The xfpt plain text to XML processor

Philip Hazel

The xfpt plain text to XML processor

Author: Philip Hazel

Revision 0.09   18 May 2012

Contents

1. Introduction

xfpt is a program that reads a marked-up ASCII source file, and converts it into XML. It was written with DocBook XML in mind, but can also be used for other forms of XML. Unlike *AsciiDoc* (**http://www.methods.co.nz/asciidoc/**), *xfpt* does not try to produce XML from a document that is also usable as a freestanding ASCII document. The input for *xfpt* is very definitely “marked up”. This makes it less ambiguous for large and/or complicated documents. *xfpt* is also much faster than *AsciiDoc* because it is written in C and does not rely on pattern matching.

xfpt is aimed at users who understand the XML that they are generating. It makes it easy to include literal XML, either in blocks, or within paragraphs. *xfpt* restricts itself to two special characters that trigger all its processing.

xfpt treats any input line that starts with a dot as a *directive* line. Directives control the way the input is processed. A small number of directives are implemented in the program itself. A macro facility makes it possible to combine these in various ways to define directives for higher-level concepts such as chapters and sections. A standard macro library that generates a simple subset of DocBook XML is provided. The only XML element that the program itself generates is `<para>`; all the others must be included as literal XML, either directly in the input text, or, more commonly, as part of the text that is generated by a macro call.

The ampersand character is special within non-literal text that is processed by *xfpt*. An ampersand introduces a *flag sequence* that modifies the output. Ampersand was chosen because it is also special in XML. As well as recognizing flag sequences that begin with an ampersand, *xfpt* converts grave accents and apostrophes that appear in non-literal text into typographic opening and closing quotes, as follows:

`` ` ``	becomes	‘
`'`	becomes	’

Within normal input text, ampersand, grave accent, and apostrophe are the only characters that cause *xfpt* to change the input text, but this applies only to non-literal text. In literal text, there are no markup characters, and only a dot at the start of a line is recognized as special. Within the body of a macro, there is one more special character: the dollar character is used to introduce an argument substitution.

Notwithstanding the previous paragraph, *xfpt* knows that it is generating XML, and in all cases when a literal ampersand or angle bracket is required in the output, the appropriate XML entity reference (`&amp;`, `&lt;`, or `&gt;`, respectively) is generated.

1.1 The xfpt command line

The format of the *xfpt* command line is:

`xfpt [`*options*`] [`*input source*`]`

If no input is specified, the standard input is read. There are four options:

-help
This option causes *xfpt* to output its “usage” message, and exit.

-o <*output destination*>
This option overrides the default destination. If the standard input is being read, the default destination is the standard output. Otherwise, the default destination is the name of the input file with the extension *.xml*, replacing its existing extension if there is one. A single hyphen character can be given as an output destination to refer to the standard output.

-S <*directory path*>
This option overrides the path to *xfpt*’s library directory that is built into the program. This makes it possible to use or test alternate libraries.

-v
This option causes *xfpt* to output its version number and exit.

1.2 A short xfpt example

Here is a very short example of a complete *xfpt* input file that uses some of the standard macros and flags:

```
.include stdflags
.include stdmacs
.docbook
.book

.chapter "The first chapter"
This is the text of the first chapter. Here is an &'italic'&
word, and here is a &*bold*& one.

.section "This is a section heading"
We can use the &*ilist*& macro to generate an itemized list:
.ilist
The first item in the list.
.next
The last item in the list.
.endlist

There are also standard macros for ordered lists, literal
layout blocks, code blocks, URL references, index entries,
tables, footnotes, figures, etc.
```

1.3 Literal and non-literal processing

xfpt processes non-directive input lines in one of four ways (known as “modes”):

- In the default mode, text is processed paragraph by paragraph.[1] The end of a paragraph is indicated by the end of the input, a blank line, or by an occurrence of the **.literal** directive. Other directives (for example, **.include**) do not of themselves terminate a paragraph. Most of the standard macros (such as **.chapter** and **.section**) force a paragraph end by starting their contents with a **.literal** directive.

  Because *xfpt* reads a whole paragraph before processing it, error messages contain the phrase “detected near line *nnn*”, where the line number is typically that of the last line of the paragraph.

- In the “literal layout” mode, text is processed line by line, but is otherwise handled as in the default mode. The only real difference this makes to the markup from the user’s point of view is that both parts of a set of paired flags must be on the same line. In this mode, error messages are more likely to contain the exact line number where the fault lies. Literal layout mode is used by the standard **.display** macro to generate `<literallayout>` elements.
- In the “literal text” mode, text is also processed line by line, but no flags are recognized. The only modification *xfpt* makes to the text is to turn ampersand and angle bracket characters into XML entity references. This mode is used by the standard **.code** macro to generate `<literallayout>` elements that include `class=monospaced`.
- In the “literal XML” mode, text lines are copied to the output without modification. This is the easiest way to include a chunk of literal XML in the output. An example might be the `<bookinfo>` element, which occurs only once in a document. It is not worth setting up a macro for a one-off item like this.

The **.literal** directive switches between the modes. It is not normally used directly, but instead is incorported into appropriate macro definitions. The **.inliteral** directive can be used to test the current mode.

---

[1] There is, however, a special case when a paragraph contains one or more footnotes. In that situation, each part of the outer paragraph is processed independently.

Directive lines are recognized and acted upon in all four modes. However, an unrecognized line that starts with a dot in the literal text or literal XML mode is treated as data. In the other modes, such a line provokes an error.

If you need to have a data line that begins with a dot in literal layout mode, you can either specify it by character number, or precede it with some non-acting markup. These two examples are both valid:

```
&#x2e;start with a dot
&''&.start with a dot
```

The second example assumes the standard flags are defined: it precedes the dot with an empty italic string. However, this is untidy because the empty string will be carried over into the XML.

In literal text or literal XML mode, it is not possible to have a data line that starts with a dot followed by the name of a directive or macro. You have to use literal layout mode if you require such output. Another solution, which is used in the source for this document (where many examples show directive lines), is to indent every displayed line by one space, and thereby avoid the problem altogether.

1.4 Format of directive lines

If an input line starts with a dot followed by a space, it is ignored by *xfpt*. This provides a facility for including comments in the input. Otherwise, the dot must be followed by a directive or macro name, and possibly one or more arguments. Arguments that are strings are delimited by white space unless they are enclosed in single or double quotes. The delimiting quote character can be included within a quoted string by doubling it. Here are some examples:

```
.literal layout
.set version 0.00
.row "Jack's house" 'Jill''s house'
```

An unrecognized directive line normally causes an error; however, in the literal text and literal XML modes, an unrecognized line that starts with a dot is treated as a data line.

1.5 Calling macros

Macros are defined by the **.macro** directive, which is described in section 3.11. There are two ways of calling a macro. It can be called in the same way as a directive, or it can be called from within text that is being processed. The second case is called an “inline macro call”.

When a macro is called as a directive, its name is given after a dot at the start of a line, and the name may be followed by any number of optional arguments, in the same way as a built-in directive (see the previous section). For example:

```
.chapter "Chapter title" chapter-reference
```

The contents of the macro, after argument substitution, are processed in exactly the same way as normal input lines. A macro that is called as a directive may contain nested macro calls.

When a macro is called from within a text string, its name is given after an ampersand, and is followed by an opening parenthesis. Arguments, delimited by commas, can then follow, up to a closing parenthesis. If an argument contains a comma or a closing parenthesis, it must be quoted. White space after a separating comma is ignored. The most common example of this type of macro call is the standard macro for generating a URL reference:

```
Refer to a URL via &url(http://x.example,this text).
```

There are differences in the behaviour of macros, depending on which way they are called. A macro that is called inline may not contain references to other macros; it must contain only text lines and calls to built-in directives. Also, newlines that terminate text lines within the macro are not included in the output.

A macro that can be called inline can always be called as a directive, but the opposite is not always true. Macros are usually designed to be called either one way or the other. However, the **.new** and **.index** macros in the standard library are examples of macros that are designed be called either way.

2. Flag sequences

Only one flag sequence is built-into the code itself. If an input line ends with three ampersands (ignoring trailing white space), the ampersands are removed, and the next input line, with any leading white space removed, is joined to the original line. This happens before any other processing, and may involve any number of lines. Thus:

```
The quick &&&
    brown &&&
      fox.
```

produces exactly the same output as:

```
The quick brown fox.
```

2.1 Flag sequences for XML entities and xfpt variables

If an ampersand is followed by a # character, a number, and a semicolon, it is understood as a numerical reference to an XML entity, and is passed through unmodified. The number can be decimal, or hexadecimal preceded by `x`. For example:

```
This is an Ohm sign: &#x2126;.
This is a degree sign: &#176;.
```

If an ampersand is followed by a letter, a sequence of letters, digits, and dots is read. If this is terminated by a semicolon, the characters between the ampersand and the semicolon are interpreted as an entity name. This can be:

- The name of an inbuilt *xfpt* variable. At present, there is only one of these, called `xfpt.rev`. Its use is described with the **.revision** directive below.
- The name of a user variable that has been set by the **.set** directive, also described below.
- The name of an XML entity. This is assumed if the name is not recognized as one of the previous types. In this case, the input text is passed to the output without modification. For example:

  ```
  This is an Ohm sign: &Ohm;.
  ```

2.2 Flag sequences for calling macros

If an ampersand is followed by a sequence of alphanumeric characters starting with a letter, terminated by an opening parenthesis, the characters between the ampersand and the parenthesis are interpreted as the name of a macro. See section 1.5 for more details.

2.3 Other flag sequences

Any other flag sequences that are needed must be defined by means of the **.flag** directive. These are of two types, standalone and paired. Both cases define replacement text. This is always literal; it is not itself scanned for flag occurrences.

Lines are scanned from left to right when flags are being interpreted. If there is any ambiguity when a text string is being scanned, the longest flag sequence wins. Thus, it is possible (as in the standard flag sequences) to define both &< and &<< as flags, provided that you never want to follow the first of them with a < character.

You can define flags that start with &#, but these must be used with care, lest they be misinterpreted as numerical references to XML entities.

A standalone flag consists of an ampersand followed by any number of non-alphanumeric characters. When it is encountered, it is replaced by its replacement text. For example, in the standard flag definitions, && is defined as a standalone flag with with the replacement text `&amp;`.

A paired flag is defined as two sequences. The first takes the same form as a standalone flag. The second also consists of non-alphanumeric characters, but need not start with an ampersand. It is often defined as the reverse of the first sequence. For example, in the standard definitions, `&'` and `'&` are defined as a flag pair for enclosing text in an `<emphasis>` element.

When the first sequence of a paired flag is encountered, its partner is expected to be found within the same text unit. In the default mode, the units are a paragraphs, or part-paragraphs if footnotes intervene. In literal layout mode, the text is processed line by line. Each member of the pair is replaced by its replacement text.

Multiple occurrences of paired flags must be correctly nested. Note that, though *xfpt* diagnoses an error for badly nested flag pairs, it does not prevent you from generating invalid XML. For example, DocBook does not allow `<emphasis>` within `<literal>`, though it does allow `<literal>` within `<emphasis>`.

2.4 Unrecognized flag sequences

If an ampersand is not followed by a character sequence in one of the forms described in the preceding sections, an error occurs.

2.5 Standard flag sequences

These are the standalone flag sequences that are defined in the *stdflags* file in the *xfpt* library:

`&&`	becomes	`&amp;` (ampersand)
`&--`	becomes	`&ndash;` (en-dash)
`&~`	becomes	`&nbsp;` (‘hard’ space)

These are the flag pairs that are defined in the *stdflags* file in the *xfpt* library:

`&"..."&`	becomes	`<quote>...</quote>`
`&'...'&`	becomes	`<emphasis>...</emphasis>`
`&*...*&`	becomes	`<emphasis role="bold">...</emphasis>`
``&`...`&``	becomes	`<literal>...</literal>`
`&_..._&`	becomes	`<filename>...</filename>`
`&(...)&`	becomes	`<command>...</command>`
`&[...]&`	becomes	`<function>...</function>`
`&%...%&`	becomes	`<option>...</option>`
`&$...$&`	becomes	`<varname>...</varname>`
`&<...>&`	becomes	`<...>`
`&<<...>>&`	becomes	`<xref linkend="..."/>`

For example, if you want to include a literal XML element in your output, you can do it like this: `&<element>&`. If you want to include a longer sequence of literal XML, changing to the literal XML mode may be more convenient.

3. Built-in directive processing

The directives that are built into the code of *xfpt* are now described in alphabetical order. You can see more examples of their use in the descriptions of the standard macros in chapter 4.

3.1 The .arg directive

This directive may appear only within the body of a macro. It must be followed by a single number, optionally preceded by a minus sign. If the number is positive (no minus sign), subsequent lines, up to a **.endarg** directive, are skipped unless the macro has been called with at least that number of arguments and the given argument is not an empty string. If the number is negative (minus sign present), subsequent lines are skipped if the macro has been called with fewer than that number of arguments, or with an empty string for the given argument. For example:

```
.macro example
.arg 2
Use these lines if there are at least 2 arguments
and the second one is not empty. Normally there would
be a reference to the 2nd argument.
.endarg
.arg -2
Use this line unless there are at least 2 arguments
and the second one is not empty.
.endarg
.endmacro
```

Note that if a macro is defined with default values for its arguments, these are not counted by the **.arg** directive, which looks only at the actual arguments in a particular macro call.

The **.arg** directive may be nested.

3.2 The .eacharg directive

This directive may appear only within the body of a macro. It may optionally be followed by a single number; if omitted the value is taken to be 1. Subsequent lines, up to a **.endeach** directive, are processed multiple times, once for each remaining argument. Unlike **.arg**, an argument that is an empty string is not treated specially. However, like **.arg**, only the actual arguments of a macro call are considered. Default argument values do not count.

The number given with **.eacharg** defines which argument to start with. If the macro is called with fewer arguments, the lines up to **.endeach** are skipped, and are not processed at all. When these lines are being processed, the remaining macro arguments can be referenced relative to the current argument. `$+1` refers to the current argument, `$+2` to the next argument, and so on.

The **.endeach** directive may also be followed by a number, again defaulting to 1. When **.endeach** is reached, the current argument number is incremented by that number. If there are still unused arguments available, the lines between **.eacharg** and **.endeach** are processed again.

This example is taken from the coding for the standard **.row** macro, which generates an `<entry>` element for each of its arguments:

```
.eacharg
&<entry>&$+1&</entry>&
.endeach
```

This example is taken from the coding for the standard **.itable** macro, which processes arguments in pairs to define the table's columns, starting from the fifth argument:

```
.eacharg 5
&<colspec colwidth="$+1" align="$+2"/>&
.endeach 2
```

The **.eacharg** directive may in principle be nested, though this does not seem useful in practice.

3.3 The .echo directive

This directive takes a single string argument. It writes it to the standard error stream. Within a macro, argument substitution takes place, but no other processing is done on the string. This directive can be useful for debugging macros or writing comments to the user.

3.4 The .endarg directive

See the description of **.arg** above.

3.5 The .endeach directive

See the description of **.eacharg** above.

3.6 The .endinliteral directive

See the description of **.inliteral** below.

3.7 The .flag directive

This directive is used to define flag sequences. The directive must be followed either by a standalone flag sequence and one string in quotes, or by a flag pair and two strings in quotes. White space separates these items. For example:

```
.flag && "&amp;"
.flag &" "&  "<quote>"  "</quote>"
```

There are more examples in the definitions of the standard flags. If you redefine an existing flag, the new definition overrides the old. There is no way to revert to the previous definition.

3.8 The .include directive

This directive must be followed by a single string argument that is the path to a file. The contents of the file are read and incorporated into the input at this point. If the string does not contain any slashes, the path to the *xfpt* library is prepended. Otherwise, the path is used unaltered. If **.include** is used inside a macro, it is evaluated each time the macro is called, and thus can be used to include a different file on each occasion.

3.9 The .inliteral directive

This directive may appear only within the body of a macro. It must be followed by one of the words “layout”, “text”, “off”, or “xml”. If the current literal mode does not correspond to the word, subsequent lines, up to a **.endinliteral** directive, are skipped. The **.inliteral** directive may be nested.

3.10 The .literal directive

This must be followed by one of the words “layout”, “text”, “off”, or “xml”. It forces an end to a previous paragraph, if there is one, and then switches between processing modes. The default mode is the “off” mode, in which text is processed paragraph by paragraph, and flags are recognized. Section 1.3 describes how input lines are processed in the four modes.

3.11 The .macro directive

This directive is used to define macros. It must be followed by a macro name, and then, optionally, by any number of arguments. The macro name can be any sequence of non-whitespace characters. The arguments in the definition provide default values. The following lines, up to **.endmacro**, form the body of the macro. They are not processed in any way when the macro is defined; they are processed only when the macro is called (see section 1.5).

Within the body of a macro, argument substitutions can be specified by means of a dollar character and an argument number, for example, `$3` for the third argument. See also **.eacharg** above for the use of `$+` to refer to relative arguments when looping through them. A reference to an argument that is not supplied, and is not given a default, results in an empty substitution.

There is also a facility for a conditional substitution. A reference to an argument of the form:

`$=`*<digits><delimiter><text><delimiter>*

inserts the text if the argument is defined and is not an empty string, and nothing otherwise. The text is itself scanned for flags and argument substitutions. The delimiter must be a single character that does not appear in the text. For example:

```
&<chapter$=2+ id="$2"+>&
```

If this appears in a macro that is called with only one argument, the result is:

```
<chapter>
```

but if the second argument is, say `abcd`, the result is:

```
<chapter id="abcd">
```

This conditional feature can be used with both absolute and relative argument references.

If a dollar character is required as data within the body of a macro, it must be doubled. For example:

```
.macro price
The price is $$1.
.endmacro
```

If you redefine an existing macro, the new definition overrides the old. There is no way to revert to the previous definition. If you define a macro whose name is the same as the name of a built-in directive you will not be able to call it, because *xfpt* looks for built-in directives before it looks for macros.

It is possible to define a macro within a macro, though clearly care must be taken with argument references to ensure that substitutions happen at the right level.

3.12 The .nest directive

This directive must be followed by one of the words “begin” or “end”. It is used to delimit a nested sequence of independent text items that occurs inside another, such as the contents of a footnote inside a paragraph. This directive is usually used inside a macro. For example, a **footnote** macro could be defined like this:

```
.macro footnote
&<footnote>&
.nest begin
.endmacro
```

At the start of a nested sequence, the current mode and paragraph state are remembered and *xfpt* then reverts to the default mode and “not in a paragraph”. At the end of a nested sequence, if a paragraph has been started, it is terminated, and then *xfpt* reverts to the previous state.

3.13 The .nonl directive

This directive must be followed by a single string argument. It is processed as an input line without a newline at the end. This facility is useful in macros when constructing a single data line from several text fragments. See for example the **.new** macro in the standard macros.

3.14 The .pop directive

xfpt keeps a stack of text strings that are manipulated by the **.push** and **.pop** directives. When the end of the input is reached, any strings that remain on the stack are popped off, processed for flags, and written to the output. In some cases (see the **.push** directive below) a warning message is given.

Each string on the stack may, optionally, be associated with an upper case letter. If **.pop** is followed by an upper case letter, it searches down the stack for a string with the same letter. If it cannot find one, it does nothing. Otherwise, it pops off, processes, and writes out all the strings down to and including the one that matches.

If **.pop** is given without a following letter, it pops one string off the stack and writes it out. If there is nothing on the stack, an error occurs.

3.15 The .push directive

This directive pushes a string onto the stack. If the rest of the command line starts with an upper case letter followed by white space or the end of the line, that letter is associated with the string that is pushed, which consists either of a quoted string, or the rest of the line. After a quoted string, the word ‘check’ may appear. In this case, if the string has not been popped off the stack by the end of processing, a warning message is output. This facility is used by the standard macros to give warnings for unclosed items such as **.ilist**.

For example, the **.chapter** macro contains this line:

```
.push C &</chapter>&
```

Earlier in the macro there is the line:

```
.pop C
```

This arrangement ensures that any previous chapter is terminated before starting a new one, and also when the end of the input is reached. The **.ilist** macro contains this line:

```
.push L "&</itemizedlist>&" check
```

Item lists are terminatated by **.endlist**, which contains:

```
.pop L
```

However, if **.endlist** is accidentally omitted (or **.ilist** is accidentally included), the appearance of ‘check’ means that a warning is issued to alert the user to a possible problem.

3.16 The .revision directive

This directive is provided to make it easy to set the `revisionflag` attribute on XML elements in a given portion of the document. The DocBook specification states that the `revisionflag` attribute is common to all elements.

The **.revision** directive must be followed by one of the words “changed”, “added”, “deleted”, or “off”. For any value other than “off”, it causes the internal variable *xfpt.rev* to be set to `revisionflag=` followed by the given argument. If the argument is “off”, the internal variable is emptied.

The contents of *xfpt.rev* are included in every `<para>` element that *xfpt* generates. In addition, a number of the standard macros contain references to *xfpt.rev* in appropriate places. Thus, setting:

```
.revision changed
```

should cause all subsequent text to be marked up with `revisionflag` attributes, until

```
.revision off
```

is encountered. Unfortunately, at the time of writing, not all DocBook processing software pays attention to the `revisionflag` attribute. Furthermore, some software grumbles that it is “unexpected” on some elements, though it does still seem to process it correctly.

For handling the most common case (setting and unsetting “changed”), the standard macros **.new** and **.wen** are provided (see section 4.13).

3.17 The .set directive

This directive must be followed by a name and a text string. It defines a user variable and gives it a name. A reference to the name in the style of an XML entity causes the string to be substituted, without further processing. For example:

```
.set version 4.99
```

This could be referenced as `&version;`. If a variable is given the name of an XML entity, you will not be able to refer to the XML entity, because local variables take precedence. There is no way to delete a local variable after it has been defined.

4. The standard macros for DocBook

A set of simple macros for commonly needed DocBook features is provided in *xfpt*'s library. This may be extended as experience with *xfpt* accumulates. The standard macros assume that the standard flags are defined, so a document that is going to use these features should start with:

```
.include stdflags
.include stdmacs
```

All the standard macros except **new**, **index**, and **url** are intended to be called as directive lines. Their names are therefore shown with a leading dot in the discussion below.

4.1 Overall setup

There are two macros that should be used only once, at the start of the document. The **.docbook** macro has no arguments. It inserts into the output file the standard header material for a DocBook XML file, which is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
"http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
```

The **.book** macro has no arguments. It generates `<book>` and pushes `</book>` onto the stack so that it will be output at the end.

4.2 Processing instructions

XML processing instructions such as `<?sdop toc_sections="no"?>` can, of course, be written written literally between `.literal xml` and `.literal off`. If there are a lot of them, this is perhaps the most convenient approach. A macro called **.pi** is provided as an easy way of setting up a short processing instruction. Its first argument is the name of the processor for which the instruction is intended, and its second argument is the contents of the instruction, for example:

```
.pi sdop 'toc_sections="yes,yes,no"'
```

This generates `<?sdop toc_sections="yes,yes,no"?>`.

4.3 Chapters, sections, and subsections

Chapters, sections, and subsections are supported by three macros that all operate in the same way. They are **.chapter**, **.section**, and **.subsection**. They take either one, two, or three arguments. The first argument is the title. If a second argument is present, and is not an empty string, it is set as an ID, and can be used in cross-references. For example:

```
.chapter "Introduction"
```

sets no ID, but

```
.section "A section title" "SECTdemo"
```

can be referenced from elsewhere in the document by a phrase such as:

```
see section &<<SECTdemo>>&
```

When the title of a chapter of section is being used as a running head or foot (for example), it may be too long to fit comfortably into the available space. DocBook provides the facility for a title abbreviation to be specified to deal with this problem. If a third argument is given to one of these macros, it causes a `<titleabbrev>` element to be generated. In this case, a second argument must also be provided, but if you do not need an ID, the second argument can be an empty string. For example:

```
.chapter "This chapter has quite a long title" "" "Long title"
```

Where and when the abbreviation is used in place of the full title is controlled by the stylesheet when the XML is processed.

4.4 Prefaces, appendixes, and colophons

The macros **.preface**, **.appendix**, and **.colophon** operate in the same way as **.chapter**, except that the first and the last have the default title strings “Preface” and “Colophon”.

4.5 Terminating chapters, etc.

The macros for chapters, sections, appendixes, etc. use the stack to ensure that each one is terminated at the correct point, without the need for an explicit terminator. For example, starting a new section automatically terminates an open subsection and a previous section.

Occasionally, however, there is a need to force an explicit termination. The **.endchapter**, **.endsection**, **.endsubsection**, **.endpreface**, **.endappendix**, and **.endcolophon** macros provide this facility. For example, if you want to include an XML processing instruction after a preface, but before the start of the following chapter, you must terminate the preface with **.endpreface**. Otherwise a processing instruction that precedes the next **.chapter** will end up inside the `<preface>` element. You should not include any actual text items at these points.

4.6 URL references

The **url** macro generates URL references, and is intended to be called inline within the text that is being processed. It generates a `<ulink>` element, and has either one or two arguments. The first argument is the URL, and the second is the text that describes it. For example:

```
More details are &url(http://x.example, here).
```

This generates the following XML:

```
More details are <ulink url="http://x.example">here</ulink>.
```

If the second argument is absent, the contents of the first argument are used instead. If **url** is called as a directive, there will be a newline in the output after `</ulink>`, which in most cases (such as the example above), you do not want.

4.7 Itemized lists

The **.ilist** macro marks the start of an itemized list, the items of which are normally rendered with bullets or similar markings. The macro can optionally be called with one argument, for which there is no default. If the argument is present, it is used to add a `mark=` attribute to the `<itemizedlist>` element that is generated. The mark names that can be used depend on the software that processes the resulting XML. For HTML output, “square” and “opencircle” work in some browsers.

The text for the first item follows the macro call. The start of the next item is indicated by the **.next** macro, and the end of the list by **.endlist**. For example:

```
.ilist
This is the first item.
.next
This is the next item.
.endlist
```

There may be more than one paragraph in an item.

4.8 Ordered lists

The **.olist** macro marks the start of an ordered list, the items of which are numbered. If no argument is given, arabic numerals are used. One of the following words can be given as the macro’s argument to specify the numeration:

`arabic`	arabic numerals
`loweralpha`	lower case letters
`lowerroman`	lower case roman numerals

`upperalpha`	upper case letters
`upperroman`	upper case roman numerals

The text for the first item follows the macro call. The start of the next item is indicated by the **.next** macro, and the end of the list by **.endlist**. For example:

```
.olist lowerroman
This is the first item.
.next
This is the next item.
.endlist
```

There may be more than one paragraph in an item.

4.9 Variable lists

A variable list is one in which each entry is composed of a set of one or more terms and an associated description. Typically, the terms are printed in a style that makes them stand out, and the description is indented underneath. The start of a variable list is indicated by the **.vlist** macro, which has one optional argument. If present, it defines a title for the list.

Each entry is defined by a **.vitem** macro, whose arguments are the terms. This is followed by the body of the entry. The list is terminated by the **.endlist** macro. For example:

```
.vlist "Font filename extensions"
.vitem "TTF"
TrueType fonts.
.vitem "PFA" "PFB"
PostScript fonts.
.endlist
```

As for the other lists, there may be more than one paragraph in an item.

4.10 Nested lists

Lists may be nested as required. Some DocBook processors automatically choose different bullets for nested itemized lists, but others do not. The **.endlist** macro has no useful arguments. Any text that follows it is treated as a comment. This can provide an annotation facility that may make the input easier to understand when lists are nested.

4.11 Displayed text

In displayed text each non-directive input line generates one output line. The `<literallayout>` DocBook element is used to achieve this. Two kinds of displayed text are supported by the standard macros. They differ in their handling of the text itself.

The macro **.display** is followed by lines that are processed in the same way as normal paragraphs: flags are interpreted, and so there may be font changes and so on. The lines are processed in literal layout mode. For example:

```
.display
&`-o`&   set output destination
&`-S`&   set library path
.endd
```

The output is as follows:

`-o` set output destination
`-S` set library path

The macro **.code** is followed lines that are not processed in any way, except to turn ampersands and angle brackets into XML entities. The lines are processed in literal text mode. In addition, `class="monospaced"` is added to the `<literallayout>` element, so that the lines are displayed in a monospaced font. For example:

```
.code
z = sqrt(x*x + y*y);
.endd
```

As the examples illustrate, both kinds of display are terminated by the **.endd** macro.

4.12 Block quotes

The macro pair **.blockquote** and **.endblockquote** are used to wrap the lines between them in a `<blockquote>` element.

4.13 Revision markings

Two macros are provided to simplify setting and unsetting the "changed" revision marking (see section 3.16). When the revised text is substantial (for example, a complete paragraph, table, display, or section), it can be placed between **.new** and **.wen**, as in this example:

```
This paragraph is not flagged as changed.
.new
This is a changed paragraph that contains a display:
.display
whatever
.endd
This is the next paragraph.
.wen
Here is the next, unmarked, paragraph.
```

When called like this, without an argument, in ordinary text, **.new** terminates the current paragraph, and **.wen** always does so. Therefore, even though there are no blank lines before **.new** or **.wen** above, the revised text will end up in a paragraph of its own. (You can, of course, put in blank lines if you wish.)

If want to indicate that just a few words inside a paragraph are revised, you can call the **new** macro with an argument. The macro can be called either as a directive or inline:

```
This is a paragraph that has
.new "a few marked words"
within it. Here are &new(some more) marked words.
```

The effect of this is to generate a `<phrase>` XML element with the `revisionflag` attribute set. The **.wen** macro is not used in this case.

You can use the **.new**/**.wen** macro pair to generate a `<phrase>` element inside a section of displayed text. For example:

```
.display
This line is not flagged as changed.
.new
This line is flagged as changed.
.wen
This line is not flagged as changed.
.endd
```

This usage works with both **.display** and **.code**. Within a **.display** section you can also call **.new** with an argument, either as a directive or inline. This does not work for **.code** because its lines are processed in literal text mode.

If you want to add revision indications to part of a table, you must use an inline call of **new** within an argument of the **.row** macro (see below). This is the only usage that works in this case.

4.14 Informal tables

The **.itable** macro starts an informal (untitled) table with some basic parameterization. If you are working on a large document that has many tables with the same parameters, the best approach is to define your own table macros, possibly calling the standard one with specific arguments.

The **.itable** macro has four basic arguments:

(1) The frame requirement for the table, which may be one of the words “all”, “bottom”, “none” (the default), “sides”, “top”, or “topbot”.

(2) The “colsep” value for the table. The default is “0”, meaning no vertical separator lines between columns. The value “1” requests vertical separator lines.

(3) The “rowsep” value for the table. The default is “0”, meaning no horizontal lines between rows. The value “1” requests horizontal separator lines.

(4) The number of columns.

These arguments must be followed by two arguments for each column. The first specifies the column width, and the second its alignmnent. A column width can be specified as an absolute dimension such as 36pt or 2in, or as a proportional measure, which has the form of a number followed by an asterisk. The two forms can be mixed – see the DocBook specification for details.

Straightforward column alignments can be specified as “center”, “left”, or “right”. DocBook also has some other possibilities, but sadly they do not seem to include “centre”.

Each row of the table is specified using a **.row** macro; the entries in the row are the macros’s arguments. The table is terminated by **.endtable**, which has no arguments. For example:

```
.itable all 1 1 2 1in left 2in center
.row "cell 11" "cell 12"
.row "cell 21" "cell 22"
.endtable
```

This specifies a framed table, with both column and row separator lines. There are two columns: the first is one inch wide and left aligned, and the second is two inches wide and centred. There are two rows. The resulting table looks like this:

cell 11	cell 12
cell 21	cell 22

The **.row** macro does not set the `revisionflag` attribute in the `<entry>` elements that it generates because this appears to be ignored by all current XML processors. However, you can use an inline call of the **new** macro within an entry to generate a `<phrase>` element with `revisionflag` set.

4.15 Formal tables

The **.table** macro starts a formal table, that is, a table that has a title, and which can be cross referenced. The first argument of this macro is the table’s title; the second is an identifier for cross-referencing. If you are not going to reference the table, an empty string must be supplied. From the third argument onwards, the arguments are identical to the **.itable** macro. For example:

```
.table "A title for the table" "" all 1 1 2 1in left 2in center
.row "cell 11" "cell 12"
.row "cell 21" "cell 22"
.endtable
```

4.16 Figures and images

A figure is enclosed between **.figure** and **.endfigure** macros. The first argument of **.figure** provides a title for the figure. The second is optional; if present, it is a tag for references to the figure.

A figure normally contains an image. The **.image** macro can be used in simple cases. It generates a `<mediaobject>` element containing an `<imageobject>`. The first argument is the name of the file containing the image. The remaining arguments are optional; an empty string must be supplied as a placeholder when one that is not required is followed by one that is set.

- The second argument specifies a scaling factor for the image, as a percentage. Thus, a value of 50 reduces the image to half size.
- The third argument specifies an alignment for the image. It must be one of `left` (default), `right` or `center` (or even `centre` if the DocBook processor you are using can handle it).
- The fourth and fifth arguments specify the depth and width, respectively. How these values are handled depends on the processing software.

Here is an example of the input for a figure, with all the image options defaulted:

```
.figure "My figure's title" "FIGfirst"
.image figure01.eps
.endfigure
```

Here is another example, where the figure is reduced to 80% and centred:

```
.figure "A reduced figure"
.image figure02.eps 80 center
.endfigure
```

4.17 Footnotes

Footnotes can be specified between **.footnote** and **.endnote** macros. Within a footnote there can be any kind of text item, including displays and tables. When a footnote occurs in the middle of a paragraph, paired flags must not straddle the footnote. This example is wrong:

```
The &'quick
.footnote
That's really fast.
.endf
brown'& fox.
```

The correct markup for this example is:

```
The &'quick'&
.footnote
That's really fast.
.endf
&'brown'& fox.
```

4.18 Indexes

The **.index** macro generates `<indexterm>` elements (index entries) in the output. It takes one or two arguments. The first is the text for the primary index term, and the second, if present, specifies a secondary index term. This macro can be called either from a directive line, or inline. However, it is mostly called as a directive, at the start of a relevant paragraph. For example:

```
.index goose "wild chase"
The chasing of wild geese...
```

You can generate “see” and “see also” index entries by using **.index-see** and **.index-seealso** instead of **.index**. The first argument of these macros is the text for the “see”. For example:

```
.index-see "chase" "wild goose"
```

This generates:

```
<indexterm>
<primary>wild goose</primary>
```

```
<see>chase</see>
</indexterm>
```

If you want to generate an index entry for a range of pages, you can use the **.index-from** and **.index-to** macros. The first argument of each of them is an ID that ties them together. The second and third arguments of **.index-from** are the primary and secondary index items. For example:

```
.index-from "ID5" "indexes" "handling ranges"
... <lines of text> ...
.index-to "ID5"
```

The **.makeindex** macro should be called at the end of the document, at the point where you want an index to be generated. It can have up to two arguments. The first is the title for the index, for which the default is “Index”. The second, if present, causes a `role=` attribute to be added to the `<index>` element that is generated. For this to be useful, you need to generate `<indexterm>` elements that have similar `role=` attributes. The standard **index** macro cannot do this. If you want to generate multiple indexes using this mechanism, it is best to define your own macros for each index type. For example:

```
.macro cindex
&<indexterm role="concept">&
&<primary>&$1&</primary>&
.arg 2
&<secondary>&$2&</secondary>&
.endarg
&</indexterm>&
.endmacro
```

This defines a **.cindex** macro for the “concept” index. At the end of the document you might have:

```
.makeindex "Concept index" "concept"
.makeindex
```

As long as the processing software can handle multiple indexes, this causes two indexes to be generated. The first is entitled “Concept index”, and contains only those index entries that were generated by the **.cindex** macro. The second contains all index entries.