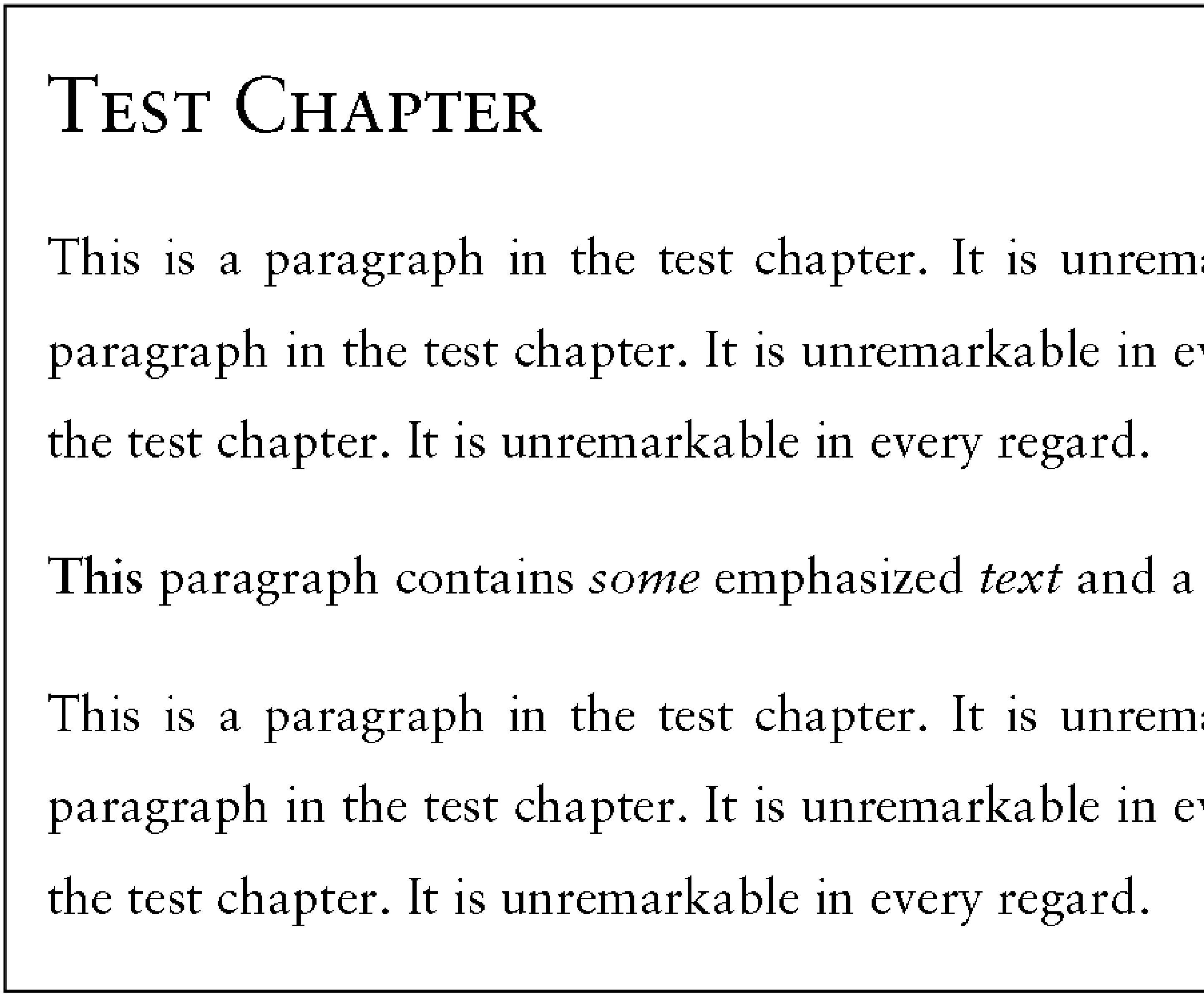
This stylesheet is capable of formatting simple DocBook documents like the one shown in Example 4.6.

Example 4.6. A Simple DocBook Document

```
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD Docbook XML V4.1.2//EN"
                  "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
<chapter><title>Test Chapter</title>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
<para>
<emphasis role="bold">This</emphasis> paragraph contains
<emphasis>some <emphasis>emphasized</emphasis> text</emphasis>
and a <superscript>super</superscript>script
and a <subscript>sub</subscript>script.
</para>
<para>
This is a paragraph in the test chapter. It is unremarkable in
every regard. This is a paragraph in the test chapter. It is
unremarkable in every regard. This is a paragraph in the test
chapter. It is unremarkable in every regard.
</para>
</chapter>
```

The result of formatting a simple document with this stylesheet can be seen in Figure 4.1.

Figure 4.1. The formatted simple document



TEST CHAPTER

This is a paragraph in the test chapter. It is unrema
paragraph in the test chapter. It is unremarkable in ev
the test chapter. It is unremarkable in every regard.

This paragraph contains *some* emphasized ***text*** and a

This is a paragraph in the test chapter. It is unrema
paragraph in the test chapter. It is unremarkable in ev
the test chapter. It is unremarkable in every regard.

We'll take a closer look at this stylesheet after you've learned a little more DSSSL.

DSSSL Stylesheets Are SGML Documents

One of the first things that may strike you about DSSSL stylesheets (aside from all the parentheses), is the fact that the stylesheet itself is an SGML document! This means that you have all the power of SGML documents at your disposal in DSSSL stylesheets. In particular, you can use entities and marked sections to build a modular stylesheet.

In fact, DSSSL stylesheets are defined so that they correspond to a particular architecture. This means that you can change the DTD used by stylesheets within the bounds of the architecture. A complete discussion of document architectures is beyond the scope of this book, but we'll show you one way to take advantage of them in your DSSSL stylesheets in the section called “The DSSSL Architecture”“ later in the chapter.

DSSSL Processing Model

A DSSSL processor builds a tree out of the source document. Each element in the source document becomes a node in the tree (processing instructions and other constructs become nodes as well). Processing the source tree begins with the root rule and continues until there are no more nodes to process.

Global Variables and Side Effects

There aren't any global variables or side effects. It can be difficult to come to grips with this, especially if you're just starting out.

It is possible to define constants and functions and to create local variables with `let` expressions, but you can't create any global variables or change anything after you've defined it.

DSSSL Expressions

DSSSL has a rich vocabulary of expressions for dealing with all of the intricacies of formatting. Many, but by no means all of them, are supported by Jade. In this introduction, we'll cover only a few of the most common.

Element expressions

Element expressions, which define the rules for formatting particular elements, make up the bulk of most DSSSL stylesheets. A simple element rule can be seen in Example 4.7. This rule says that a `para` element should be formatted by making a paragraph (see the section called “Make expressions”“).

Example 4.7. A Simple DSSSL Rule

```
(element para
  (make paragraph
    space-before: 8pt
    (process-children)))
```

An element expression can be made more specific by specifying an element and its ancestors instead of just specifying an element. The rule `(element title ...)` applies to all `Title` elements, but a rule that begins `(element (figure title) ...)` applies only to `Title` elements that are immediate children of `Figure` elements.

If several rules apply, the most specific rule is used.

When a rule is used, the node in the source tree that was matched becomes the “current node” while that element expression is being processed.

Make expressions

A make expression specifies the characteristics of a “flow object.” Flow objects are abstract representations of content (paragraphs, rules, tables, and so on). The expression:

```
(make paragraph
  font-size: 12pt
  line-spacing: 14pt ...)
```

specifies that the content that goes “here” is to be placed into a paragraph flow object with a font-size of 12pt and a line-spacing of 14pt (all of the unspecified characteristics of the flow object are defaulted in the appropriate way).

They're called flow objects because DSSSL, in its full generality, allows you to specify the characteristics of a sequence of flow objects and a set of areas on the physical page where you can place content. The content of the flow objects is then “poured on to” (or flows in to) the areas on the page(s).

In most cases, it's sufficient to think of the make expressions as constructing the flow objects, but they really only specify the *characteristics* of the flow objects. This detail is apparent in one of the most common and initially confusing pieces of DSSSL jargon: the sosofo. Sosofo stands for a “specification of a sequence of flow objects.” All this means is that processing a document may result in a nested set of `make` expressions (in other words, the paragraph may contain a table that contains rows that contain cells that contain paragraphs, and so on).

The general form of a `make` expression is:

```
(make flow-object-name
  keyword1: value1
  keyword2: value2
  ...
  keywordn: valuen
  (content-expression))
```

Keyword arguments specify the characteristics of the flow object. The specific characteristics you use depends on the flow object. The *`content-expression`* can vary; it is usually another make expression or one of the processing expressions.

Some common flow objects in the print stylesheet are:

`simple-page-sequence`

> Contains a sequence of pages. The keyword arguments of this flow object let you specify margins, headers and footers, and other page-related characteristics. Print stylesheets should always produce one or more `simple-page-sequence` flow objects.
>
> Nesting `simple-page-sequence` does not work. Characteristics on the inner sequences are ignored.

`paragraph`

> A paragraph is used for any block of text. This may include not only paragraphs in the source document, but also titles, the terms in a definition list, glossary entries, and so on. Paragraphs in DSSSL can be nested.

`sequence`

> A sequence is a wrapper. It is most frequently used to change inherited characteristics (like font style) of a set of flow objects without introducing other semantics (such as line breaks).

`score`

A score flow object creates underlining, strike-throughs, or overlining.

`table`

A table flow object creates a table of rows and cells.

The HTML stylesheet uses the SGML backend, which has a different selection of flow objects.

`element`

Creates an element. The content of this `make` expression will appear between the start and end tags. The expression:

```
(make element gi: "H1"
     (literal "Title"))
```

produces `<H1>Title</H1>`.

`empty-element`

Creates an empty element that may not have content. The expression:

```
(make empty-element gi: "BR"
     attributes: '(("CLEAR" "ALL")))
```

produces `<BR CLEAR="ALL">`.

`sequence`

Produces no output in of itself as a wrapper, but is still required in DSSSL contexts in which you want to output several flow objects but only one object top-level object may be returned.

`entity-ref`

Inserts an entity reference. The expression:

```
(make entity-ref name: "nbsp")
```

produces `&nbsp;`.

In both stylesheets, a completely empty flow object is constructed with `(empty-sosofo)`.

Selecting data

Extracting parts of the source document can be accomplished with these functions:

`(data` *`nd`*`)`

Returns all of the character data from *`nd`* as a string.

`(attribute-string "`*`attr`*`"` *`nd`*`)`

Returns the value of the *`attr`* attribute of *`nd`*.

`(inherited-attribute-string "`*`attr`*`"` *`nd`*`)`

Returns the value of the *`attr`* attribute of *`nd`*. If that attribute is not specified on *`nd`*, it searches up the hierarchy for the first ancestor element that does set the attribute, and returns its value.

Selecting elements

A common requirement of formatting is the ability to reorder content. In order to do this, you must be able to select other elements in the tree for processing. DSSSL provides a number of functions that select other elements. These functions all return a list of nodes.

`(current-node)`

Returns the current node.

`(children` *`nd`*`)`

Returns the children of *`nd`*.

`(descendants` *`nd`*`)`

Returns the descendants of *`nd`* (the children of *`nd`* and all their children's children, and so on).

`(parent` *`nd`*`)`

Returns the parent of *`nd`*.

`(ancestor "`*`name`*`"` *`nd`*`)`

Returns the first ancestor of *`nd`* named *`name`*.

`(element-with-id "`*`id`*`")`

Returns the element in the document with the ID *`id`*, if such an element exists.

`(select-elements` *`node-list`* `"`*`name`*`")`

Returns all of the elements of the *`node-list`* that have the name *`name`*. For example, `(select-elements (descendants (current-node)) "para")` returns a list of all the paragraphs that are descendants of the current node.

`(empty-node-list)`

Returns a node list that contains no nodes.

Other functions allow you to manipulate node lists.

`(node-list-empty?` *`nl`*`)`

Returns true if (and only if) *`nl`* is an empty node list.

`(node-list-length` *`nl`*`)`

Returns the number of nodes in *`nl`*.

`(node-list-first` *`nl`*`)`

Returns a node list that consists of the single node that is the first node in *`nl`*.

`(node-list-rest` *`nl`*`)`

Returns a node list that contains all of the nodes in *`nl`* except the first node.

There are many other expressions for manipulating nodes and node lists.

Processing expressions

Processing expressions control which elements in the document will be processed and in what order. Processing an element is performed by finding a matching element rule and using that rule.

`(process-children)`

Processes all of the children of the current node. In most cases, if no process expression is given, processing the children is the default behavior.

`(process-node-list` *`nl`*`)`

Processes each of the elements in *`nl`*.

Define expressions

You can declare your own functions and constants in DSSSL. The general form of a function declaration is:

```
(define (function args)
  function-body)
```

A constant declaration is:

```
(define constant
  constant-function-body)
```

The distinction between constants and functions is that the body of a constant is evaluated when the definition occurs, while functions are evaluated when they are used.

Conditionals

In DSSSL, the constant `#t` represents true and `#f` false. There are several ways to test conditions and take action in DSSSL.

`if`

The form of an `if` expression is:

```
(if condition
  true-expression
  false-expression)
```

If the condition is true, the *`true-expression`* is evaluated, otherwise the *`false-expression`* is evaluated. You must always provide an expression to be evaulated when the condition is not met. If you want to produce nothing, use `(empty-sosofo)`.

`case`

`case` selects from among several alternatives:

```
(case expression
  ((constant1) (expression1)
  ((constant2) (expression2)
  ((constant3) (expression3)
  (else else-expression))
```

The value of the expression is compared against each of the constants in turn and the expression associated with the first matching constant is evaulated.

`cond`

`cond` also selects from among several alternatives, but the selection is performed by evaluating each expression:

```
(cond
  ((condition1) (expression1)
  ((condition2) (expression2)
  ((condition3) (expression3)
  (else else-expression))
```

The value of each conditional is calculated in turn. The expression associated with the first condition that is true is evaluated.

Any expression that returns `#f` is false; all other expressions are true. This can be somewhat counterintuitive. In many programming languages, it's common to assume that “empty” things are false (0 is false, a null pointer is false, an empty set is false, for example.) In DSSSL, this isn't the case; note, for example, that an empty node list is not `#f` and is therefore true. To avoid these difficulties, always use functions that return true or false in conditionals. To test for an empty node list, use `(node-list-empty?)`.

Let expressions

The way to create local variables in DSSSL is with `(let)`. The general form of a `let` expression is:

```
(let ((var1 expression1)
      (var2 expression2)
      ...
      (varn expressionn))
  let-body)
```

In a `let`; expression, all of the variables are defined “simultaneously.” The expression that defines *`var2`* cannot contain any references to any other variables defined in the same `let` expression. A `let*` expression allows variables to refer to each other, but runs slightly slower.

Variables are available only within the *`let-body`*. A common use of `let` is within a `define` expression:

```
(define (cals-rule-default nd)
   (let* ((table (ancestor "table" nd))
          (frame (if (attribute-string "frame" table)
                     (attribute-string "frame" table)
                     "all")))
    (equal? frame "all")))
```

This function creates two local variables `table` and `frame`. `let` returns the value of the last expression in the body, so this function returns true if the `frame` attribute on the table is `all` or if no `frame` attribute is present.

Loops

DSSSL doesn't have any construct that resembles the “for loop” that occurs in most imperative languages like C and Java. Instead, DSSSL employs a common trick in functional languages for implementing a loop: tail recursion.

Loops in DSSSL use a special form of `let`. This loop counts from 1 to 10:

```
(let ❶loopvar ❷((count 1))
  ❸(if (> count 10)
    ❹#t
    (❺loopvar ❻(+ count 1))))
```

❶ This variable controls the loop. It is declared without an initial value, immediately after the `let` operand.

❷ Any number of additional local variables can be defined after the loop variable, just as they can in any other `let` expression.

❸ If you ever want the loop to end, you have to put some sort of a test in it.

❹ This is the value that will be returned.

❺ Note that you iterate the loop by using the loop variable as if it was a function name.

❻ The arguments to this “function” are the values that you want the local variables declared in ❷ to have in the next iteration.

A Closer Look at Example 4.5

Example 4.5 is a style sheet that contains a style specification. Stylesheets may consist of multiple specifications, as we'll see in the section called “A Single Stylesheet for Both Print and HTML”.“

The actual DSSSL code goes in the style specification body, within the style specification. Each construction rule processes different elements from the source document.

Processing chapters

`Chapter`s are processed by the `chapter` construction rule. Each `Chapter` is formatted as a `simple-page-sequence`. Every print stylesheet should format a document as one or more simple page sequences. Characteristics on the simple page sequence can specify headers and footers as well as margins and other page parameters.

One important note about simple page sequences: they cannot nest. This means that you cannot blindly process divisions (`Part`s, `Reference`) and the elements they contain (`Chapter`s, `RefEntry`s) as simple page sequences. This sometimes involves a little creativity.

Processing titles

The `make` expression in the `title` element rule ensures that `Title`s are formatted in large, bold print.

This construction rule applies equally to `Chapter` titles, `Figure` titles, and `Book` titles. It's unlikely that you'd want all of these titles to be presented in the same way, so a more robust stylesheet would have to arrange the processing of titles with more context. This might be achieved in the way that nested `Emphasis` elements are handled in the section called “Processing emphasis””.

Processing paragraphs

`Para` elements are simply formatted as paragraphs.

Processing emphasis

Processing `Emphasis` elements is made a little more interesting because we want to consider an attribute value and the possibility that `Emphasis` elements can be nested.

In the simple case, in which we're processing an `Emphasis` element that is not nested, we begin by testing the value of the `role` attribute. If the content of that attribute is the string `strong`, it is formatted in bold; otherwise, it is formatted in italic.

The nested case is handled by the `(emphasis emphasis)` rule. This rule simply formats the content using an upright (nonitalic) font. This rule, like the rule for `Title`s, is not robust. `Emphasis` nested inside `strong` `Emphasis` won't be distinguished, for example, and nestings more than two elements deep will be handled just as nestings that are two deep.

Processing subscripts and superscripts

Processing `Subscript` and `Superscript` elements is really handled by the `super-sub-script` function. There are several interesting things about this function:

The `plus-or-minus` argument

> You might ordinarily think of passing a keyword or boolean argument to the `super-sub-script` function to indicate whether subscripts or superscripts are desired. But with Scheme, it's possible to pass the actual function as an argument!
>
> Note that in the element construction rules for `Superscript` and `Subscript`, we pass the actual functions `+` and `-`. In the body of `super-sub-script`, we use the `plus-or-minus` argument as a function name (it appears immediately after an open parenthesis).

The optional argument

> `optional` arguments are indicated by `#!optional` in the function declaration. Any number of `optional` arguments may be given, but each must specify a default value. This is accomplished by listing each argument and default value (an expression) as a pair.
>
> In `super-sub-script`, the optional argument `sosofo` is initialized to `process-children`. This means that at the point where the function is *called*, `process-children` is evaluated and the resulting `sosofo` is passed to the function.

Use of inherited characteristics

> It is possible to use the “current” value of an inherited characteristic to calculate a new value. Using this technique, superscripts and subscripts will be presented at 80 percent of the current font size.

Customizing the Stylesheets

The best way to customize the stylesheets is to write your own “driver” file; this is a stylesheet that contains your local modifications and then includes the appropriate stylesheet from the standard distribution by reference. This allows you to make local changes and extensions without modifying the distributed files, which makes upgrading to the next release much simpler.

Writing Your Own Driver

A basic driver file looks like this:

```
<!DOCTYPE style-sheet PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN" [
<!ENTITY dbstyle PUBLIC "-//Norman Walsh//DOCUMENT DocBook Print Stylesheet//EN" CDATA DSSSL>
]>

<style-sheet>
<style-specification use="docbook">
<style-specification-body>

;; your changes go here...

</style-specification-body>
</style-specification>
<external-specification id="docbook" document="dbstyle">
</style-sheet>
```

There are two public identifiers associated with the Modular DocBook Stylesheets:

- `-//Norman Walsh//DOCUMENT DocBook Print Stylesheet//EN`
- `-//Norman Walsh//DOCUMENT DocBook HTML Stylesheet//EN`

The former selects the print stylesheet and the latter selects the HTML stylesheet. There is an SGML Open catalog file in the distribution that maps these public identifiers to the stylesheet files.

You can add your own definitions, or redefinitions, of stylesheet rules and parameters so that

```
;; your changes go here...
```

occurs in the previous example.

For a concrete example of a driver file, see `plain.dsl` in the `docbook/print` directory in the stylesheet distribution (or on the CD-ROM). This is a customization of the print stylesheet, which turns off title page and TOC generation.

Changing the Localization

As distributed, the stylesheets use English for all generated text, but other localization files are also provided. At the time of this writing, the stylesheets support Catalan, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Italian, Japanese, Norwegian, Polish, Portuguese, Portuguese (Brazil), Romanian, Russian, Slovak, Spanish, and Swedish. (If you can write a localization for another language, *please* contribute it.)

There are two ways to switch languages: by specifying a `lang` attribute, or by changing the default language in a customization.

Using the `lang` attribute

One of the DocBook common attributes is `lang`. If you specify a language, the DocBook stylesheets will use that language (and all its descendants, if no other language is specified) for generated text within that element.

Table 4.1 summarizes the language codes for the supported languages.[15] The following chapter uses text generated in French:

```
<chapter lang="fr"><title>Bêtises</title>
<para>Pierre qui roule n'amasse pas de mousse.</para>
</chapter>
```

Table 4.1. DocBook Stylesheet Language Codes

Language Code	Language
`da`	Danish
`de`	German
`en`	English
`es`	Spanish
`fi`	Finnish
`fr`	French
`it`	Italian
`nl`	Dutch
`no`	Norwegian
`pl`	Polish
`pt`	Portuguese
`ru`	Russian
`sv`	Swedish

Changing the default language

If no `lang` attribute is specified, the default language is used. You can change the default language with a driver.

In the driver, define the default language. Table 4.1 summarizes the language codes for the supported languages. The following driver makes German the default language:

```
<!DOCTYPE style-sheet PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN" [
<!ENTITY dbstyle PUBLIC "-//Norman Walsh//DOCUMENT DocBook Print Stylesheet//EN" CDATA DSSSL>
]>

<style-sheet>
<style-specification use="docbook">
<style-specification-body>

(define %default-language% "dege")

</style-specification-body>
```

---

[15] Language codes should conform to IETF RFC 3066.

```
</style-specification>
<external-specification id="docbook" document="dbstyle">
</style-sheet>
```

There are two other settings that can be changed only in a driver. Both of these settings are turned off in the distributed stylesheet:

`%gentext-language%`

> If a language code is specified in `%gentext-language%`, then that language will be used for all generated text, regardless of any `lang` attribute settings in the document.

`%gentext-use-xref-language%`

> If turned on (defined as `#t`), then the stylesheets will generate the text associated with a cross reference using the language of the target, not the current language. Consider the following book:
>
> ```
> <book><title>A Test Book</title>
> <preface>
> <para>There are three chapters in this book: <xref linkend=c1>,
> <xref linkend=c2>, and <xref linkend=c3>.
> </para>
> </preface>
> <chapter lang=usen><title>English</title> ... </chapter>
> <chapter lang=fr><title>French</title> ... </chapter>
> <chapter lang=dege><title>Deutsch</title> ... </chapter>
> </book>
> ```
>
> The standard stylesheets render the Preface as something like this:
>
> > There are three chapters in this book: Chapter 1, Chapter 2, and Chapter 3.
>
> With `%gentext-use-xref-language%` turned on, it would render like this:
>
> > There are are three chapters in this book: Chapter 1, Chapitre 2, and Kapitel 3.

A Single Stylesheet for Both Print and HTML

A DSSSL stylesheet consists of one or more “style specifications.” Using more than one style specification allows you to build a single stylesheet file that can format with either the print or SGML backends. Example 4.8 shows a stylesheet with two style specifications.

Example 4.8. both.dsl: A Stylesheet with Two Style Specifications

```
<!DOCTYPE style-sheet PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN" [
<!ENTITY html-ss
  PUBLIC "-//Norman Walsh//DOCUMENT DocBook HTML Stylesheet//EN" CDATA dsssl>
<!ENTITY print-ss
  PUBLIC "-//Norman Walsh//DOCUMENT DocBook Print Stylesheet//EN" CDATA dsssl>
]>
<style-sheet>
<style-specification id="print" use="print-stylesheet">
<style-specification-body>

;; customize the print stylesheet
```

```
</style-specification-body>
</style-specification>
<style-specification id="html" use="html-stylesheet">
<style-specification-body>

;; customize the html stylesheet

</style-specification-body>
</style-specification>
<external-specification id="print-stylesheet" document="print-ss">
<external-specification id="html-stylesheet"  document="html-ss">
</style-sheet>
```

Once you have stylesheets with more than one style specification, you have to be able to indicate which style specification you want to use. In Jade, you indicate this by providing the ID of the style specification after the stylesheet filename, separated with a hash mark: `#`.

Using the code from Example 4.8, you can format a document using the print stylesheet by running:

```
jade -t rtf -d both.dsl#print file.sgm
```

and using the HTML stylesheet by running:

```
jade -t sgml -d both.dsl#html file.sgm
```

Dealing with Multiple Declarations

The DocBook SGML DTD and the DocBook DSSSL Stylesheets happen to use the same SGML declaration. This makes it very easy to run Jade with DocBook. However, you may sometimes wish to use Jade with other document types, for example the DocBook XML DTD, which has a different declaration. There are a couple of ways to do this.

Pass the Declaration Explicitly

If your stylesheets parse fine with the default declaration, but you want to use an alternate declaration with a particular document, just pass the declaration on the command line:

```
jade options the-declaration the-document
```

Note that there's no option required before the declaration; it simply occurs before the first filename. Jade concatenates all of the files that you give it together, and parses them as if they were one document.

Use the Catalogs

The other way to fix this is with a little catalog trickery.

First, note that Jade always looks in the file called `catalog` in the same directory as the document that it is loading, and uses settings in that file in preference to settings in other catalogs.

With this fact, we can employ the following trick:

- Put a `catalog` file in the directory that contains your stylesheets, which contain an `SGMLDECL` directive. Jade understands the directive, which points to the SGML declaration that you should use when parsing the stylesheets. For the DocBook stylesheets, the DocBook declaration works fine.

- In the directory that contains the document you want to process, create a `catalog` file that contains an `SGMLDECL` directive that points to the SGML declaration that should be used when parsing the document.

There's no easy way to have both the stylesheet and the document in the same directory if they must be processed with different declarations. But this is usually not too inconvenient.

The DSSSL Architecture

The concept of an architecture was promoted by HyTime. In some ways, it takes the standard SGML/XML notions of the role of elements and attributes and inverts them. Instead of relying on the name of an element to assign its primary semantics, it uses the values of a small set of fixed attributes.

While this may be counterintuitive initially, it has an interesting benefit. An architecture-aware processor can work transparently with many different DTDs. A small example will help illustrate this point.

Note

The following example demonstrates the concept behind architectures, but for the sake of simplicity, it does not properly implement an architecture as defined in HyTime.

Imagine that you wrote an application that can read an SGML/XML document containing a letter (conforming to some letter DTD), and automatically print an envelope for the letter. It's easy to envision how this works. The application reads the content of the letter, extracts the address and return address elements from the source, and uses them to generate an envelope:

```
<?xml version='1.0'>
<!DOCTYPE letter "/share/sgml/letter/letter.dtd" [
<!ENTITY myaddress "/share/sgml/entities/myaddress.xml">
]>
<letter>
<returnaddress>&myaddress;</returnaddress>
<address>
<name>Leonard Muellner</name>
<company>O'Reilly &amp; Associates</company>
<street>90 Sherman Street</street>
<city>Cambridge</city><state>MA</state><zip>02140</zip>
</address>
<body>
<salutation>Hi Lenny</salutation>
...
</body>
```

The processor extracts the `Returnaddress` and `Address` elements and their children and prints the envelope accordingly.

Now suppose that a colleague from payroll comes by and asks you to adapt the application to print envelopes for mailing checks, using the information in the payroll database, which has a different DTD. And a week later, someone from sales comes by and asks if you can modify the application to use the contact information DTD. After a while, you would have 11 versions of this program to maintain.

Suppose that instead of using the actual element names to locate the addresses in the documents, you asked each person to add a few attributes to their DTD. By forcing the attributes to have fixed values, they'd automatically be present in each document, but authors would never have to worry about them.

For example, the address part of the letter DTD might look like this:

```
<!ELEMENT address (name, company? street*, city, state, zip)>
<!ATTLIST address
 ADDRESS CDATA #FIXED "START"
>

<!ELEMENT name (#PCDATA)*>
<!ATTLIST name
 ADDRESS CDATA #FIXED "NAME"
>

<!ELEMENT company (#PCDATA)*>
<!ATTLIST company
 ADDRESS CDATA #FIXED "COMPANY"
>

<!ELEMENT street (#PCDATA)*>
<!ATTLIST street
 ADDRESS CDATA #FIXED "STREET"
>

<!ELEMENT city (#PCDATA)*>
<!ATTLIST city
 ADDRESS CDATA #FIXED "CITY"
>

<!ELEMENT state (#PCDATA)*>
<!ATTLIST state
 ADDRESS CDATA #FIXED "STATE"
>

<!ELEMENT zip (#PCDATA)*>
<!ATTLIST zip
 ADDRESS CDATA #FIXED "ZIP"
>
```

Effectively, each address in a letter would look like this:

```
<address ADDRESS="START">
<name ADDRESS="NAME">Leonard Muellner</name>
<company ADDRESS="COMPANY">O'Reilly &amp;amp; Associates</company>
<street> ADDRESS="STREET">90 Sherman Street</street>
<city ADDRESS="CITY">Cambridge</city><state ADDRESS="STATE">MA</state>
<zip ADDRESS="ZIP">02140</zip>
</address>
```

In practice, the author would not include the `ADDRESS` attributes; they are automatically provided by the DTD because they are `#FIXED`.[16]

Now the address portion of the payroll DTD might look like this:

```
<!ELEMENT employee (name, mailingaddress)>
```

---

[16] The use of uppercase names here is intentional. These are not attributes that an author is ever expected to type. In XML, which is case-sensitive, using uppercase for things like this reduces the likelihood of collision with “real” attribute names in the DTD.

```
<!ELEMENT name (#PCDATA)*>
<!ATTLIST name
 ADDRESS CDATA #FIXED "NAME"
>

<!ELEMENT mailingaddress (addrline1, addrline2,
                          city, state.or.province, postcode)>
<!ATTLIST mailingaddress
 ADDRESS CDATA #FIXED "START"
>

<!ELEMENT addrline1 (#PCDATA)*>
<!ATTLIST addrline1
 ADDRESS CDATA #FIXED "STREET"
>

<!ELEMENT addrline2 (#PCDATA)*>
<!ATTLIST addrline2
 ADDRESS CDATA #FIXED "STREET"
>

<!ELEMENT city (#PCDATA)*>
<!ATTLIST city
 ADDRESS CDATA #FIXED "CITY"
>

<!ELEMENT state.or.province (#PCDATA)*>
<!ATTLIST state.or.province
 ADDRESS CDATA #FIXED "STATE"
>

<!ELEMENT postcode (#PCDATA)*>
<!ATTLIST postcode
 ADDRESS CDATA #FIXED "ZIP"
>
```

The employee records will look like this:

```
<employee><name ADDRESS="NAME">Leonard Muellner</name>
<mailingaddress ADDRESS="START">
<addrline1 ADDRESS="STREET">90 Sherman Street</addrline1>
<city ADDRESS="CITY">Cambridge</city>
<state.or.province ADDRESS="STATE">MA</state.or.province>
<postcode ADDRESS="ZIP">02140</postcode>
</mailingaddress>
</employee>
```

Your application no longer cares about the actual element names. It simply looks for the elements with the correct attributes and uses them. This is the power of an architecture: it provides a level of abstraction that processing applications can use to their advantage. In practice, architectural forms are a bit more complex to set up because they have facilities for dealing with attribute name conflicts, among other things.

Why have we told you all this? Because DSSSL is an architecture. This means you can modify the stylesheet DTD and still run your stylesheets through Jade.

Consider the case presented earlier in Example 4.8. In order to use this stylesheet, you must specify three things: the backend you want to use, the stylesheet you want to use, and the style specification you want to use. If you mismatch any of the parameters, you'll get the wrong results. In practice, the problem is compounded further:

- Some stylesheets support several backends (RTF, TeX, and SGML).
- Some stylesheets support only some backends (RTF and SGML, but not TeX or MIF).
- Some stylesheets support multiple outputs using the same backend (several kinds of HTML output, for example, using the SGML backend: HTML, HTMLHelp, JavaHelp, and so on).
- If you have complex stylesheets, some backends may require additional options to define parameter entities or stylesheet options.

None of this complexity is really necessary, after all, the options don't change—you just have to use the correct combinations. The mental model is really something like this: “I want a certain kind of output, TeX say, so I have to use this combination of parameters.”

You can summarize this information in a table to help keep track of it:

Desired Output	Backend	Style specification	Options	Supported?
rtf	rtf	print	-V rtf-backend	yes
tex	tex	print	-V tex-backend -i tex	yes
html	sgml	htmlweb	-i html	yes
javahelp	sgml	help	-i help	yes
htmlhelp				no

Putting this information in a table will help you keep track of it, but it's not the best solution. The ideal solution is to keep this information on your system, and let the software figure it all out. You'd like to be able to run a command, tell it what output you want from what stylesheet, what file you want to process, and then let it figure everything else out. For example:

```
format html mybook.dsl mydoc.sgm
```

One way to do this is to put the configuration data in a separate file, and have the **format** command load it out of this other file. The disadvantage of this solution is that it introduces another file that you have to maintain and it's independent from the stylesheet so it isn't easy to keep it up-to-date.

In the DSSSL case, a better alternative is to modify the stylesheet DTD so you can store the configuration data *in the stylesheet*. Using this alternate DTD, your `mybook.dsl` stylesheets might look like this:

```
<!DOCTYPE style-sheet
  PUBLIC "-//Norman Walsh//DTD Annotated DSSSL Style Sheet V1.2//EN" [
<!-- perhaps additional declarations here -->
]>
<style-sheet>
<title>DocBook Stylesheet</title>
<doctype pubid="-//OASIS//DTD DocBook V3.1//EN">
<doctype pubid="-//Davenport//DTD DocBook V3.0//EN">
<doctype pubid="-//Norman Walsh//DTD Website V1.4//EN">
<backend name="rtf"  backend="rtf"  fragid="print"
         options="-V rtf-backend" default="true">
<backend name="tex"  backend="tex"  fragid="print"
```

```
         options="-V tex-backend -i tex">
<backend name="html" backend="sgml" fragid="htmlweb" options="-i html">
<backend name="javahelp" backend="sgml" fragid="help"  options="-i help">
<backend name="htmlhelp" supported="no">
<style-specification id="print" use="docbook">
<style-specification-body>
.
.
.
```

In this example, the stylesheet has been annotated with a title, a list of the public IDs to which it is applicable, and a table that provides information about the output formats that it supports.

Using this information, the **format** command can get all the information it needs to construct the appropriate call to Jade. To make HTML from `myfile.sgm`, **format** would run the following:

```
jade -t sgml -d mybook.dsl#htmlweb -i html myfile.sgm
```

The additional information, titles and public IDs, can be used as part of a GUI interface to simplify the selection of stylesheets for an author.

The complete annotated stylesheet DTD, and an example of the **format** command script, are provided on the CD-ROM.

A Brief Introduction to XSL

Bob Stayton

Using XSL tools to publish DocBook documents

There is a growing list of tools to process DocBook documents using XSL stylesheets. Each tool implements parts or all of the XSL standard, which actually has several components:

Extensible Stylesheet Language (XSL)

> A language for expressing stylesheets written in XML. It includes the formatting object language, but refers to separate documents for the transformation language and the path language.

XSL Transformation (XSLT)

> The part of XSL for transforming XML documents into other XML documents, HTML, or text. It can be used to rearrange the content and generate new content.

XML Path Language (XPath)

> A language for addressing parts of an XML document. It is used to find the parts of your document to apply different styles to. All XSL processors use this component.

To publish HTML from your XML documents, you just need an XSLT engine. To get to print, you need an XSLT engine to produce formatting objects (FO), which then must be processed with a formatting object processor to produce PostScript or PDF output.

James Clark's XT was the first useful XSLT engine, and it is still in wide use. It is written in Java, so it runs on many platforms, and it is free (http://www.jclark.com). XT comes with James Clark's nonvalidating parser XP, but you can substitute a different Java parser. Here is a simple example of using XT from the Unix command line to produce HTML:

You'll need to alter your *`CLASSPATH`* environment variable to include the path to where you put the `.jar` files from the XT distribution.

```
CLASSPATH=xt.jar:xp.jar:sax.jar
export CLASSPATH
java  com.jclark.xsl.sax.Driver filename.xml docbook/html/docbook.xsl > output.html
```

If you replace the HTML stylesheet with a formatting object stylesheet, XT will produce a formatting object file. Then you can convert that to PDF using FOP, a formatting object processor available for free from the Apache XML Project (http://xml.apache.org). Here is an example of that two stage processing:

```
CLASSPATH=xt.jar:xp.jar:sax.jar:fop.jar
export CLASSPATH
java  com.jclark.xsl.sax.Driver filename.xml docbook/fo/docbook.xsl > output.fo
java  org.apache.fop.apps.CommandLine output.fo output.pdf
```

As of this writing, some other XSLT processors to choose from include:

- 4XSLT, written in Python, from FourThought LLC (http://www.fourthought.com)
- Sablotron, written in C++, from Ginger Alliance (http://www.gingerall.com)
- Saxon, written in Java, from Michael Kay (http://users.iclway.co.uk/mhkay/saxon)
- Xalan, written in Java, from the Apache XML Project (http://xml.apache.org)
- XML::XSLT,written in Perl, from Geert Josten and Egon Willighagen (http://www.cpan.org)

For print output, these additional tools are available for processing formatting objects:

- XEP (written in Java) from RenderX (http://www.renderx.com).
- PassiveTeX from Sebastian Rahtz (http://users.ox.ac.uk/~rahtz/passivetex/).

A brief introduction to XSL

XSL is both a transformation language and a formatting language. The XSLT transformation part lets you scan through a document's structure and rearrange its content any way you like. You can write out the content using a different set of XML tags, and generate text as needed. For example, you can scan through a document to locate all headings and then insert a generated table of contents at the beginning of the document, at the same time writing out the content marked up as HTML. XSL is also a rich formatting language, letting you apply typesetting controls to all components of your output. With a good formatting backend, it is capable of producing high quality printed pages.

An XSL stylesheet is written using XML syntax, and is itself a well-formed XML document. That makes the basic syntax familiar, and enables an XML processor to check for basic syntax errors. The stylesheet instructions use special element names, which typically begin with `xsl:` to distinguish them from any XML tags you want to appear in the output. The XSL namespace is identified at the top of the stylesheet file. As with other XML, any XSL elements that are not empty will require a closing tag. And some XSL elements have specific attributes that control their behavior. It helps to keep a good XSL reference book handy.

Here is an example of a simple XSL stylesheet applied to a simple XML file to generate HTML output.

Example 4.9. Simple XML file

```
<?xml version="1.0"?>
<document>
<title>Using a mouse</title>
```

```
<para>It's easy to use a mouse. Just roll it
around and click the buttons.</para>
</document>
```

Example 4.10. Simple XSL stylesheet

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

<xsl:template match="document">
  <HTML><HEAD><TITLE>
    <xsl:value-of select="./title"/>
  </TITLE>
  </HEAD>
  <BODY>
    <xsl:apply-templates/>
  </BODY>
  </HTML>
</xsl:template>

<xsl:template match="title">
  <H1><xsl:apply-templates/></H1>
</xsl:template>

<xsl:template match="para">
  <P><xsl:apply-templates/></P>
</xsl:template>

</xsl:stylesheet>
```

Example 4.11. HTML output

```
<HTML>
<HEAD>
<TITLE>Using a mouse</TITLE>
</HEAD>
<BODY>
<H1>Using a mouse</H1>
<P>It's easy to use a mouse. Just roll it
around and click the buttons.</P>
</BODY>
</HTML>
```

XSL processing model

XSL is a template language, not a procedural language. That means a stylesheet specifies a sample of the output, not a sequence of programming steps to generate it. A stylesheet consists of a mixture of output samples with instructions of what to put in each sample. Each bit of output sample and instructions is called a *template*.

In general, you write a template for each element type in your document. That lets you concentrate on handling just one element at a time, and keeps a stylesheet modular. The power of XSL comes from processing the templates recursively. That is, each template handles the processing of its own element, and then calls other templates to process its children, and so on. Since an XML document is always a single root element at the top level that contains all of the nested descendent elements, the XSL templates also start at the top and work their way down through the hierarchy of elements.

Take the DocBook `<para>` paragraph element as an example. To convert this to HTML, you want to wrap the paragraph content with the HTML tags `<p>` and `</p>`. But a DocBook `<para>` can contain any number of in-line DocBook elements marking up the text. Fortunately, you can let other templates take care of those elements, so your XSL template for `<para>` can be quite simple:

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

The `<xsl:template>` element starts a new template, and its `match` attribute indicates where to apply the template, in this case to any `<para>` elements. The template says to output a literal `<p>` string and then execute the `<xsl:apply-templates/>` instruction. This tells the XSL processor to look among all the templates in the stylesheet for any that should be applied to the content of the paragraph. If each template in the stylesheet includes an `<xsl:apply-templates/>` instruction, then all descendents will eventually be processed. When it is through recursively applying templates to the paragraph content, it outputs the `</p>` closing tag.

Context is important

Since you aren't writing a linear procedure to process your document, the context of where and how to apply each modular template is important. The `match` attribute of `<xsl:template>` provides that context for most templates. There is an entire expression language, XPath, for identifying what parts of your document should be handled by each template. The simplest context is just an element name, as in the example above. But you can also specify elements as children of other elements, elements with certain attribute values, the first or last elements in a sequence, and so on. Here is how the DocBook `<formalpara>` element is handled:

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

<xsl:template match="formalpara/title">
  <b><xsl:apply-templates/></b>
  <xsl:text> </xsl:text>
</xsl:template>

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

There are three templates defined, one for the `<formalpara>` element itself, and one for each of its children elements. The `match` attribute value `formalpara/title` in the second template is an XPath expression indicating a `<title>` element that is an immediate child of a `<formalpara>` element. This distinguishes such titles from other `<title>` elements used in DocBook. XPath expressions are the key to controlling how your templates are applied.

In general, the XSL processor has internal rules that apply templates that are more specific before templates that are less specific. That lets you control the details, but also provides a fallback mechanism to a less specific template when you don't supply the full context for every combination of elements. This feature is illustrated by the third template, for `formalpara/para`. By including this template, the stylesheet processes a `<para>` within `<formalpara>` in a special way, in this case by not outputting the HTML `<p>` tags already output by its parent. If this template had not been included, then the processor would have fallen back to the template specified by `match="para"` described above, which would have output a second set of `<p>` tags.

You can also control template context with XSL *modes*, which are used extensively in the DocBook stylesheets. Modes let you process the same input more than once in different ways. A `mode` attribute in an `<xsl:template>` definition adds a specific mode name to that template. When the same mode name is used in `<xsl:apply-templates/>`, it acts as a filter to narrow the selection of templates to only those selected by the `match` expression *and* that have that mode name. This lets you define two different templates for the same element match that are applied under different contexts. For example, there are two templates defined for DocBook `<listitem>` elements:

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

<xsl:template match="listitem" mode="xref">
  <xsl:number format="1"/>
</xsl:template>
```

The first template is for the normal list item context where you want to output the HTML `<li>` tags. The second template is called with `<xsl:apply-templates select="$target" mode="xref"/>` in the context of processing `<xref>` elements. In this case the `select` attribute locates the ID of the specific list item and the `mode` attribute selects the second template, whose effect is to output its item number when it is in an ordered list. Because there are many such special needs when processing `<xref>` elements, it is convenient to define a mode name `xref` to handle them all. Keep in mind that mode settings do *not* automatically get passed down to other templates through `<xsl:apply-templates/>`.

Programming features

Although XSL is template-driven, it also has some features of traditional programming languages. Here are some examples from the DocBook stylesheets.

```
Assign a value to a variable:
<xsl:variable name="refelem" select="name($target)"/>

If statement:
<xsl:if test="$show.comments">
    <i><xsl:call-template name="inline.charseq"/></i>
</xsl:if>

Case statement:
<xsl:choose>
    <xsl:when test="@columns">
        <xsl:value-of select="@columns"/>
    </xsl:when>
    <xsl:otherwise>1</xsl:otherwise>
</xsl:choose>

Call a template by name like a subroutine, passing parameter values and accepting a return value:
<xsl:call-template name="xref.xreflabel">
   <xsl:with-param name="target" select="$target"/>
</xsl:call-template>
```

However, you can't always use these constructs as you do in other programming languages. Variables in particular have very different behavior.

Using variables and parameters

XSL provides two elements that let you assign a value to a name: `<xsl:variable>` and `<xsl:param>`. These share the same name space and syntax for assigning names and values. Both can be referred to using the `$name` syntax.

The main difference between these two elements is that a param's value acts as a default value that can be overridden when a template is called using a `<xsl:with-param>` element as in the last example above.

Here are two examples from DocBook:

```
<xsl:param name="cols">1</xsl:param>
<xsl:variable name="segnum" select="position()"/>
```

In both elements, the name of the parameter or variable is specified with the `name` attribute. So the name of the `param` here is `cols` and the name of the `variable` is `segnum`. The value of either can be supplied in two ways. The value of the first example is the text node "1" and is supplied as the content of the element. The value of the second example is supplied as the result of the expression in its `select` attribute, and the element itself has no content.

The feature of XSL variables that is odd to new users is that once you assign a value to a variable, you cannot assign a new value within the same scope. Doing so will generate an error. So variables are not used as dynamic storage bins they way they are in other languages. They hold a fixed value within their scope of application, and then disappear when the scope is exited. This feature is a result of the design of XSL, which is template-driven and not procedural. This means there is no definite order of processing, so you can't rely on the values of changing variables. To use variables in XSL, you need to understand how their scope is defined.

Variables defined outside of all templates are considered global variables, and they are readable within all templates. The value of a global variable is fixed, and its global value can't be altered from within any template. However, a template can create a local variable of the same name and give it a different value. That local value remains in effect only within the scope of the local variable.

Variables defined within a template remain in effect only within their permitted scope, which is defined as all following siblings and their descendants. To understand such a scope, you have to remember that XSL instructions are true XML elements that are embedded in an XML family hierarchy of XSL elements, often referred to as parents, children, siblings, ancestors and descendants. Taking the family analogy a step further, think of a variable assignment as a piece of advice that you are allowed to give to certain family members. You can give your advice only to your younger siblings (those that follow you) and their descendents. Your older siblings won't listen, neither will your parents or any of your ancestors. To stretch the analogy a bit, it is an error to try to give different advice under the same name to the same group of listeners (in other words, to redefine the variable). Keep in mind that this family is not the elements of your document, but just the XSL instructions in your stylesheet. To help you keep track of such scopes in hand-written stylesheets, it helps to indent nested XSL elements. Here is an edited snippet from the DocBook stylesheet file `pi.xsl` that illustrates different scopes for two variables:

```
 1 <xsl:template name="dbhtml-attribute">
 2 ...
 3    <xsl:choose>
 4       <xsl:when test="$count>count($pis)">
 5          <!-- not found -->
 6       </xsl:when>
 7       <xsl:otherwise>
 8          <xsl:variable name="pi">
 9             <xsl:value-of select="$pis[$count]"/>
10          </xsl:variable>
11          <xsl:choose>
12             <xsl:when test="contains($pi,concat($attribute, '='))">
13             <xsl:variable name="rest" select="substring-after($pi,concat($attribute,'='))"/>
14                <xsl:variable name="quote" select="substring($rest,1,1)"/>
15                <xsl:value-of select="substring-before(substring($rest,2),$quote)"/>
16             </xsl:when>
17             <xsl:otherwise>
18             ...
19             </xsl:otherwise>
```

```
20          </xsl:choose>
21       </xsl:otherwise>
22    </xsl:choose>
23 </xsl:template>
```

The scope of the variable `pi` begins on line 8 where it is defined in this template, and ends on line 20 when its last sibling ends.[17] The scope of the variable `rest` begins on line 13 and ends on line 15. Fortunately, line 15 outputs an expression using the value before it goes out of scope.

What happens when an `<xsl:apply-templates/>` element is used within the scope of a local variable? Do the templates that are applied to the document children get the variable? The answer is no. The templates that are applied are not actually within the scope of the variable. They exist elsewhere in the stylesheet and are not following siblings or their descendants.

To pass a value to another template, you pass a parameter using the `<xsl:with-param>` element. This parameter passing is usually done with calls to a specific named template using `<xsl:call-template>`, although it works with `<xsl:apply-templates>` too. That's because the called template must be expecting the parameter by defining it using a `<xsl:param>` element with the same parameter name. Any passed parameters whose names are not defined in the called template are ignored.

Here is an example of parameter passing from `docbook.xsl`:

```
<xsl:call-template name="head.content">
   <xsl:with-param name="node" select="$doc"/>
</xsl:call-template>
```

Here a template named `head.content` is being called and passed a parameter named `node` whose content is the value of the `$doc` variable in the current context. The top of that template looks like this:

```
<xsl:template name="head.content">
   <xsl:param name="node" select="."/>
```

The template is expecting the parameter because it has a `<xsl:param>` defined with the same name. The value in this definition is the default value. This would be the parameter value used in the template if the template was called without passing that parameter.

Generating HTML output.

You generate HTML from your DocBook XML files by applying the HTML version of the stylesheets. This is done by using the HTML driver file `docbook/html/docbook.xsl` as your stylesheet. That is the master stylesheet file that uses `<xsl:include>` to pull in the component files it needs to assemble a complete stylesheet for producing HTML.

The way the DocBook stylesheet generates HTML is to apply templates that output a mix of text content and HTML elements. Starting at the top level in the main file `docbook.xsl`:

```
<xsl:template match="/">
  <xsl:variable name="doc" select="*[1]"/>
  <html>
  <head>
    <xsl:call-template name="head.content">
      <xsl:with-param name="node" select="$doc"/>
    </xsl:call-template>
  </head>
```

---

[17] Technically, the scope extends to the end tag of the parent of the `<xsl:variable>` element. That is effectively the last sibling.

```
  <body>
    <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>
```

This template matches the root element of your input document, and starts the process of recursively applying templates. It first defines a variable named `doc` and then outputs two literal HTML elements `<html>` and `<head>`. Then it calls a named template `head.content` to process the content of the HTML `<head>`, closes the `<head>` and starts the `<body>`. There it uses `<xsl:apply-templates/>` to recursively process the entire input document. Then it just closes out the HTML file.

Simple HTML elements can generated as literal elements as shown here. But if the HTML being output depends on the context, you need something more powerful to select the element name and possibly add attributes and their values. Here is a fragment from `sections.xsl` that shows how a heading tag is generated using the `<xsl:element>` and `<xsl:attribute>` elements:

```
 1 <xsl:element name="h{$level}">
 2   <xsl:attribute name="class">title</xsl:attribute>
 3   <xsl:if test="$level<3">
 4     <xsl:attribute name="style">clear: all</xsl:attribute>
 5   </xsl:if>
 6   <a>
 7     <xsl:attribute name="name">
 8       <xsl:call-template name="object.id"/>
 9     </xsl:attribute>
10     <b><xsl:copy-of select="$title"/></b>
11   </a>
12 </xsl:element>
```

This whole example is generating a single HTML heading element. Line 1 begins the HTML element definition by identifying the name of the element. In this case, the name is an expression that includes the variable `$level` passed as a parameter to this template. Thus a single template can generate `<h1>`, `<h2>`, etc. depending on the context in which it is called. Line 2 defines a `class="title"` attribute that is added to this element. Lines 3 to 5 add a `style="clear all"` attribute, but only if the heading level is less than 3. Line 6 opens an `<a>` anchor element. Although this looks like a literal output string, it is actually modified by lines 7 to 9 that insert the `name` attribute into the `<a>` element. This illustrates that XSL is managing output elements as active element nodes, not just text strings. Line 10 outputs the text of the heading title, also passed as a parameter to the template, enclosed in HTML boldface tags. Line 11 closes the anchor tag with the literal `</a>` syntax, while line 12 closes the heading tag by closing the element definition. Since the actual element name is a variable, it couldn't use the literal syntax.

As you follow the sequence of nested templates processing elements, you might be wondering how the ordinary text of your input document gets to the output. In the file `docbook.xsl` you will find this template that handles any text not processed by any other template:

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

This template's body consists of the "value" of the text node, which is just its text. In general, all XSL processors have some built-in templates to handle any content for which your stylesheet doesn't supply a matching template. This template serves the same function but appears explicitly in the stylesheet.

Generating formatting objects.

You generate formatting objects from your DocBook XML files by applying the fo version of the stylesheets. This is done by using the fo driver file `docbook/fo/docbook.xsl` as your stylesheet. That is the master stylesheet file that uses `<xsl:include>` to pull in the component files it needs to assemble a complete stylesheet for producing formatting objects. Generating a formatting objects file is only half the process of producing typeset output. You also need a formatting object processor such as the Apache XML Project's FOP as described in an earlier section.

The DocBook fo stylesheet works in a similar manner to the HTML stylesheet. Instead of outputting HTML tags, it outputs text marked up with `<fo:`*`something`*`>` tags. For example, to indicate that some text should be kept in-line and typeset with a monospace font, it might look like this:

```
<fo:inline-sequence font-family="monospace">/usr/man</fo:inline-sequence>
```

The templates in `docbook/fo/inline.xsl` that produce this output for a DocBook `<filename>` element look like this:

```
<xsl:template match="filename">
  <xsl:call-template name="inline.monoseq"/>
</xsl:template>

<xsl:template name="inline.monoseq">
  <xsl:param name="content">
    <xsl:apply-templates/>
  </xsl:param>
  <fo:inline-sequence font-family="monospace">
    <xsl:copy-of select="$content"/>
  </fo:inline-sequence>
</xsl:template>
```

There are dozens of fo tags and attributes specified in the XSL standard. It is beyond the scope of this document to cover how all of them are used in the DocBook stylesheets. Fortunately, this is only an intermediate format that you probably won't have to deal with very much directly unless you are writing your own stylesheets.

Customizing DocBook XSL stylesheets

The DocBook XSL stylesheets are written in a modular fashion. Each of the HTML and FO stylesheets starts with a driver file that assembles a collection of component files into a complete stylesheet. This modular design puts similar things together into smaller files that are easier to write and maintain than one big stylesheet. The modular stylesheet files are distributed among four directories:

common/

contains code common to both stylesheets, including localization data

fo/

a stylesheet that produces XSL FO result trees

html/

a stylesheet that produces HTML/XHTML result trees

lib/

contains schema-independent functions

The driver files for each of HTML and FO stylesheets are `html/docbook.xsl` and `fo/docbook.xsl`, respectively. A driver file consists mostly of a bunch of `<xsl:include>` instructions to pull in the component templates, and then defines some top-level templates. For example:

```
<xsl:include href="../VERSION"/>
<xsl:include href="../lib/lib.xsl"/>
<xsl:include href="../common/l10n.xsl"/>
<xsl:include href="../common/common.xsl"/>
<xsl:include href="autotoc.xsl"/>
<xsl:include href="lists.xsl"/>
<xsl:include href="callout.xsl"/>
...
<xsl:include href="param.xsl"/>
<xsl:include href="pi.xsl"/>
```

The first four modules are shared with the FO stylesheet and are referenced using relative pathnames to the common directories. Then the long list of component stylesheets starts. Pathnames in include statements are always taken to be relative to the including file. Each included file must be a valid XSL stylesheet, which means its root element must be `<xsl:stylesheet>`.

Stylesheet inclusion vs. importing

XSL actually provides two inclusion mechanisms: `<xsl:include>` and `<xsl:import>`. Of the two, `<xsl:include>` is the simpler. It treats the included content as if it were actually typed into the file at that point, and doesn't give it any more or less precedence relative to the surrounding text. It is best used when assembling dissimilar templates that don't overlap what they match. The DocBook driver files use this instruction to assemble a set of modules into a stylesheet.

In contrast, `<xsl:import>` lets you manage the precedence of templates and variables. It is the preferred mode of customizing another stylesheet because it lets you override definitions in the distributed stylesheet with your own, without altering the distribution files at all. You simply import the whole stylesheet and add whatever changes you want.

The precedence rules for import are detailed and rigorously defined in the XSL standard. The basic rule is that any templates and variables in the importing stylesheet have precedence over equivalent templates and variables in the imported stylesheet. Think of the imported stylesheet elements as a fallback collection, to be used only if a match is not found in the current stylesheet. You can customize the templates you want to change in your stylesheet file, and let the imported stylesheet handle the rest.

Note

Customizing a DocBook XSL stylesheet is the opposite of customizing a DocBook DTD. When you customize a DocBook DTD, the rules of XML and SGML dictate that the *first* of any duplicate declarations wins. Any subsequent declarations of the same element or entity are ignored. The architecture of the DTD provides slots for inserting your own custom declarations early enough in the DTD for them to override the standard declarations. In contrast, customizing an XSL stylesheet is simpler because your definitions have precedence over imported ones.

You can carry modularization to deeper levels because module files can also include or import other modules. You'll need to be careful to maintain the precedence that you want as the modules get rolled up into a complete stylesheet.

Customizing with `<xsl:import>`

There is currently one example of customizing with `<xsl:import>` in the HTML version of the DocBook stylesheets. The `xtchunk.xsl` stylesheet modifies the HTML processing to output many smaller HTML files rather than a single large file per input document. It uses XSL extensions defined only in the XSL processor **XT**. In the driver file `xtchunk.xsl`, the first instruction is `<xsl:import href="docbook.xsl"/>`. That instruction imports the original driver file, which in turn uses many `<xsl:include>` instructions to include all the modules. That single import instruction gives the new stylesheet the complete set of DocBook templates to start with.

After the import, `xtchunk.xsl` redefines some of the templates and adds some new ones. Here is one example of a redefined template:

```
Original template in autotoc.xsl
<xsl:template name="href.target">
  <xsl:param name="object" select="."/>
  <xsl:text>#</xsl:text>
  <xsl:call-template name="object.id">
    <xsl:with-param name="object" select="$object"/>
  </xsl:call-template>
</xsl:template>

New template in xtchunk.xsl
<xsl:template name="href.target">
  <xsl:param name="object" select="."/>
  <xsl:variable name="ischunk">
    <xsl:call-template name="chunk">
      <xsl:with-param name="node" select="$object"/>
    </xsl:call-template>
  </xsl:variable>

  <xsl:apply-templates mode="chunk-filename" select="$object"/>

  <xsl:if test="$ischunk='0'">
    <xsl:text>#</xsl:text>
    <xsl:call-template name="object.id">
      <xsl:with-param name="object" select="$object"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
```

The new template handles the more complex processing of HREFs when the output is split into many HTML files. Where the old template could simply output *`#object.id`*, the new one outputs *`filename#object.id`*.

Setting stylesheet variables

You may not have to define any new templates, however. The DocBook stylesheets are parameterized using XSL variables rather than hard-coded values for many of the formatting features. Since the `<xsl:import>` mechanism also lets you redefine global variables, this gives you an easy way to customize many features of the DocBook stylesheets. Over time, more features will be parameterized to permit customization. If you find hardcoded values in the stylesheets that would be useful to customize, please let the maintainer know.

Near the end of the list of includes in the main DocBook driver file is the instruction `<xsl:include href="param.xsl"/>`. The `param.xsl` file is the most important module for customizing a DocBook XSL stylesheet. This module contains no templates, only definitions of stylesheet variables. Since these variables are defined outside of any template, they are global variables and apply to the entire stylesheet. By redefining these variables in an importing stylesheet, you can change the behavior of the stylesheet.

To create a customized DocBook stylesheet, you simply create a new stylesheet file such as `mystyle.xsl` that imports the standard stylesheet and adds your own new variable definitions. Here is an example of a complete custom stylesheet that changes the depth of sections listed in the table of contents from two to three:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version='1.0'
                xmlns="http://www.w3.org/TR/xhtml1/transitional"
                exclude-result-prefixes="#default">

<xsl:import href="docbook.xsl"/>

<xsl:variable name="toc.section.depth">3</xsl:variable>
<!-- Add other variable definitions here -->

</xsl:stylesheet>
```

Following the opening stylesheet element are the import instruction and one variable definition. The variable `toc.section.depth` was defined in `param.xsl` with value "2", and here it is defined as "3". Since the importing stylesheet takes precedence, this new value is used. Thus documents processed with `mystyle.xsl` instead of `docbook.xsl` will have three levels of sections in the tables of contents, and all other processing will be the same.

Use the list of variables in `param.xsl` as your guide for creating a custom stylesheet. If the changes you want are controlled by a variable there, then customizing is easy.

Writing your own templates

If the changes you want are more extensive than what is supported by variables, you can write new templates. You can put your new templates directly in your importing stylesheet, or you can modularize your importing stylesheet as well. You can write your own stylesheet module containing a collection of templates for processing lists, for example, and put them in a file named `mylists.xsl`. Then your importing stylesheet can pull in your list templates with a `<xsl:include href="mylists.xsl"/>` instruction. Since your included template definitions appear after the main import instruction, your templates will take precedence.

You'll need to make sure your new templates are compatible with the remaining modules, which means:

- Any named templates should use the same name so calling templates in other modules can find them.
- Your template set should process the same elements matched by templates in the original module, to ensure complete coverage.
- Include the same set of `<xsl:param>` elements in each template to interface properly with any calling templates, although you can set different values for your parameters.
- Any templates that are used like subroutines to return a value should return the same data type.

Writing your own driver

Another approach to customizing the stylesheets is to write your own driver file. Instead of using `<xsl:import href="docbook.xsl"/>`, you copy that file to a new name and rewrite any of the `<xsl:include/>` instructions to assemble a custom collection of stylesheet modules. One reason to do this is to speed up processing by reducing the size of the stylesheet. If you are using a customized DocBook DTD that omits many elements you never use, you might be able to omit those modules of the stylesheet.

Localization

The DocBook stylesheets include features for localizing generated text, that is, printing any generated text in a language other than the default English. In general, the stylesheets will switch to the language identified by a `lang` attribute when processing elements in your documents. If your documents use the `lang` attribute, then you don't need to customize the stylesheets at all for localization.

As far as the stylesheets go, a `lang` attribute is inherited by the descendents of a document element. The stylesheet searches for a `lang` attribute using this XPath expression:

```
<xsl:variable name="lang-attr"
         select="($target/ancestor-or-self::*/@lang
                  |$target/ancestor-or-self::*/@xml:lang)[last()]"/>
```

This locates the attribute on the current element or its most recent ancestor. Thus a `lang` attribute is in effect for an element and all of its descendents, unless it is reset in one of those descendents. If you define it in only your document root element, then it applies to the whole document:

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.0//EN" "docbook.dtd">
<book lang="fr">
...
</book>
```

When text is being generated, the stylesheet checks the most recent `lang` attribute and looks up the generated text strings for that language in a localization XML file. These are located in the `common` directory of the stylesheets, one file per language. Here is the top of the file `fr.xml`:

```
<localization language="fr">

<gentext key="abstract"                 text="R&#x00E9;sum&#x00E9;"/>
<gentext key="answer"                   text="R:"/>
<gentext key="appendix"                 text="Annexe"/>
<gentext key="article"                  text="Article"/>
<gentext key="bibliography"             text="Bibliographie"/>
...
```

The stylesheet templates use the gentext key names, and then the stylesheet looks up the associated text value when the document is processed with that lang setting. The file `l10n.xml` (note the `.xml` suffix) lists the filenames of all the supported languages.

You can also create a custom stylesheet that sets the language. That might be useful if your documents don't make appropriate use of the `lang` attribute. The module `l10n.xsl` defines two global variables that can be overridden with an importing stylesheet as described above. Here are their default definitions:

```
<xsl:variable name="l10n.gentext.language"></xsl:variable>
<xsl:variable name="l10n.gentext.default.language">en</xsl:variable>
```

The first one sets the language for all elements, regardless of an element's `lang` attribute value. The second just sets a default language for any elements that haven't got a `lang` setting of their own (or their ancestors).

5

Customizing DocBook

$Revision: 1.2 $
$Date: 2002/04/18 22:06:46 $

For the applications you have in mind, DocBook “out of the box” may not be exactly what you need. Perhaps you need additional inline elements or perhaps you want to remove elements that you never want your authors to use. By design, DocBook makes this sort of customization easy.

This chapter explains how to make your own customization layer. You might do this in order to:

- Add new elements
- Remove elements
- Change the structure of existing elements
- Add new attributes
- Remove attributes
- Broaden the range of values allowed in an attribute
- Narrow the range of values in an attribute to a specific list or a fixed value

You can use customization layers to extend DocBook or subset it. Creating a DTD that is a strict subset of DocBook means that all of your instances are still completely valid DocBook instances, which may be important to your tools and stylesheets, and to other people with whom you share documents. An extension adds new structures, or changes the DTD in a way that is not compatible with DocBook. Extensions can be very useful, but might have a great impact on your environment.

Customization layers can be as small as restricting an attribute value or as large as adding an entirely different hierarchy on top of the inline elements.

Should You Do This?

Changing a DTD can have a wide-ranging impact on the tools and stylesheets that you use. It can have an impact on your authors and on your legacy documents. This is especially true if you make an extension. If you rely on your support staff to install and maintain your authoring and publishing tools, check with them before you invest a lot of time modifying the DTD. There may be additional issues that are outside your immediate control. Proceed with caution.

That said, DocBook is designed to be easy to modify. This chapter assumes that you are comfortable with SGML/XML DTD syntax, but the examples presented should be a good springboard to learning the syntax if it's not already familiar to you.

If You Change DocBook, It's Not DocBook Anymore!

The DocBook DTD is usually referenced by its public identifier:

```
-//OASIS//DTD DocBook V3.1//EN
```

Previous versions of DocBook, V3.0 and the V2 variants, used the owner identifier Davenport, rather than OASIS.

If you make any changes to the structure of the DTD, it is imperative that you alter the public identifier that you use for the DTD and the modules you changed. The license agreement under which DocBook is distributed gives you complete freedom to change, modify, reuse, and generally hack the DTD in any way you want, except that you must not call your alterations “DocBook.”

You should change both the owner identifier and the description. The original DocBook formal public identifiers use the following syntax:

-//OASIS//*text-class* DocBook *description* V*version*//EN

Your own formal public identifiers should use the following syntax in order to record their DocBook derivation:

-//*your-owner-ID*//*text-class* DocBook V*version*-Based [Subset|Extension|Variant] *your-descrip-and-version*//*lang*

For example:

```
-//O'Reilly//DTD DocBook V3.0-Based Subset V1.1//EN
```

If your DTD is a proper subset, you can advertise this status by using the `Subset` keyword in the description. If your DTD contains any markup model extensions, you can advertise this status by using the `Extension` keyword. If you'd rather not characterize your variant specifically as a subset or an extension, you can leave out this field entirely, or, if you prefer, use the `Variant` keyword.

There is only one file that you may change without changing the public identifier: `dbgenent.mod`. And you can add only entity and notation declarations to that file. (You can add anything you want, naturally, but if you add anything other than entity and notation declarations, you must change the public identifier!)

Customization Layers

SGML and XML DTDs are really just collections of declarations. These declarations are stored in one or more files. A complete DTD is formed by combining these files together logically. Parameter entities are used for this purpose. Consider the following fragment:

```
<!ENTITY % dbpool SYSTEM "dbpool.mod"> ❶

<!ENTITY % dbhier SYSTEM "dbhier.mod"> ❷

%dbpool;                               ❸

%dbhier;                               ❹
```

① This line declares the parameter entity `dbpool` and associates it with the file `dbpool.mod`.

② This line declares the parameter entity `dbhier` and associates it with the file `dbhier.mod`.

③ This line references `dbpool`, which loads the file `dbpool.mod` and inserts its content here.

④ Similarly, this line loads `dbhier.mod`.

It is an important feature of DTD parsing that entity declarations can be repeated. If an entity is declared more than once, then the *first* declaration is used. Given this fragment:

```
<!ENTITY foo "Lenny">
<!ENTITY foo "Norm">
```

The replacement text for `&foo;` is “Lenny.”

These two notions, that you can break a DTD into modules referenced with parameter entities and that the first entity declaration is the one that counts, are used to build “customization layers.” With customization layers you can write a DTD that references some or all of DocBook, but adds your own modifications. Modifying the DTD this way means that you never have to edit the DocBook modules directly, which is a tremendous boon to maintaining your modules. When the next release of DocBook comes out, you usually only have to make changes to your customization layer and your modification will be back in sync with the new version.

Customization layers work particularly well in DocBook because the base DTD makes extensive use of parameter entities that can be redefined.

Understanding DocBook Structure

DocBook is a large and, at first glance, fairly complex DTD. Much of the apparent complexity is caused by the prolific use of parameter entities. This was an intentional choice on the part of the maintainers, who traded “raw readability” for customizability. This section provides a general overview of the structure of the DTD. After you understand it, DocBook will probably seem much less complicated.

DocBook Modules

DocBook is composed of seven primary modules. These modules decompose the DTD into large, related chunks. Most modifications are restricted to a single chunk.

Figure 5.1 shows the module structure of DocBook as a flowchart.

Figure 5.1. Structure of the DocBook DTD

The modules are:

`docbook.dtd`

The main driver file. This module declares and references the other top-level modules.

`dbhier.mod`

The hierarchy. This module declares the elements that provide the hierarchical structure of DocBook (sets, books, chapters, articles, and so on).

Changes to this module alter the top-level structure of the DTD. If you want to write a DocBook-derived DTD with a different structure (something other than a book), but with the same paragraph and inline-level elements, you make most of your changes in this module.

`dbpool.mod`

The information pool. This module declares the elements that describe content (inline elements, bibliographic data, block quotes, sidebars, and so on) but are not part of the large-scale hierarchy of a document. You can incorporate these elements into an entirely different element hierarchy.

The most common reason for changing this module is to add or remove inline elements.

`dbnotn.mod`

The notation declarations. This module declares the notations used by DocBook.

This module can be changed to add or remove notations.

`dbcent.mod`

The character entities. This module declares and references the ISO entity sets used by DocBook.

Changes to this module can add or remove entity sets.

`dbgenent.mod`

The general entities. This is a place where you can customize the general entities available in DocBook instances.

This is the place to add, for example, boiler plate text, logos for institutional identity, or additional notations understood by your local processing system.

`cals-tbl.dtd`

The CALS Table Model. CALS is an initiative by the United States Department of Defense to standardize the document types used across branches of the military. The CALS table model, published in MIL-HDBK-28001, was for a long time the most widely supported SGML table model (one might now argue that the HTML table model is more widely supported by some definitions of “widely supported”). In any event, it is the table model used by DocBook.

DocBook predates the publication of the OASIS Technical Resolution TR 9503:1995[i], which defines an industry standard exchange table model and thus incorporates the *full* CALS Table Model.

---

[i] http://www.oasis-open.org/html/a503.htm

Most changes to the CALS table model can be accomplished by modifying parameter entities in `dbpool.mod`; changing this DTD fragment is strongly discouraged. If you want to use a different table model, remove this one and add your own.

`*.gml`

The ISO standard character entity sets. These entity sets are not actually part of the official DocBook distribution, but are referenced by default.

There are some additional modules, initially undefined, that can be inserted at several places for "redeclaration." This is described in more detail in the section called "Removing Admonitions from Table Entries"."

DocBook Parameterization

Customization layers are possible because DocBook has been extensively parameterized so that it is possible to make any changes that might be desired without ever editing the actual distributed modules. The parameter entities come in several flavors:

`%*.class;`

Classes group elements of a similar type: for example all the lists are in the `%list.class;`.

If you want to add a new kind of something (a new kind of list or a new kind of verbatim environment, for example), you generally want to add the name of the new element to the appropriate class.

`%*.mix;`

Mixtures are collections of classes that appear in content models. For example, the content model of the `Example` element includes `%example.mix;`. Not every element's content model is a single mixture, but elements in the same class tend to have the same mixture in their content model.

If you want to change the content model of some class of elements (lists or admonitions, perhaps), you generally want to change the definition of the appropriate mixture.

`%*.module;`

The `%*.module;` parameter entities control marked sections around individual elements and their attribute lists. For example, the element and attribute declarations for `Abbrev` occur within a marked section delimited by `%abbrev.module;`.

If you want to remove or redefine an element or its attribute list, you generally want to change its module marked section to `IGNORE` and possibly add a new definition for it in your customization layer.

`%*.element;`

The `%*.element;` parameter entities were introduced in DocBook V3.1; they control marked sections around individual element declarations.

`%*.attlist;`

The `%*.attlist;` parameter entities were introduced in DocBook V3.1; they control marked sections around individual attribute list declarations.

`%*.inclusion;`, `%*.exclusion;`

These parameter entities control the inclusion and exclusion markup in element declarations.

Changing these declarations allows you to make global changes to the inclusions and exclusions in the DTD.

`%local.*;`

The `%local.*;` parameter entities are a local extension mechanism. You can add markup to most entity declarations simply by declaring the appropriate local parameter entity.

The General Structure of Customization Layers

Although customization layers vary in complexity, most of them have the same general structure as other customization layers of similar complexity.

In the most common case, you probably want to include the entire DTD, but you want to make some small changes. These customization layers tend to look like this:

```
❶
 Overrides of Entity Declarations Here
❷
 <!ENTITY % orig-docbook "-//OASIS//DTD DocBook V3.1//EN">
 %orig-docbook;
❸
 New/Modified Element and Attribute Declarations Here
```

❶ Declare new values for parameter entities (`%local.*;`, `%*.element;`, `%*.attlist;`) that you wish to modify.

❷ Include the entire DocBook DTD by parameter entity reference.

❸ Add new element and attribute declarations for any elements that you added to the DTD.

In slightly more complex customization layers, the changes that you want to make are influenced by the interactions between modules. In these cases, rather than including the whole DTD at once, you include each of the modules separately, perhaps with entity or element declarations between them:

```

Overrides of Most Entity Declarations Here

<!ENTITY % orig-pool "-//OASIS//ELEMENTS DocBook Information Pool V3.1//EN">
%orig-pool;

Overrides of Document Hierarchy Entities Here

<!ENTITY % orig-hier "-//OASIS//ELEMENTS DocBook Document Hierarchy V3.1//EN">
%orig-hier;

New/Modified Element and Attribute Declarations Here

<!ENTITY % orig-notn "-//OASIS//ENTITIES DocBook Notations V3.1//EN">
%orig-notn;
```

```
<!ENTITY % orig-cent "-//OASIS//ENTITIES DocBook Character Entities V3.1//EN">
%orig-cent;

<!ENTITY % orig-gen  "-//OASIS//ENTITIES DocBook Additional General Entities V3.1//EN">
%orig-gen;
```

Finally, it's worth noting that in the rare case in which you need certain kinds of very simple, “one-off” customizations, you can do them in the document subset:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN" [
Overrides of Entity Declarations Here
New/Modified Element and Attribute Declarations Here
]>
<book>...</book>
```

Writing, Testing, and Using a Customization Layer

The procedure for creating, testing, and using a customization layer is always about the same. In this section, we'll go through the process in some detail. The rest of the sections in this chapter describe a range of useful customization layers.

Deciding What to Change

If you're considering writing a customization layer, there must be something that you want to change. Perhaps you want to add an element or attribute, remove one, or change some other aspect of the DTD.

Adding an element, particularly an inline element, is one possibility. If you're writing documentation about an object-oriented system, you may have noticed that DocBook provides `ClassName` but not `MethodName`. Suppose you want to add `MethodName`?

Deciding How to Change a Customization Layer

Figuring out what to change may be the hardest part of the process. The organization of the parameter entities is quite logical, and, bearing in mind the organization described in the section called “Understanding DocBook Structure”,” finding something similar usually provides a good model for new changes.

`MethodName` is similar to `ClassName`, so `ClassName` is probably a good model. `ClassName` is an inline element, not a hierarchy element, so it's in `dbpool.mod`. Searching for “classname” in `dbpool.mod` reveals:

```
<!ENTITY % local.tech.char.class "">
<!ENTITY % tech.char.class
        "Action|Application|ClassName|Command|ComputerOutput
        |Database|Email|EnVar|ErrorCode|ErrorName|ErrorType|Filename
        |Function|GUIButton|GUIIcon|GUILabel|GUIMenu|GUIMenuItem
        |GUISubmenu|Hardware|Interface|InterfaceDefinition|KeyCap
        |KeyCode|KeyCombo|KeySym|Literal|Constant|Markup|MediaLabel
        |MenuChoice|MouseButton|MsgText|Option|Optional|Parameter
        |Prompt|Property|Replaceable|ReturnValue|SGMLTag|StructField
        |StructName|Symbol|SystemItem|Token|Type|UserInput|VarName
        %local.tech.char.class;">
```

Searching further reveals the element and attribute declarations for `ClassName`.

It would seem (and, in fact, it is the case) that adding `MethodName` can be accomplished by adding it to the local extension mechanism for `%tech.char.class;`, namely `%local.tech.char.class;`, and adding element and attribute declarations for it. A customization layer that does this can be seen in Example 5.1.

Example 5.1. Adding MethodName with a Customization Layer

```
<!ENTITY % local.tech.char.class "|MethodName">          ①


<!-- load DocBook -->                                    ②
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;


<!ELEMENT MethodName - - ((%smallcptr.char.mix;)+)  ③>

<!ATTLIST MethodName                                     ④
        %common.attrib;
        %classname.role.attrib;
        %local.classname.attrib;
>
```

① Declare the appropriate parameter entity (these are described in the section called “DocBook Parameterization””). The declaration in your customization layer is encountered first, so it overrides the definition in the DocBook DTD (all the local classes are defined as empty in the DTD).

② Use a parameter entity to load the entire DocBook DTD.

③ Add an element declaration for the new element. The content model for this element is taken directly from the content model of `ClassName`.

④ Add an attribute list declaration for the new element. These are the same attributes as `ClassName`.

Using Your Customization Layer

In order to use the new customization layer, you must save it in a file, for example `mydocbk.dtd`, and then you must use the new DTD in your document.

The simplest way to use the new DTD is to point to it with a system identifier:

```
<!DOCTYPE chapter SYSTEM "/path/to/mydocbk.dtd">
<chapter><title>My Chapter</title>
<para>
The Java <classname>Math</classname> class provides a
<methodname>abs</methodname> method to compute absolute value of a number.
</para>
</chapter>
```

If you plan to use your customization layer in many documents, or exchange it with interchange partners, consider giving your DTD its own public identifier, as described in the section called “If You Change DocBook, It's Not DocBook Anymore!””

In order to use the new public identifier, you must add it to your catalog:

```
PUBLIC "-//Your Organization//DTD DocBook V3.1-Based Extension V1.0//EN"
       "/share/sgml/mydocbk.dtd"
```

and use that public identifier in your documents:

```
<!DOCTYPE chapter
  PUBLIC "-//Your Organization//DTD DocBook V3.1-Based Extension V1.0//EN">
<chapter><title>My Chapter</title>
<para>
The Java <classname>Math</classname> class provides a
<methodname>abs</methodname> method to compute absolute value of a number.
</para>
</chapter>
```

If you're using XML, remember that you must provide a system identifier that satisfies the requirements of a Uniform Resource Identifier (URI).

Testing Your Work

DTDs, by their nature, contain many complex, interrelated elements. Whenever you make a change to the DTD, it's always wise to use a validating parser to double-check your work. A parser like **nsgmls** from James Clark's SP can identify elements (attributes, parameter entities) that are declared but unused, as well as ones that are used but undeclared.

A comprehensive test can be accomplished with **nsgmls** using the `-wall` option. Create a simple test document and run:

```
nsgmls ①-sv ②-wall test.sgm
```

① The `-s` option tells **nsgmls** to suppress its normal output (it will still show errors, if there are any). The `-v` option tells **nsgmls** to print its version number; this ensures that you always get *some* output, even if there are no errors.

② The `-wall` option tells **nsgmls** to provide a comprehensive list of all errors and warnings. You can use less verbose, and more specific options instead; for example, `-wundefined` to flag undefined elements or `-wunused-param` to warn you about unused parameter entities. The **nsgmls** documentation provides a complete list of warning types.

DocBook V3.1 Warnings

If you run the preceding command over DocBook V3.1, you'll discover one warning generated by the DTD:

```
nsgmls:I: SP version "1.3"
nsgmls:cals-tbl.dtd:314:37:W: content model is mixed but does not allow #PCDATA everywhere
```

This is not truly an error in the DTD, and can safely be ignored. The warning is caused by “pernicious mixed content” in the content model of DocBook's `Entry` element. See the `Entry` reference page for a complete discussion.

Removing Elements

DocBook has a large number of elements. In some authoring environments, it may be useful or necessary to remove some of these elements.

Removing MsgSet

`MsgSet` is a favorite target. It has a complex internal structure designed for describing interrelated error messages, especially on systems that may exhibit messages from several different components. Many technical documents can do without it, and removing it leaves one less complexity to explain to your authors.

Example 5.2 shows a customization layer that removes the `MsgSet` element from DocBook:

Example 5.2. Removing MsgSet

```
<!ENTITY % compound.class "Procedure|SideBar"> ①

<!ENTITY % msgset.content.module "IGNORE">     ②
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

① Remove `MsgSet` from the `%compound.class;`. This is the only place in the DTD where `MsgSet` is referenced.

② Exclude the definition of `MsgSet` and all of its subelements from the DTD.

Removing Computer Inlines

DocBook contains a large number of computer inlines. The DocBook inlines define a domain-specific vocabulary. If you're working in another domain, many of them may be unnecessary. You can remove a bunch of them by redefining the `%tech.char.class;` parameter entity and then excluding the declarations for the elements removed. The initial definition of `%tech.char.class;` is:

```
<!ENTITY % tech.char.class
    "Action|Application|ClassName|Command|ComputerOutput
    |Database|Email|EnVar|ErrorCode|ErrorName|ErrorType|Filename
    |Function|GUIButton|GUIIcon|GUILabel|GUIMenu|GUIMenuItem
    |GUISubmenu|Hardware|Interface|InterfaceDefinition|KeyCap
    |KeyCode|KeyCombo|KeySym|Literal|Markup|MediaLabel|MenuChoice
    |MouseButton|MsgText|Option|Optional|Parameter|Prompt|Property
    |Replaceable|ReturnValue|SGMLTag|StructField|StructName
    |Symbol|SystemItem|Token|Type|UserInput
    %local.tech.char.class;">
```

When examining this list, it seems that you can delete all of the inlines except, perhaps, `Application`, `Command`, `Email`, `Filename`, `Literal`, `Replaceable`, `Symbol`, and `SystemItem`. The following customization layer removes them.

Example 5.3. Removing Computer Inlines

```
<!ENTITY % tech.char.class
        "Application|Command|Email|Filename|Literal
        |Replaceable|Symbol|SystemItem">
<!ENTITY % action.module "IGNORE">
<!ENTITY % classname.module "IGNORE">
<!ENTITY % computeroutput.module "IGNORE">
<!ENTITY % database.module "IGNORE">
<!ENTITY % envar.module "IGNORE">
<!ENTITY % errorcode.module "IGNORE">
<!ENTITY % errorname.module "IGNORE">
<!ENTITY % errortype.module "IGNORE">
<!--<!ENTITY % function.module "IGNORE">-->
<!ENTITY % guibutton.module "IGNORE">
<!ENTITY % guiicon.module "IGNORE">
<!ENTITY % guilabel.module "IGNORE">
<!ENTITY % guimenu.module "IGNORE">
<!ENTITY % guimenuitem.module "IGNORE">
<!ENTITY % guisubmenu.module "IGNORE">
<!ENTITY % hardware.module "IGNORE">
<!ENTITY % interface.module "IGNORE">
<!ENTITY % interfacedefinition.module "IGNORE">
<!--<!ENTITY % keycap.module "IGNORE">-->
<!ENTITY % keycode.module "IGNORE">
<!--<!ENTITY % keycombo.module "IGNORE">-->
<!--<!ENTITY % keysym.module "IGNORE">-->
<!ENTITY % markup.module "IGNORE">
<!ENTITY % medialabel.module "IGNORE">
<!ENTITY % menuchoice.module "IGNORE">
<!--<!ENTITY % mousebutton.module "IGNORE">-->
<!--<!ENTITY % msgtext.module "IGNORE">-->
<!--<!ENTITY % option.module "IGNORE">-->
<!--<!ENTITY % optional.module "IGNORE">-->
<!--<!ENTITY % parameter.module "IGNORE">-->
<!ENTITY % prompt.module "IGNORE">
<!ENTITY % property.module "IGNORE">
<!ENTITY % returnvalue.module "IGNORE">
<!ENTITY % sgmltag.module "IGNORE">
<!ENTITY % structfield.module "IGNORE">
<!ENTITY % structname.module "IGNORE">
<!ENTITY % token.module "IGNORE">
<!ENTITY % type.module "IGNORE">
<!ENTITY % userinput.module "IGNORE">
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

Initially we removed several more elements from `%tech.char.class;` (`%function.module;`, `%keycap.module;`), but using the testing procedure described in the section called “Testing Your Work”,” we discovered that these elements are used in other content models. Because they are used in other content modules, they cannot simply be removed from the DTD by deleting them from `% tech.char.class;`. Even though they can't be deleted outright, we've taken them out of most inline contexts.

It's likely that a customization layer that removed this many technical inlines would also remove some larger technical structures (`MsgSet`, `FuncSynopsis`), which allows you to remove additional elements from the DTD.

Removing Synopsis Elements

Another possibility is removing the complex Synopsis elements. The customization layer in Example 5.4 removes `CmdSynopsis` and `FuncSynopsis`.

Example 5.4. Removing CmdSynopsis and FuncSynopsis

```
<!ENTITY % synop.class "Synopsis">
<!-- Instead of "Synopsis|CmdSynopsis|FuncSynopsis %local.synop.class;" -->

<!ENTITY % funcsynopsis.content.module "IGNORE">
<!ENTITY % cmdsynsynopsis.content.module "IGNORE">

<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

Completely removing all Synopsis elements would require a more extensive customization. You can't make any of the `%*.class;` parameter entities completely empty without changing all of the parameter entities that use them. See the section called “Removing an Entire Class”.”

Removing Sectioning Elements

Perhaps you want to restrict your authors to only three levels of sectioning. To do that, you must remove the `Sect4` and `Sect5` elements, as shown in Example 5.5.

Example 5.5. Removing Sect4 and Sect5 Elements

```
examples/remv.sect4.dtd
```

In order to completely remove an element that isn't in the information pool, it is usually necessary to redefine the elements that include it. In this case, because we're removing the `Sect4` element, we must redefine the `Sect3` element that uses it.

Removing Admonitions from Table Entries

All of the customization layers that we've examined so far have been fairly straightforward. This section describes a much more complex customization layer. Back in the section called “DocBook Modules”” we mentioned that several additional modules existed for “redeclaration.” The customization layer developed in this section cannot be written without them.

The goal is to remove admonitions (`Warning`, `Caution`, `Note`) from table entries.

Example 5.6 is a straightforward, and incorrect, attempt.

Example 5.6. Removing Admonitions (First Attempt: Incorrect)

```
<!-- THIS CUSTOMIZATION LAYER CONTAINS ERRORS -->
<!ENTITY % tabentry.mix
        "%list.class;
        |%linespecific.class;
        |%para.class;        |Graphic
        %local.tabentry.mix;">
```

```
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

Because the parameter entity `%tabentry.mix;` defines the mixture of elements allowed in table entries, you should remove admonitions.

If you attempt to parse this DTD, you'll find that the declaration of `%tabentry.mix;` contains errors. While you can redefine parameter entities, you cannot make reference to entities that have not been defined yet, so the use of `%list.class;`, `%linespecific.class;`, and so on, aren't allowed.

Your second attempt might look like Example 5.7.

Example 5.7. Removing Admonitions (Second Attempt: Incorrect)

```
<!-- THIS CUSTOMIZATION LAYER DOESN'T WORK -->
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
<!ENTITY % tabentry.mix
        "%list.class;
        |%linespecific.class;
        |%para.class;        |Graphic
        %local.tabentry.mix;">
```

Declaring `%tabentry.mix;` after the DTD has been loaded removes the errors.

This example contains no errors, but it also doesn't have any effect. Remember, only the first entity declaration counts, so the declaration of `%tabentry.mix;` in `dbpool.mod` is the one used, not your redeclaration.

The only way to fix this problem is to make use of one of the redeclaration placeholders in DocBook.

Redeclaration placeholders are spots in which you can insert definitions into the middle of the DTD. There are four redeclaration placeholders in DocBook:

`%rdbmods;`

Inserted in `docbook.dtd`, between `dbpool.mod` and `dbhier.mod`. This placeholder is controlled by the `%intermod.redecl.module;` marked section.

`%rdbpool;`

Inserted in the middle of `dbpool.mod`, between the `%*.class;` and `%*.mix;` entity declarations. This placeholder is controlled by the `%dbpool.redecl.module;` marked section.

`%rdbhier;`

Inserted in the middle of `dbhier.mod`, between the `%*.class;` and `%*.mix;` entity declarations. This placeholder is controlled by the `%dbhier.redecl.module;` marked section.

`%rdbhier2;`

Also inserted into `dbhier.mod`, after the `%*.mix;` entity declarations. This placeholder is controlled by the `%dbhier.redecl2.module;` marked section.

Use the redeclaration placeholder that it occurs nearest to, but before the entity that you want to redeclare. In our case, this is `%rdbpool;`, as seen in Example 5.8.

Example 5.8. Removing Admonitions (Third Attempt: Correct, if confusing)

```
<!ENTITY % dbpool.redecl.module "INCLUDE">
<!ENTITY % rdbpool
'<!ENTITY % local.tabentry.mix "">
<!ENTITY % tabentry.mix
        "&#37;list.class;
        |&#37;linespecific.class;
        |&#37;para.class;        |Graphic
        &#37;local.tabentry.mix;">'>

<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

Example 5.8 uses numeric character entity references to escape the `%` signs in the entity declarations and nests an entity declaration in another parameter entity. All of this is perfectly legal, but a bit confusing. A clearer solution, and the only practical solution if you're doing anything more than a single redeclaration, is to place the new declarations in another file and include them in your customization layer by reference, like this:

Example 5.9. Removing Admonitions (Fourth Attempt: Correct)

In your customization layer:

```
<!ENTITY % dbpool.redecl.module "INCLUDE">
<!ENTITY % rdbpool SYSTEM "rdbpool.mod">

<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

In `rdbpool.mod`:

```
<!ENTITY % local.tabentry.mix "">
<!ENTITY % tabentry.mix
        "%list.class;
        |%linespecific.class;
        |%para.class;        |Graphic
        %local.tabentry.mix;">
```

Removing an Entire Class

Perhaps the modification that you want to make is to completely remove an entire class of elements. (If you have no need for synopsis elements of any sort, why not remove them?) In order to remove an entire class of elements, you must not only redefine the class as empty, but you must also redefine all of the parameter entities that use that class. The customization layer below completely removes the `% synop.class;` from DocBook. It requires a customization layer, shown in Example 5.10, that includes both a redeclaration module in `dbpool.mod` and a redeclaration module in `dbhier.mod`.

Example 5.10. Removing synop.class

In the customization layer:

```
<!ENTITY % synop.class "">

<!ENTITY % dbpool.redecl.module "INCLUDE">
<!ENTITY % rdbpool SYSTEM "remv.synop.class.rdbpool.mod">

<!ENTITY % dbhier.redecl.module "INCLUDE">
<!ENTITY % rdbhier SYSTEM "remv.synop.class.rdbhier.mod">

<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

In `remv.synop.class.rdbpool.mod`:

In `remv.synop.class.rdbhier.mod`:

Removing Attributes

Just as there may be more elements than you need, there may be more attributes.

Removing an Attribute

Suppose you want to remove the `RenderAs` attribute from the `Sect1` element. `RenderAs` allows the author to “cheat” in the presentation of hierarchy by specifying that the stylesheet should render a `Sect1` as something else: a `Sect3`, perhaps. Example 5.11 details the removal of `RenderAs`.

Example 5.11. Removing RenderAs from Sect1

```
<!ENTITY % sect1.module "IGNORE">                    ❶

                                                     ❷
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;

<!ENTITY % local.sect1.attrib "">                    ❸

<!ENTITY % sect1.role.attrib "%role.attrib;">        ❹

<!ELEMENT Sect1 - O (Sect1Info?, (%sect.title.content;), (%nav.class;)*,   ❺
        (((%divcomponent.mix;)+,
        ((%refentry.class;)* | Sect2* | SimpleSect*))
        | (%refentry.class;)+ | Sect2+ | SimpleSect+), (%nav.class;)*)
        +(%ubiq.mix;)>

<!ATTLIST Sect1                                        ❻
        %label.attrib;
        %status.attrib;
        %common.attrib;
        %sect1.role.attrib;
        %local.sect1.attrib;
>
```

❶ Turn off the `Sect1` module so that the element and attribute declarations in the DTD will be ignored.

❷ Include the DocBook DTD.

❸ By keeping the local attribute declaration, we leave open the possibility of a simple customization layer on top of our customization layer.

❹ Similarly, we keep the parameterized definition of the `Role` attribute.

❺ We're changing the attribute list, not the element, so we've simply copied the `Sect1` element declaration from the DocBook DTD.

❻ Finally, we declare the attribute list, leaving out the `RenderAs`.

Subsetting the Common Attributes

DocBook defines eleven common attributes; these attributes appear on *every* element. Depending on how you're processing your documents, removing some of them can both simplify the authoring task and improve processing speed.

Some obvious candidates are:

Effectivity attributes (`Arch` , `OS`,...)

If you're not using all of the effectivity attributes in your documents, you can get rid of up to seven attributes in one fell swoop.

`Lang`

If you're not producing multilingual documents, you can remove `Lang`.

`Remap`

The `Remap` attribute is designed to hold the name of a semantically equivalent construct from a previous markup scheme (for example, a Microsoft Word style template name, if you're converting from Word). If you're authoring from scratch, or not preserving previous constructs with `Remap`, you can get rid of it.

`XrefLabel`

If your processing system isn't using `XrefLabel`, it's a candidate as well.

The customization layer in Example 5.12 reduces the common attributes to just `ID` and `Lang`.

Example 5.12. Removing Common Attributes

```
<!ENTITY % common.attrib
"ID   ID    #IMPLIED
 Lang CDATA #IMPLIED"
>
<!ENTITY % idreq.common.attrib
"ID   ID    #REQUIRED
 Lang CDATA #IMPLIED"
>
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```

By definition, whatever attributes you define in the `%common.attrib;` and `%idreq.common.attrib;` parameter entities are the common attributes. In `dbpool.mod` , these parameter entities are defined in terms of other parameter entities, but there's no way to preserve that structure in your customization layer.

Adding Elements: Adding a Sect6

Adding a structural (as opposed to information pool) element generally requires adding its name to a class and then providing the appropriate definitions. Example 5.13 extends DocBook by adding a `Sect6` element.

Example 5.13. Adding a Sect6 Element

```
<!ENTITY % sect5.module "IGNORE">
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
<!-- Add Sect6 to content model of Sect5 -->
<!ENTITY % sect5.role.attrib "%role.attrib;">
<!ELEMENT Sect5 - O (Sect5Info?, (%sect.title.content;), (%nav.class;)*,
        (((%divcomponent.mix;)+,
                ((%refentry.class;)* | Sect6* | SimpleSect*))
        | (%refentry.class;)+ | Sect6+ | SimpleSect+), (%nav.class;)*)>
<!ATTLIST Sect5
        %label.attrib;
        %status.attrib;
        %common.attrib;
        %sect5.role.attrib;
>
<!ENTITY % sect6.role.attrib "%role.attrib;">
<!ELEMENT Sect6 - O (Sect6Info?, (%sect.title.content;), (%nav.class;)*,
        (((%divcomponent.mix;)+, ((%refentry.class;)* | SimpleSect*))
        | (%refentry.class;)+ | SimpleSect+), (%nav.class;)*)>
<!ATTLIST Sect6
        %label.attrib;
        %status.attrib;
        %common.attrib;
        %sect6.role.attrib;
>
```

Here we've redefined `Sect5` to include `Sect6` and provided a declaration for `Sect6`. Note that we didn't bother to provide `RenderAs` attributes in our redefinitions. To properly support `Sect6`, you might want to redefine all of the sectioning elements so that `Sect6` is a legal attribute value for `RenderAs`.

Other Modifications: Classifying a Role

The `Role` attribute, found on almost all of the elements in DocBook, is a `CDATA` attribute that can be used to subclass an element. In some applications, it may be useful to modify the definition of `Role` so that authors must choose one of a specific set of possible values.

In Example 5.14, `Role` on the `Procedure` element is constrained to the values `Required` or `Optional`.

Example 5.14. Changing Role on Procedure

```
<!ENTITY % procedure.role.attrib "Role (Required|Optional) #IMPLIED">
<!-- load DocBook -->
<!ENTITY % DocBookDTD PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
%DocBookDTD;
```