The xmlformat XML Document Formatter

Paul DuBois <paul@kitebird.com>

Table of Contents

1. Introduction

xmlformat is a formatter (or "pretty-printer") for XML documents. It is useful when you want XML documents to have a standard format. This includes situations such as the following:

- XML documents that are maintained in a version control system, where people who use different XML editors work on the documents. XML editors typically impose their own style conventions on files. The application of different style conventions to successive document revisions can result in large version diffs where most of the bulk is related only to changes in format rather than content. This can be a problem if, for example, the version control system automatically sends the diffs to a committer's mailing list that people read. If documents are rewritten to a common format before they are committed, these diffs become smaller. They better reflect content changes and are easier for people to scan and understand.

- Similarly, if you send an XML document to someone who edits it and sends it back, it's easier to see what was changed by putting the before and after versions in a common format. This is a simple alternative to using a more sophisticated semantic XML diff utility.

Of course, these benefits can be obtained by using any XML pretty printer. So why does **xmlformat** exist? Because most XML formatters reformat documents using a set of built-in rules that determine the output style. Some allow you to select from one of several predefined output styles. That's fine if the style you want is the style produced by one of these tools. If not, you're stuck. That's where **xmlformat** comes in, because it's configurable. You can specify formatting options in a file and **xmlformat** will apply them to your documents. If you have different applications for which you want different styles, you can select the style by using the appropriate configuration file.

xmlformat has a default overall output style, but you can redefine the default style, and you can override the default on a per-element basis. For example, you can indicate whether the element should be treated as a block element, an inline element, or a verbatim element. For any block element, you can control several formatting properties:

- Spacing (line breaks) between nested sub-elements. You can also control spacing between the element's opening and closing tags and its content. (The general assumption is that if a block element has non-inline sub-elements, you'll want to space those sub-elements evenly within the enclosing block, though possibly with different spacing between the opening tag and the first child, or between the last child and the closing tag.)

- Indentation of nested sub-elements.

- Whitespace normalization and line-wrapping of text within the element.

xmlformat is free software. You can redistribute it or modify it under the terms specified in the `LICENSE` file.

For installation instructions, see the `INSTALL` file.

2. How to Use xmlformat

To format the file `mydoc.xml` using the default formatting options, use the following command. (% represents your shell prompt here; do not type it as part of the command.)

```
% xmlformat mydoc.xml
```

(**xmlformat** might be installed as **xmlformat.pl** or **xmlformat.rb**, depending on implementation language. In that case, you should invoke it under the appropriate name.)

The built-in formatting options cause each element to begin a new line, with sub-element indentation of one space, and no text normalization. Suppose `mydoc.xml` looks like this:

```
<table> <row> <cell> A </cell> <cell> B </cell> </row>
<row> <cell> C </cell> <cell> D </cell> </row> </table>
```

xmlformat will produce this result by default:

```
<table>
 <row>
  <cell> A </cell>
  <cell> B </cell>
 </row>
 <row>
  <cell> C </cell>
  <cell> D </cell>
 </row>
</table>
```

The default style is perhaps suitable for data-oriented XML documents that contain no mixed-content elements. For more control over output, specify a configuration file.

If the formatting options are stored in a file named `xf-opts.conf`, you can apply them to the document by specifying a `--config-file` option:

```
% xmlformat --config-file=xf-opts.conf mydoc.xml
```

If you do not specify a configuration file using a `--config-file` (or `--f`) option, **xmlformat** uses the following rules to determine what formatting options to use:

- If the environment variable `XMLFORMAT_CONF` is defined, **xmlformat** uses its value as the name of the configuration file.
- Otherwise, if a file named `xmlformat.conf` exists in the current directory, **xmlformat** uses it as the configuration file.
- Otherwise, **xmlformat** uses a set of built-in formatting options.

Configuration options and configuration file syntax are described in Section 4, " Using Configuration Files ".

To see the command-line options that **xmlformat** supports, invoke it with the `--help` or `--h` option:

```
% xmlformat --help
Usage: xmlformat [options] xml-file

Options:
--help, -h
    Print this message
--backup suffix -b suffix
    Back up the input document, adding suffix to the input
    filename to create the backup filename.
--canonized-output
    Proceed only as far as the document canonization stage,
    printing the result.
--check-parser
    Parse the document into tokens and verify that their
    concatenation is identical to the original input document.
    This option suppresses further document processing.
--config-file file_name, -f file_name
    Specify the configuration filename. If no file is named,
    xmlformat uses the file named by the environment variable
    XMLFORMAT_CONF, if it exists, or ./xmlformat.conf, if it
    exists. Otherwise, xmlformat uses built-in formatting
    options.
--in-place, -i
    Format the document in place, replacing the contents of
    the input file with the reformatted document. (It's a
    good idea to use --backup along with this option.)
--show-config
    Show configuration options after reading configuration
    file. This option suppresses document processing.
--show-unconfigured-elements
    Show elements that are used in the document but for
    which no options were specified in the configuration
    file. This option suppresses document output.
--verbose, -v
    Be verbose about processing stages.
--version, -V
```

```
Show version information and exit.
```

Warning

Do not use the `--in-place` or `-i` reformatting option until you are certain your configuration options are set up the way you want. Unpleasant consequences may occur otherwise. For example, if you have verbatim elements that you have forgotten to declare as verbatim, they will be reformatted and you will have to restore them to their original state later. Use of the `--backup` or `-b` option can help you recover from this kind of problem.

xmlformat writes the result to the standard output by default. To perform an "in-place" conversion that writes the reformatted document back to the original file, use the `--in-place` or `-i` option. This is useful when you want to format multiple documents with a single command; streaming multiple output documents to the standard output concatenates them, which is likely not what you want.

Because in-place formatting replaces the original document, it's prudent to make a backup of the original using the `--backup` (or `-b`) option. This option takes a suffix value to be added to each input filename to produce the backup filename.

To inspect the default (built-in) configuration options, use this command:

```
% xmlformat --config-file=/dev/null --show-config
```

3. The Document Processing Model

XML documents consist primarily of elements arranged in nested fashion. Elements may also contain text. **xmlformat** acts to rearrange elements by removing or adding line breaks and indentation, and to reformat text.

3.1. Document Components

XML elements within input documents may be of three types:

- block elements

  This is the default element type. The DocBook `<chapter>`, `<sect1>`, and `<para>` elements are examples of block elements.

  Typically a block element will begin a new line. (That is the default formatting behavior, although **xmlformat** allows you to override it.)

  Spacing between sub-elements can be controlled, and sub-elements can be indented. Whitespace in block element text may be normalized. If normalization is in effect, line-wrapping may be applied as well. Normalization and line-wrapping may be appropriate for a block element with mixed content (such as `<para>`).

- inline elements

  These are elements that are contained within a block or within other inlines. The DocBook `<emphasis>` and `<literal>` elements are examples of inline elements.

  Normalization and line-wrapping of inline element tags and content is handled the same way as for the enclosing block element. In essence, an inline element is treated as part of parent's "text" content.

- verbatim elements

No formatting is done for verbatim elements. The DocBook `<programlisting>` and `<screen>` elements are examples of verbatim elements.

Verbatim element content is written out exactly as it appears in the input document. This also applies to child elements. Any formatting that would otherwise be performed on them is suppressed when they occur within a verbatim element.

xmlformat never reformats element tags. In particular, it does not change whitespace betweeen attributes or which attribute values. This is true even for inline tags within line-wrapped block elements.

xmlformat handles empty elements as follows:

- If an element appears as `<abc/>` in the input document, it is written as `<abc/>`.
- If an element appears as `<abc></abc>`, it is written as `<abc></abc>`. No line break is placed between the two tags.

XML documents may contain other constructs besides elements and text:

- Processing instructions
- Comments
- `DOCTYPE` declaration
- `CDATA` sections

xmlformat handles these constructs much the same way as verbatim elements. It does not reformat them.

3.2. Line Breaks and Indentation

Line breaks within block elements are controlled by the `entry-break`, `element-break`, and `exit-break` formatting options. A break value of *n* means *n* newlines. (This produces *n*-1 blank lines.)

Example. Suppose input text looks like this:

```
<elt>
<subelt/> <subelt/> <subelt/>
</elt>
```

Here, an `<elt>` element contains three nested `<subelt>` elements, which for simplicity are empty.

This input can be formatted several ways, depending on the configuration options. The following examples show how to do this.

1. To produce output with all sub-elements are on the same line as the `<elt>` element, add a section to the configuration file that defines `<elt>` as a block element and sets all its break values to 0:

   ```
   elt
     format          block
     entry-break     0
     exit-break      0
     element-break   0
   ```

   Result:

```
<elt><subelt/><subelt/><subelt/></elt>
```

2. To leave the sub-elements together on the same line, but on a separate line between the `<elt>` tags, leave the `element-break` value set to 0, but set the `entry-break` and `exit-break` values to 1. To suppress sub-element indentation, set `subindent` to 0.

```
elt
  format          block
  entry-break     1
  exit-break      1
  element-break   0
  subindent       0
```

Result:

```
<elt>
<subelt/><subelt/><subelt/>
</elt>
```

3. To indent the sub-elements, make the `subindent` value greater than zero.

```
elt
  format          block
  entry-break     1
  exit-break      1
  element-break   0
  subindent       2
```

Result:

```
<elt>
  <subelt/><subelt/><subelt/>
</elt>
```

4. To cause the each sub-element begin a new line, change the `element-break` to 1.

```
elt
  format          block
  entry-break     1
  exit-break      1
  element-break   1
  subindent       2
```

Result:

```
<elt>
  <subelt/>
  <subelt/>
```

```
  <subelt/>
</elt>
```

5. To add a blank line between sub-elements, increase the `element-break` from 1 to 2.

```
elt
  format          block
  entry-break     1
  exit-break      1
  element-break   2
  subindent       2
```

Result:

```
<elt>
  <subelt/>

  <subelt/>

  <subelt/>
</elt>
```

6. To also produce a blank line after the `<elt>` opening tag and before the closing tag, increase the `entry-break` and `exit-break` values from 1 to 2.

```
elt
  format          block
  entry-break     2
  exit-break      2
  element-break   2
  subindent       2
```

Result:

```
<elt>

  <subelt/>

  <subelt/>

  <subelt/>

</elt>
```

7. To have blank lines only after the opening tag and before the closing tag, but not have blank lines between the sub-elements, decrease the `element-break` from 2 to 1.

```
elt
  format          block
  entry-break     2
  exit-break      2
```

```
  element-break   1
  subindent       2
```

Result:

```
<elt>

  <subelt/>
  <subelt/>
  <subelt/>

</elt>
```

Breaks within block elements are suppressed in certain cases:

- Breaks apply to nested block or verbatim elements, but not to inline elements, which are, after all, inline. (If you really want an inline to begin a new line, define it as a block element.)

- Breaks are not applied to text within non-normalized blocks. Non-normalized text should not be changed, and adding line breaks changes the text.

  For example if `<x>` elements are normalized, you might elect to format this:

  ```
  <x>This is a sentence.</x>
  ```

  Like this:

  ```
  <x>
  This is a sentence.
  </x>
  ```

  Here, breaks are added before and after the text to place it on a separate line. But if `<x>` is not normalized, the text content will be written as it appears in the input, to avoid changing it.

3.3. Text Handling

The XML standard considers whitespace nodes insignificant in elements that contain only other elements. In other words, for elements that have element content, sub-elements may optionally be separated by whitespace, but that whitespace is insignificant and may be ignored.

An element that has mixed content may have text (`#PCDATA`) content, optionally interspersed with sub-elements. In this case, whitespace-only nodes may be significant.

xmlformat treats only literal whitespace as whitespace. This includes the space, tab, newline (linefeed), and carriage return characters. **xmlformat** does not resolve entity references, so entities such as `&#32;` or `&#x20;` that represent whitespace characters are seen as non-whitespace text, not as whitespace.

xmlformat doesn't know whether a block element has element content or mixed content. It handles text content as follows:

- If an element has element content, it will have only sub-elements and possibly all-whitespace text nodes. In this case, it is assumed that you'll want to control line-break behavior between sub-elements, so that

the (all-whitespace) text nodes can be discarded and replaced with the proper number of newlines, and possibly indentation.

- If an element has mixed content, you may want to leave text nodes alone, or you may want to normalize (and possibly line-wrap) them. In **xmlformat**, normalization converts runs of whitespace characters to single spaces, and discards leading and trailing whitespace.

To achieve this kind of formatting, **xmlformat** recognizes `normalize` and `wrap-length` configuration options for block elements. They affect text formatting as follows:

- You can enable or disable text normalization by setting the `normalize` option to `yes` or `no`.
- Within a normalized block, runs of whitespace are converted to single spaces. Leading and trailing whitespace is discarded. Line-wrapping and indenting may be applied.
- In a non-normalized block, text nodes are not changed as long as they contain any non-whitespace characters. No line-wrapping or indenting is applied. However, if a text node contains only whitespace (for example, a space or newline between sub-elements), it is assumed to be insignficant and is discarded. It may be replaced by line breaks and indentation when output formatting occurs.

Consider the following input:

```
<row> <cell> A </cell> <cell> B </cell> </row>
```

Suppose that the `<row>` and `<cell>` elements both are to be treated as non-normalized. The contents of the `<cell>` elements are text nodes that contain non-whitespace characters, so they would not be reformatted. On the other hand, the spaces between tags are all-whitespace text nodes and are not significant. This means that you could reformat the input like this:

```
<row><cell> A </cell><cell> B </cell></row>
```

Or like this:

```
<row>
<cell> A </cell><cell> B </cell>
</row>
```

Or like this:

```
<row>
  <cell> A </cell>
  <cell> B </cell>
</row>
```

In each of those cases, the whitespace between tags was subject to reformatting, but the text content of the `<cell>` elements was not.

The input would *not* be formatted like this:

```
<row><cell>A</cell><cell>B</cell></row>
```

Or like this:

```
<row>
  <cell>
    A
  </cell>
  <cell>
   B
  </cell>
</row>
```

In both of those cases, the text content of the `<cell>` elements has been modified, which is not allowed within non-normalized blocks. You would have to declare `<cell>` to have a `normalize` value of `yes` to achieve either of those output styles.

Now consider the following input:

```
<para> This is a        sentence. </para>
```

Suppose that `<para>` is to be treated as a normalized element. It could be reformatted like this:

```
<para>This is a sentence.</para>
```

Or like this:

```
<para>
This is a sentence.
</para>
```

Or like this:

```
<para>
  This is a sentence.
</para>
```

Or even (with line-wrapping) like this:

```
<para>
  This is a
  sentence.
</para>
```

The preceding description of normalization is a bit oversimplified. Normalization is complicated by the possibility that non-normalized elements may occur as sub-elements of a normalized block. In the following example, a verbatim block occurs in the middle of a normalized block:

```
<para>This is a paragraph that contains
<programlisting>
a code listing
</programlisting>
in the middle.
</para>
```

In general, when this occurs, any whitespace in text nodes adjacent to non-reformatted nodes is discarded.

There is no "preserve all whitespace as is" mode for block elements. Even if normalization is disabled for a block, any all-whitespace text nodes are considered dispensible. If you really want all text within an element to be preserved intact, you should declare it as a verbatim element. (Within verbatim elements, nothing is ever reformatted, so whitespace is significant as a result.)

If you want to see how **xmlformat** handles whitespace nodes and text normalization, invoke it with the `--canonized-output` option. This option causes **xmlformat** to display the document after it has been canonized by removing whitespace nodes and performing text normalization, but before it has been reformatted in final form. By examining the canonized document, you can see what effect your configuration options have on treatment of the document before line-wrapping and indentation is performed and line breaks are added.

4. Using Configuration Files

An **xmlformat** configuration file specifies formatting options to be associated with particular elements in XML documents. For example, you can format `<itemizedlist>` elements differently than `<orderedlist>` elements. (However, you cannot format `<listitem>` elements differentially depending on the type of list in which they occur.) You can also specify options for a "pseudo-element" named `*DEFAULT`. These options are applied to any element for which the options are not specified explicitly.

The following sections describe the general syntax of configuration files, then discuss the allowable formatting options that can be assigned to elements.

4.1. Configuration File Syntax

A configuration file consists of sections. Each section begins with a line that names one or more elements. (Element names do not include the "<" and ">" angle brackets.) The element line is followed by option lines that each name a formatting option and its value. Each option is applied to every element named on its preceding element line.

Element lines and option lines are distinguished based on leading whitespace (space or tab characters):

- Element lines have no leading whitespace.
- Option lines begin with at least one whitespace character.

On element lines that name multiple elements, the names should be separated by spaces or commas. These are legal element lines:

```
para title
para,title
para, title
```

On option lines, the option name and value should be separated by whitespace and/or an equal sign. These are legal option lines:

```
normalize yes
normalize=yes
normalize = yes
```

Blank lines are ignored.

Lines that begin "#" as the first non-white character are taken as comments and ignored. Comments beginning with "#" may also follow the last element name on an element line or the option value on an option line.

Example configuration file:

```
para
  format        block
  entry-break   1
  exit-break    1
  normalize     yes
  wrap-length   72

literal replaceable userinput command option emphasis
  format        inline

programlisting
  format        verbatim
```

It is not necessary to specify all of an element's options at the same time. Thus, this configuration file:

```
para, title
  format block
  normalize yes
title
  wrap-length 50
para
  wrap-length 72
```

Is equivalent to this configuration file:

```
para
  format block
  normalize yes
  wrap-length 72
title
  format block
  normalize yes
  wrap-length 50
```

If an option is specified multiple times for an element, the last value is used. For the following configuration file, `para` ends up with a `wrap-length` value of 68:

```
para
  format        block
  wrap-length   60
  wrap-length   72
para
  wrap-length   68
```

To continue an element line onto the next line, end it with a backslash character. **xmlformat** will interpret the next line as containing more element names for the current section:

```
chapter appendix article \
section simplesection \
sect1 sect2 sect3 \
sect4 sect5
  format        block
  entry-break   1
  element-break 2
  exit-break    1
  normalize     no
  subindent     0
```

Continuation can be useful when you want to apply a set of formatting options to a large number of elements. Continuation lines are allowed to begin with whitespace (though it's possible they may appear to the casual observer to be option lines if they do).

Continuation is not allowed for option lines.

A configuration file may contain options for two special "pseudo-element" names: `*DOCUMENT` and `*DEFAULT`. (The names begin with a "*" character so as not to conflict with valid element names.)

`*DEFAULT` options apply to any element that appears in the input document but that was not configured explicitly in the configuration file.

`*DOCUMENT` options are used primarily to control line breaking between top-level nodes of the document, such as the XML declaration, the `DOCTYPE` declaration, the root element, and any comments or processing instructions that occur outside the root element.

It's common to supply `*DEFAULT` options in a configuration file to override the built-in values. However, it's normally best to leave the `*DOCUMENT` options alone, except possibly to change the `element-break` value.

Before reading the input document, **xmlformat** sets up formatting options as follows:

1. It initializes the built-in `*DOCUMENT` and `*DEFAULT` options,

2. It reads the contents of the configuration file, assigning formatting options to elements as listed in the file.

   Note that although `*DOCUMENT` and `*DEFAULT` have built-in default values, the defaults they may be overridden in the configuration file.

3. After reading the configuration file, any missing formatting options for each element are filled in using the options from the `*DEFAULT` pseudo-element. For example, if `para` is defined as a block element but no `subindent` value is defined, `para` "inherits" the `subindent` value from the `*DEFAULT` settings.

Missing options are filled in from the `*DEFAULT` options only *after* reading the entire configuration file. For the settings below, `*DEFAULT` has a `subindent` value of 2 (not 0) after the file has been read. Thus, `para` also is assigned a `subindent` value of 2.

```
*DEFAULT
  subindent 0
para
  format block
```

```
  normalize yes
*DEFAULT
  subindent 2
```

4.2. Formatting Options

The allowable formatting options are as follows:

```
format {block | inline | verbatim}
entry-break n
element-break n
exit-break n
subindent n
normalize {no | yes}
wrap-length n
```

A value list shown as `{ value1 | value2 | ... }` indicates that the option must take one of the values in the list. A value shown as *n* indicates that the option must have a numeric value.

Details for each of the formatting options follow.

- `format {block | inline | verbatim}`

  This option is the most important, because it determines the general way in which the element is formatted, and it determines whether the other formatting options are used or ignored:

  - For block elements, all other formatting options are significant.
  - For inline elements, all other formatting options are ignored. Inline elements are normalized, wrapped, and indented according to the formatting options of the enclosing block element.
  - For verbatim elements, all other formatting options are ignored. The element content is written out verbatim (literally), without change, even if it contains other sub-elements. This means no normalization of the contents, no indenting, and no line-wrapping. Nor are any breaks added within the element.

  A configuration file may *specify* any option for elements of any type, but **xmlformat** will ignore inapplicable options. One reason for this is to allow you to experiment with changing an element's format type without having to disable other options.

  If you use the `--show-config` command-line option to see the configuration that **xmlformat** will use for processing a document, it displays only the applicable options for each element.

- `entry-break n`

  `element-break n`

  `exit-break n`

  These options indicate the number of newlines (line breaks) to write after the element opening tag, between child sub-elements, and before the element closing tag. They apply only to block elements.

  A value of 0 means "no break". A value of 1 means one newline, which causes the next thing to appear on the next line with no intervening blank line. A value *n* greater than 1 produces *n*-1 intervening blank lines. Some examples:

- An `entry-break` value of 0 means the next token will appear on same line immediately after the opening tag.
- An `exit-break` value of 0 means the closing tag will appear on same line immediately after the preceding token.

- `subindent` *n*

  This option indicates the number of spaces by which to indent child sub-elements, relative to the indent of the enclosing parent. It applies only to block elements. The value may be 0 to suppress indenting, or a number *n* greater than 0 to produce indenting.

  This option does not affect the indenting of the element itself. That is determined by the `subindent` value of the element's own parent.

  Note: `subindent` does not apply to text nodes in non-normalized blocks, which are written as is without reformatting. `subindent` also does not apply to verbatim elements or to the following non-element constructs, all of which are written with no indent:

  - Processing instructions
  - Comments
  - `DOCTYPE` declarations
  - `CDATA` sections

- `normalize {no | yes}`

  This option indicates whether or not to perform whitespace normalization in text. This option is used for block elements, but it also affects inline elements because their content is normalized the same way as their enclosing block element.

  If the value is `no`, whitespace-only text nodes are not considered significant and are discarded, possibly to be replaced with line breaks and indentation.

  If the value is `yes`, normalization causes removal of leading and trailing whitespace within the element, and conversion of runs of whitespace characters (including line-ending characters) to single spaces.

  Text normalization is discussed in more detail in Section 3.3, “ Text Handling ”.

- `wrap-length` *n*

  Line-wrapping length. This option is used only for block elements and line-wrapping occurs only if normalization is enabled. The option affects inline elements because they are line-wrapped the same way as their enclosing block element.

  Setting the `wrap-length` option to 0 disables wrapping. Setting it to a value *n* greater than 0 enables wrapping to lines at most *n* characters long. (Exception: If a word longer than *n* characters occurs in text to be wrapped, it is placed on a line by itself. A word will never be broken into pieces.) The line length is adjusted by the current indent when wrapping is performed to keep the right margin of wrapped text constant. For example if the `wrap-length` value is 60 and the current indent is 10, lines are wrapped to a maximum of 50 characters.

  Any prevailing indent is added to the beginning of each line, unless the text will be written immediately following a tag on the same line. This can occur if the text occurs after the opening tag of the block and

the `entry-break` is 0, or the text occurs after the closing tag of a sub-element and the `element-break` is 0.

5. How xmlformat Works

Briefly, **xmlformat** processes an XML document using the following steps:

1. Read the document into memory as a single string.
2. Parse the document into a list of tokens.
3. Convert the list of tokens into nodes in a tree structure, tagging each node according to the token type.
4. Discard extraneous whitespace nodes and normalize text nodes. (The meaning of "normalize" is described in Section 3.3, “ Text Handling ”.)
5. Process the tree to produce a single string representing the reformatted document.
6. Print the string.

xmlformat is not an XSLT processor. In essence, all it does is add or delete whitespace to control line breaking, indentation, and text normalization.

xmlformat uses the REX parser developed by Robert D. Cameron (see Section 7, “ References ”). REX performs a parse based on a regular expression that operates on a string representing the XML document. The parse produces a list of tokens. REX does a pure lexical scan that performs no alteration of the text except to tokenize it. In particular:

- REX doesn't normalize any whitespace, including line endings. This is true for text elements, and for whitespace within tags (including between attributes and within attribute values). Any normalization or reformatting to be done is performed in later stages of **xmlformat** operation.
- REX leaves entity references untouched. It doesn't try to resolve them. This means it doesn't complain about undefined entities, which to my mind is an advantage. (A pretty printer shouldn't have to read a DTD or a schema.)
- If the XML is malformed, errors can be detected easily: REX produces error tokens that begin with "<" but do not end with ">".

xmlformat expects its input documents to be legal XML. It does not consider fixing broken documents to be its job, so if **xmlformat** finds error tokens in the result produced by REX, it lists them and exits.

Assuming the document contains no error tokens, **xmlformat** uses the token list to construct a tree structure. It categorizes each token based on its initial characters:

Initial Characters	**Token Type**
`<!--`	comment
`<?`	processing instruction (this includes the `<?xml?>` instruction)
`<!DOCTYPE`	`DOCTYPE` declaration
`<![`	`CDATA` section
`</`	element closing tag
`<`	element opening tag

Anything token not beginning with one of the sequences shown in the preceding table is a text token.

The token categorization determineas the node types of nodes in the document tree. Each node has a label that identifies the node type:

Label	Node Type
`comment`	comment node
`pi`	processing instruction node
`DOCTYPE`	`DOCTYPE` declaration node
`CDATA`	`CDATA` section node
`elt`	element node
`text`	text node

If the document is not well-formed, tree construction will fail. In this case, **xmlformat** displays one or more error messages and exits. For example, this document is invalid:

```
<p>This is a <strong>malformed document.</p>
```

Running that document through **xmlformat** produces the following result:

```
MISMATCH open (strong), close (p); malformed document?
Non-empty tag stack; malformed document?
Non-empty children stack; malformed document?
Cannot continue.
```

That is admittedly cryptic, but remember that it's not **xmlformat**'s job to repair (or even diagnose) bad XML. If a document is not well-formed, you may find Tidy a useful tool for fixing it up.

Tokens of each type except element tokens correspond to single distinct nodes in the document. Elements are more complex. They may consist of multiple tokens, and may contain children:

- An element with a combined opening/closing tag (such as `<abc/>`) consists of a single token.
- An element with separate opening and closing tags (such as `<abc>...</abc>`) consists of at least the two tags, plus any children that appear between the tags.

Element opening tag tokens include any attributes that are present, because **xmlformat** performs no tag reformatting. Tags are preserved intact in the output, including any whitespace between attributes or within attribute values.

In addition to the type value that labels a node as a given node type, each node has content:

- For all node types except elements, the content is the text of the token from which the node was created.
- For element nodes, the content is the list of child nodes that appear within the element. An empty element has an empty child node list. In addition to the content, element nodes contain other information:
  - The literal text of the opening and closing tags. If an element is written in single-tag form (`<abc/>`), the closing tag is empty.
  - The element name that is present in the opening tag. (This is maintained separately from the opening tag so that a pattern match need not be done on the opening tag each time it's necessary to determine the element name.)

After constructing the node tree, **xmlformat** performs two operations on it:

- The tree is "canonized" to normalize text nodes and to discard extraneous whitespace nodes. A whitespace node is a text node consisting of nothing but whitespace characters (space, tab, carriage return, linefeed (newline)). Decisions about which whitespace nodes are extraneous are based on the configuration options supplied to **xmlformat**.

- The canonized tree is used to produce formatted output. **xmlformat** performs line-wrapping of element content, and adds indentation and line breaks. Decisions about how to apply these operations are based on the configuration options.

Here's an example input document, representing a single-row table:

```
<table>
  <row>
    <cell>1</cell><cell>2</cell>
    <cell>3</cell>
  </row></table>
```

After reading this in and constructing the tree, the canonized output looks like this:

```
<table><row><cell>1</cell><cell>2</cell><cell>3</cell></row></table>
```

The output after applying the default formatting options looks like this:

```
<table>
 <row>
  <cell>1</cell>
  <cell>2</cell>
  <cell>3</cell>
 </row>
</table>
```

6. Prerequisites

xmlformat has very few prerequisites. It requires no extra modules other than an option-processing module. In particular:

- **xmlformat** requires no XML processing modules. XML parsing is done with a single (rather complex) regular expression developed by Robert D. Cameron. A paper that discusses development of this parsing expression is available; see Section 7, “ References ”.

- **xmlformat** requires no text-processing modules such as Text::Wrap. I tested Text::Wrap to see if it was suitable for **xmlformat**. It was not, for the following reasons:

  - If Text::Wrap encounters an individual word that is longer than the line length, older versions of Text::Wrap invoke die(). In newer versions, you can have long words left intact.

  - Text::Wrap converts runs of spaces in the leading indent to tabs. (Though this can be suppressed.)

  - Text::Wrap reformats inline tags (and may change attribute values). **xmlformat** preserves tags intact.

  In addition, the simple algorithm used by **xmlformat** appears to be about twice as fast as Text::Wrap (at least on Mac OS X).

7. References

1. Original REX paper by Robert D. Cameron:

   ```
   http://www.cs.sfu.ca/~cameron/REX.html
   ftp://fas.sfu.ca/pub/cs/TR/1998/CMPT1998-17.html
   ```

   This paper contains REX implementations in Perl, JavaScript, and LEX. The Perl implementation was used as the basis of XML parsing in **xmlformat.pl** and **xmlformat.rb**.

2. A Python implementation of REX:

   ```
   http://mail.python.org/pipermail/xml-sig/1999-November/001628.html
   ```

3. A PHP implementation of REX:

   ```
   http://traumwind.de/computer/php/REX/
   ```