XML Processing With `gawk`

A User's Guide for the XML extension of GNU Awk
Edition 1.2
February, 2017

Jürgen Kahrs with contributions from
Stefan Tramm, Manuel Collado and Andrew Schorr

Published by:

Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1335 USA
Phone: +1-617-542-5942
Fax: +1-617-542-2652
Email: gnu@gnu.org
URL: http://www.gnu.org/

Copyright (C) 2000–2002, 2004–2007, 2014, 2017 Free Software Foundation, Inc.

This is Edition 1.2 of *XML Processing With* `gawk`, for the 1.0.4 (or later) version of the XML extension of the GNU implementation of AWK.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "GNU General Public License", with the Front-Cover Texts being "A GNU Manual", and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License".

a. The FSF's Back-Cover Text is: "You have the freedom to copy and modify this GNU manual."

Table of Contents

Preface 1
Foreword to Edition 0.3 2
Foreword to Edition 1.2 3

1 AWK and XML Concepts 5
1.1 How does XML fit into AWK's execution model ? 5
1.2 How to traverse the tree with gawk 8
1.3 Looking closer at the XML file 9

2 Reading XML Data with POSIX AWK 11
2.1 Steve Coile's xmlparse.awk script 11
2.2 Jan Weber's getXML script 15
2.3 A portable subset of `gawk-xml` 17
2.3.1 Converting a script from `gawk-xml` into portable subset . . . 17
2.3.2 Converting a script from portable subset into `gawk-xml` . . . 18

3 XML Core Language Extensions of gawk 19
3.1 Checking for well-formedness 19
3.2 Printing an outline of an XML file 20
3.3 Pulling data out of an XML file 21
3.4 Character data and encoding of character sets 23
3.5 Dealing with DTDs 26
3.6 Sorting out all kinds of data from an XML file 28

4 Some Convenience with the xmllib library . . . 31
4.1 Introduction Examples 31
4.2 Main features 32
Character Data (CDATA) 33
Start- and End-elements (SE, EE, PATH, ATTR[]) 33
Comments (CM) 33
Processing Instructions (PI) 33
Real Character Data (XmlCDATA) 33
grep function 34
XmlStartElement and XmlEndElement functions 34
XmlPathTail function 34
XmlTraceAttr function 34
Simple String manipulation functions 34
Minor Issues 35
4.3 Usage of xmllib.awk 35
Ad hoc Queries (grep-like tools) 36
Formatter and Converter (sed-like tools) 36
Comparison to XSLT 37

5 DOM-like access with the xmltree library ... 41

6 Problems from the newsgroups comp.text.xml and comp.lang.awk ... 43

6.1 Extract the elements where i="Y" ... 43
6.2 Convert XMLTV file to tabbed ASCII ... 44
6.3 Finding the minimum value of a set of data ... 46
6.4 Updating DTD to agree with its use in doc's ... 48
6.5 Working with XML paths ... 48

7 Some Advanced Applications ... 51

7.1 Copying and Modifying with the `xmlcopy.awk` library script ... 51
7.2 Reading an RSS news feed ... 53
7.3 Using a service via SOAP ... 56
7.4 Loading XML data into PostgreSQL ... 62
7.5 Converting XML data into tree drawings ... 67
7.6 Generating a DTD from a sample file ... 70
7.7 Generating a recursive descent parser from a sample file ... 73
7.8 A parser for Microsoft Excel's XML file format ... 78

8 Reference of XML features ... 81

8.1 XML features built into the gawk interpreter ... 81
8.1.1 `XMLDECLARATION`: integer indicates begin of document ... 81
8.1.2 `XMLMODE`: integer for switching on XML processing ... 81
8.1.3 `XMLSTARTELEM`: string holds tag upon entering element ... 82
8.1.4 `XMLATTR`: array holds attribute names and values ... 82
8.1.5 `XMLENDELEM`: string holds tag upon leaving element ... 83
8.1.6 `XMLCHARDATA`: string holds character data ... 83
8.1.7 `XMLPROCINST`: string holds processing instruction target ... 83
8.1.8 `XMLCOMMENT`: string holds comment ... 84
8.1.9 `XMLSTARTCDATA`: integer indicates begin of CDATA ... 84
8.1.10 `XMLENDCDATA`: integer indicates end of CDATA ... 84
8.1.11 `LANG`: env variable holds default character encoding ... 84
8.1.12 `XMLCHARSET`: string holds current character set ... 85
8.1.13 `XMLSTARTDOCT`: root tag name indicates begin of DTD ... 85
8.1.14 `XMLENDDOCT`: integer indicates end of DTD ... 86
8.1.15 `XMLUNPARSED`: string holds unparsed characters ... 86
8.1.16 `XMLERROR`: string holds textual error description ... 86
8.1.17 `XMLROW`: integer holds current row of parsed item ... 86
8.1.18 `XMLCOL`: integer holds current column of parsed item ... 86
8.1.19 `XMLLEN`: integer holds length of parsed item ... 86
8.1.20 `XMLDEPTH`: integer holds nesting depth of elements ... 86
8.1.21 `XMLPATH`: string holds nested tags of parsed elements ... 87
8.1.22 `XMLENDDOCUMENT`: integer indicates end of XML data ... 87
8.1.23 `XMLEVENT`: string holds name of event ... 87
8.1.24 `XMLNAME`: string holds name assigned to XMLEVENT ... 87

8.2 `gawk-xml` Core Language Interface Summary 88
8.2.1 Verbose Interface - One dedicated predefined variable for each event class: XML*eventname* 88
8.2.2 Concise Interface - Reduced set of variables shared by all events 88
8.3 xmllib 90
8.4 xmlbase 91
NAME 91
USAGE 91
DESCRIPTION 91
NOTES 91
LIMITATIONS 91
8.5 xmlcopy 92
NAME 92
USAGE 92
DESCRIPTION 92
Token reconstruction 92
Token modification 92
NOTES 92
LIMITATIONS 92
8.6 xmlsimple 93
NAME 93
USAGE 93
DESCRIPTION 93
Short token variable names 93
Collecting character data 94
Whitespace handling 94
Record ancestors information 94
Path related functions 94
Grep-like facilities 95
NOTES 95
LIMITATIONS 95
8.7 xmltree 96
NAME 96
USAGE 96
DESCRIPTION 96
Automatic storage of the element tree 96
Processing the tree in the END clause 96
Printing tree fragments 96
Selecting tree fragments 97
The path expression language 97
NOTES 98
LIMITATIONS 98
8.8 xmlwrite 99
NAME 99
USAGE 99
DESCRIPTION 99
Output file and mode 99

XML prologue 100
Processing Instructions and Comments 100
Elements and attributes 100
Character data 100
Unparsed markup 101
Higher level convenience functions 101
Integration with the XML extension 101
NOTES 101
LIMITATIONS 101

9 Reference of Books and Links 103

9.1 Good Books 103
9.2 Links to the Internet 104

Appendix A GNU Free Documentation License .. 105

Index 113

Preface

In June of 2003, I was confronted with some textual configuration files in XML format and I was scared by the fact that my favorite tools (`grep` and `awk`) turned out to be mostly useless for extracting information from these files. It looked as if AWK's way of processing files line by line had to be replaced by a node-traversal of tree-like XML data. For the first implementation of an extended `gawk`, I chose the `expat` library to help me reading XML files.

With a little help from Stefan Tramm I went on selecting features and implemented what is now called XMLgawk over the Christmas Holidays 2003. In June 2004, Manuel Collado joined us and started collecting his comments and proposals for the extension. Manuel also wrote a library for reading XML files into a DOM-like structure.

In Septermber 2004, I wrote the first version of this book. Andrew Schorr flooded my mailbox with patches and suggestions for changes. His initiative pushed me into starting the SourceForge project. This happened in March 2005 and since then, all software changes go into a CVS source tree at SourceForge (thanks to them for providing this service). Andrew's urgent need for a production system drove development in early 2005. Significant changes were made:

1. Parsing speed was doubled to increase efficiency when reading large data bases.
2. Manuel suggested and Andrew implemented some simplifications in user-visible patterns like `XMLEVENT` and `XMLNAME`.
3. Andrew encapsulated XMLgawk into a `gawk` extension, loadable as a dynamic library at runtime. This also allowed for building `gawk` without the XML extension. That's how the `-l` option and `@load` were introduced.
4. Andrew cleaned up the autotool mechanism (`Makefile.am` etc.) and found an installation mechanism which allows an easy and collision-free installation in the same directory as Arnold's GNU Awk. He also made Arnold's `igawk` obsolete by implementing the `-i` option. April 2005 saw the Alpha release of `xgawk`, as a branch of `gawk-3.1.4`.

In August 2005, Hirofumi Saito held a presentation at the *Lightweight Language Day and Night* (`http://ll.jus.or.jp/2005/files/lldn_awk_2005.pdf`) in Japan. His little slideshow demonstrated the importance of multibyte characters in any modern programming language. Hirofumi Saito also did the localization of our source code for the Japanese language. Kimura Koichi reported and fixed some problems in handling of multibyte characters. He also found ways to get all this running on several flavours of Microsoft Windows.

Meanwhile in Summer 2005, Arnold had released `gawk-3.1.5` and I applied all his 219 patches to our CVS tree over the Christmas Holidays 2005. Andrew applied some more bug fixes from the GNU mailing archive and so the current Beta release of `xgawk-3.1.5` is already a bit ahead of Arnold's `gawk-3.1.5`.

Jürgen Kahrs
Bremen, Germany
April, 2006

Foreword to Edition 0.3

In August 2006, Arnold and friends set up a mirror of Arnold's source tree as a CVS repository at Savannah (`http://savannah.gnu.org/projects/gawk`). It is now much easier for us to understand recent changes in Arnold's source tree. We strive to merge all of them immediately to our source tree. This merge process has been enormously simplified by a weekly `cron` job mechanism (implemented by Andrew) that examines recent activities in Arnold's tree and sends an email to our mailing list.

Some more problems and fixes in handling multibyte characters have been reported by our Japanese friends to Arnold and us. For example, Hirofumi Saito and others forwarded patches for the half-width katakana characters in character classes in ShiftJIS locale.

Since January 2007, there is a new target `valgrind` in the top level `Makefile`. This feature was implemented for detection of memory leaks in the interpreter while running the regression test cases. We found small memory leaks in our `time` and `mpfr` extension instantly with this new feature.

March 2007 saw much activity. First we introduced Victor Paesa's new extension for the GD library. Then we merged Paul Eggert's file `floatcomp.c` (floating point / long comparison) from Arnold's source tree. We also merged the changes in regression test cases and documentation due to changed behaviour in numerical calculations (infinity and not a number) and formatting of these.

Stefan Tramm held a 5-minute *Lightning Talk on xgawk* at OSCON 06.

Hirofumi Saito took part in the *Lightweight Language Conference 2006*.

The new Chapter 2 [Reading XML Data with POSIX AWK], page 11, describes a template script `getXMLEVENT.awk` that allows us to write portable scripts in a manner that is a mostly compatible subset of the XMLgawk API. Such scripts can be run on any POSIX-compliant AWK interpreter – not just `xgawk`.

The new Section 7.1 [Copying and Modifying with the `xmlcopy.awk` library script], page 51, describes a library script for making slightly modified copies of XML data.

Thanks to Andrew's mechanism for systematic reporting of patches applied by Arnold to his `gawk-stable` tree, Andrew and I caught up with recent changes in Arnold's source tree. As a consequence, `xgawk` is now based upon the recent official `gawk-3.1.6` release.

Jürgen Kahrs
Bremen, Germany
December, 2007

Foreword to Edition 1.2

In October 2014 the `xgawk` project has been restructured. The set of `gawk` extensions has been splitted. There is now a separate directory and distribution archive for each individual `gawk` extension. The SourceForge project name has changed to `gawkextlib`.

XMLgawk is no longer the name of the whole set of extensions, nor of the individual XML extension. The XML extension is now called `gawk-xml`. The `xmlgawk` name designates a script that invokes gawk with the XML extension loaded and a convenience `xmllib.awk` included.

The 1.2 edition of this manual includes documentation of the companion `xml*.awk` libraries. The body of the manual has changed only a little, but has been revised in order to update obsolete names, references or versions of the related stuff.

Manuel Collado
February, 2017

FIXME: This document has not been completed yet. The incomplete portions have been marked with comments like this one.

1 AWK and XML Concepts

This chapter provides a (necessarily) brief intoduction to XML concepts. For many applications of `gawk` XML processing, we hope that this is enough. For more advanced tasks, you will need deeper background, and it may be necessary to switch to other tools like XSL processors

1.1 How does XML fit into AWK's execution model ?

But before we look at XML, let us first reiterate how AWK's program execution works and what to expect from XML processing within this framework. The `gawk` man page summarizes AWK's basic execution model as follows:

> An AWK program consists of a sequence of pattern-action statements and optional function definitions.
> *pattern* { *action statements* }
> *function name(parameter list)* { *statements* }
> . . . For each record in the input, gawk tests to see if it matches any pattern in the AWK program. For each pattern that the record matches, the associated action is executed. The patterns are tested in the order they occur in the program. Finally, after all the input is exhausted, gawk executes the code in the END block(s) (if any).

A look at a short and simple example will reveal the strength of this abstract description. The following script implements the Unix tool `wc` (well, almost, but not completely).

```
BEGIN { words=0 }
{ words+=NF }
END { print NR, words }
```

Before opening the file to be processed, the word counter is initialized with 0. Then the file is opened and for each line the number of fields (which equals the number of words) is added to the current word counter. After reading all lines of the file, the resulting word counter is printed as well as the number of lines.

Store the lines above in a file named `wc.awk` and invoke it with

```
gawk -f wc.awk datafile.xml
```

This kind of invocation will work on all platforms. In a Unix environment (or in the Cygwin Unix-emulation on top of Microsoft Windows) it is more comfortable to store the script above into an executable file. To do so, write a file named `wc.awk`, with the first line being

```
#!/usr/bin/gawk -f
```

followed by the lines above. Then make the file `wc.wk` executable with

```
chmod a+x wc.awk
```

and invoke it as

```
wc.awk datafile.xml
```

When looking at Figure 1.1 from top to bottom, you will recognize that each line of the data file is represented by a row in the figure. In each row you see `NR` (the number of the

current line) on the left and the pattern (the condition for execution) and its action on the right. The first and last rows represent BEGIN (initialization) and END (finalization).

Input data	NR **(given)**	**Pattern**	**Action**
Before reading	*undefined*	BEGIN	words=0
Line	1		words+=NF
Line	2		words+=NF
Line	3		words+=NF
Line	...		words+=NF
After reading	*total lines*	END	print NR, words

Figure 1.1: Execution model of an AWK program with ASCII data, proceeding top to bottom

We could use this script to process any XML file. But the result it yielded would not be too meaningful to us. When processing XML files, you are not really interested in the number of lines or words. Take, for example, this XML file, a DocBook file (http://xml.web.cern.ch/XML/goossens/dbatcern/doc-structure.html#exa.dbgeneral) to be precise.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
 "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">

<book id="hello-world" lang="en">

<bookinfo>
<title>Hello, world</title>
</bookinfo>

<chapter id="introduction">
<title>Introduction</title>
<para>This is the introduction. It has two sections</para>

<sect1 id="about-this-book">
<title>About this book</title>
<para>This is my first DocBook file.</para>
</sect1>

<sect1 id="work-in-progress">
<title>Warning</title>
<para>This is still under construction.</para>
</sect1>

</chapter>
</book>
```

Figure 1.2: Example of some XML data (DocBook file)

Reading through this jungle of angle brackets, you will notice that the notion of a line is not an adequate concept to describe what you see. AWK's idea of *records* and *fields* only makes sense in a rectangular world of textual data being stored in rows and columns. This notion is blind to XML's notion of structuring textual data into markup blocks (like `<title>Introduction</title>`), with beginning and ending being marked as such by angle brackets. Furthermore, XML's markup blocks can contain other blocks (like a `chapter` contains a `title` and a `para`). XML sees textual data as a tree with deeply nested nodes (markup blocks). A tree is a dynamic data structure; some people call it a *recursive* structure because a tree contains other trees, which may contain even other trees. These sub-trees are not numbered (as rows and columns) but they have names. Now that we have a coarse understanding of the structure of an XML file, we can choose an adequate way of picturing the situation. XML data has a tree structure, so let's draw the example file in Figure 1.2 above as a tree (see Figure 1.3).

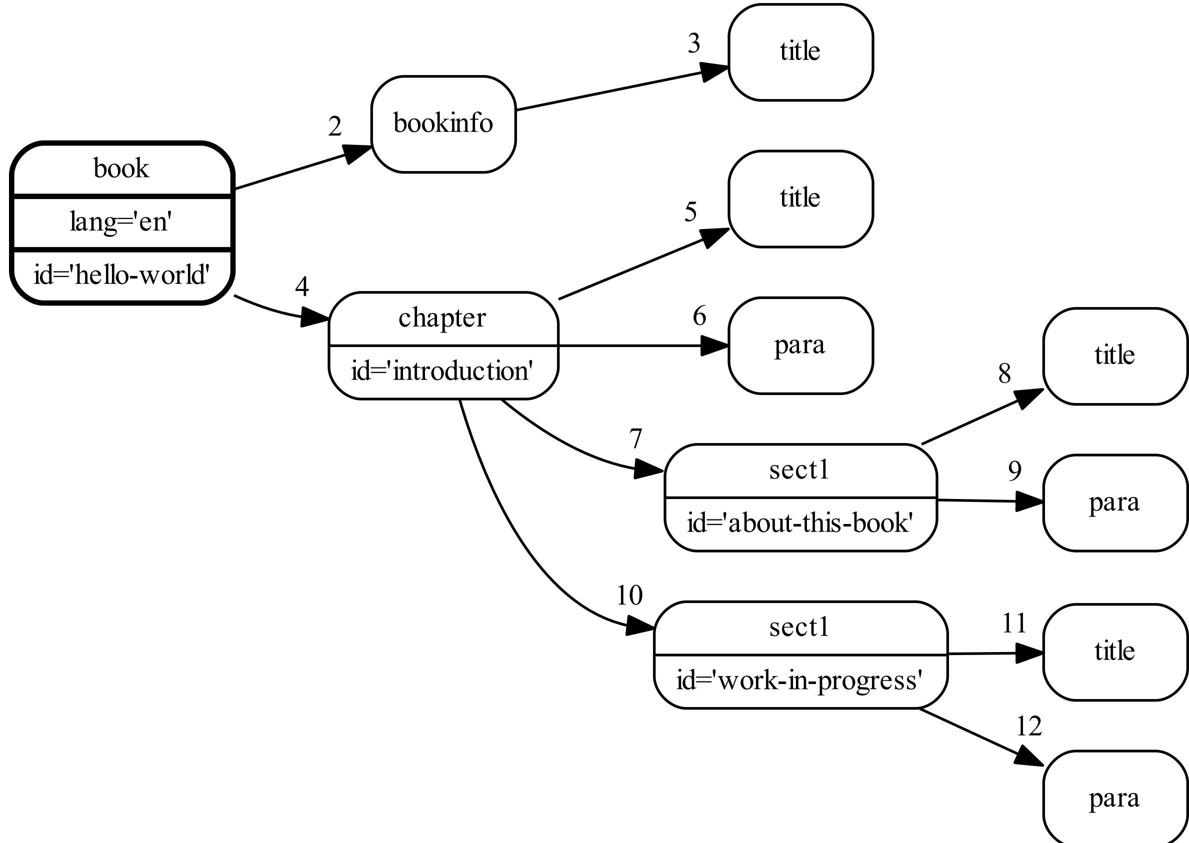

Figure 1.3: XML data (DocBook file) as a tree

You can easily see that each markup block is drawn as a node in this tree. The edges in the tree reveal the nesting of the markup blocks in a much more lucid way than the textual representation. Each edge indicates that the markup block which has an arrow pointing to it, is contained in the markup block from which which the edge comes. Such edges indicate the "*parent-child*" relationship.

1.2 How to traverse the tree with gawk

Now, what could be the equivalent of a `wc` command when dealing with such trees of markup blocks ? We could count the nodes of the tree. You can store and invoke the following script in the same way as you did for the previous script.

```
BEGIN        { nodes = 0   }
XMLSTARTELEM { nodes ++    }
END          { print nodes }
```

If you invoke this script with the data file in Figure 1.2, the number of nodes will be printed immediately:

```
gawk -l xml -f node_count.awk dbfile.xml
12
```

Notice the similarity between this example script and the original `wc.awk` which counts words. Instead of going over the lines, this script traverses the tree and increments the node counter each time a node is found. After a closer look you will find several differences between the previous script and the present one:

1. The command line for `gawk` has an additional parameter `-l xml`. This is necessary for loading the XML extension into the `gawk` interpreter so that the `gawk` interpreter knows that the file to be opened is an XML file and has to be treated differently.
2. The node counting happens in an action which has a pattern. Unlike the previous script (which counted on *every* line) we are interested in counting the nodes only. The occurence of a node (the beginning of a markup block) is indicated by the `XMLSTARTELEM` pattern.
3. There is no equivalent of the word count here, only the node count.
4. It is not clear in which order the nodes of the tree are traversed. The `bookinfo` node and the `chapter` node are both positioned directly under the `book` node; but which is counted first ? The answer becomes clear when we return to the textual representation of the tree — textual order induces traversal order.

Do you see the numbers near the arrow heads ? These are the numbers indicating traversal order. The number 1 is missing because it is clear that the root node (framed with a bold line) is visited first. Computer Scientists call this traversal order *depth-first* (`http://en.wikipedia.org/wiki/Depth-first`) because at each node, its children (the deeper nodes) are visited before going on with nodes at the same level. There are other orders of traversal (*breadth-first* (`http://en.wikipedia.org/wiki/Breadth-first_search`)) but the textual order in Figure 1.2 enforces the numbering in Figure 1.3.

The tree in Figure 1.3 is not balanced. The very last nodes are nested so deep that they are printed on the very right of the margin in Figure 1.3. This is not the case for the upper part of the drawing. Sometimes it is useful to know the maximum depth of such a tree. The following script traverses all nodes and at each node it compares actual depth and maximum depth to find and remember the largest depth.

```
@load "xml"
XMLSTARTELEM {
  depth++
  if (depth > max_depth)
    max_depth = depth
}
XMLENDELEM   { depth-- }
END { print max_depth }
```

Figure 1.4: Finding the maximum depth of the tree representation of an XML file with the script `max_depth.awk`

If you compare this script to the previous one, you will again notice some subtle differences.

1. `@load "xml"` is a replacement for the `-l xml` on the command line. If the source text of your script is stored in an executable file, you should start the script with loading all extensions into the interpreter. The command line option `-l xml` should only be used as a shorthand notation when you are working with a one-line command line.
2. The variable `depth` is not initialized. This is not necessary because all variables in `gawk` have a value of 0 if they are used for the first time without a prior initialization.
3. The most important difference you will find is the new pattern `XMLENDELEM`. This is the counterpart of the pattern `XMLSTARTELEM`. One is true upon entering a node, the other is true upon leaving the node. In the textual representation, these patterns mark the beginning and the ending of a markup block. Each time the script enters a markup block, the `depth` counter is increased and each time a markup block is left, the `depth` counter is decreased.

Later we will learn that this script can be shortened even more by using the builtin variable `XMLDEPTH` which contains the nesting depth of markup blocks at any point in time. With the use of this variable, the script in Figure 1.4 becomes one of these one-liners which are so typical for daily work with `gawk`.

1.3 Looking closer at the XML file

If you already know the basics of XML terminology, you can skip this section and advance to the next chapter. Otherwise, we recommend studying the O'Reilly book XML in a Nutshell (`http://www.oreilly.com/catalog/xmlnut3/`), which is a good combination of tutorial and reference. Basic terminology can be found in chapter 2 (XML Fundamentals). If you prefer (free) online tutorials, then we recommend w3schools (`http://www.w3schools.com/xml/default.asp`). See Section 9.2 [Links to the Internet], page 104, for additional valuable material.

Before going on reading, you should make sure you know the meaning of the following terms. Instead of leaving you on your own with learning these terms, we will give an informal and insufficient explanation of each of the terms. Always refer to Figure 1.2 for an example and consider looking the term up in one of the sources given above.

- Tag: name of a node
- Attribute: variable having a name (`lang`) and a value (`en`)

- Element: sub-tree, for example `bookinfo` including `title`
- Well-Formed: properly nested file; one tree with quoted, tag-wise distinct attributes
- DTD: formal description about which elements and attributes a file contains
- Schema: same use as DTD, but more detailed and formally itself XML (unlike DTD)
- Valid: conforming to a formal specification, usually given as a DTD or a Schema
- Processing Instruction: screwed special-purpose element whose name is `"?"`; first data line often is

  ```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  ```

- Character Data: textual data inside an element between the tags
- Mixed Content: element that has character data inside it
- Encoding: name of a mapping between text symbols and byte sequence (ISO-8859-1)
- UTF-8: default encoding of XML; covers all text symbols available, possibly multi-byte

Still reading ? Be warned that these definitions are formally incorrect. They are meant to get you on the right track. Each ambitious propeller head will happily tear these definitions apart. If you are seriously striving to become an XML propeller head yourself, then you should not miss reading the original defining documents about the XML technology (`http://www.w3.org/TR/2004/REC-xml-20040204/`). The proper playing ground for anxious aspirants is the newsgroup comp.text.xml (`news://comp.text.xml`). I am glad none of those propeller heads reads `gawk` books — they would kill me.

2 Reading XML Data with POSIX AWK

Some users will try to avoid the use of the new language features described earlier. They want to write portable scripts; they have to refrain from using features which are not part of the standardized POSIX AWK (`http://www.opengroup.org/onlinepubs/000095399/utilities/awk.html`). Since the XML extension of GNU Awk is not part of the POSIX standard, these users have to find different ways of reading XML data.

2.1 Steve Coile's xmlparse.awk script

Implementing a complete XML reader in POSIX AWK would mean that all subtle details of Unicode encodings had to be handled. It doesn't make sense to go into such details with an AWK script. But in 2001, Steve Coile wrote a parser which is good enough if your XML data consists of simple tagged blocks of ASCII characters. His script is available on the Internet as `xmlparse.awk` (`ftp://ftp.freefriends.org/arnold/Awkstuff/xmlparser.awk`). The source code of `xmlparse.awk` is well documented and ready-to-use for anyone having access to the Internet.

Begin your exploration of `xmlparse.awk` by downloading it. As of Summer 2007, there is a typo in the file that has to be corrected before you can start to work with the parser. Insert a hashmark character (`#`) in front of the comment in line 342.

```
wget ftp://ftp.freefriends.org/arnold/Awkstuff/xmlparser.awk
vi xmlparser.awk
342G
i#
ESC
:wq
```

While you're editing the parser script, have a look at the comments. This is a well-documented script that explains its implementation as well as some use cases. For example, the header summarizes almost all details that a user will need to remember (see Figure 2.1). There is a negligible inconsistency in the header: The file is really named `xmlparser.awk` and not `xmlparse.awk` as stated in the header. From a user's perspective, the most important constraint to keep in mind is that this XML parser needs a *modern* variant of AWK. This means a POSIX compliant AWK; the old Solaris implentation `oawk` will not be able to interpret this XML parser script as intended. Invoke the XML parser for the first time with

```
awk -f xmlparser.awk docbook_chapter.xml
```

Compare the output to the original file's content (see Figure 1.2) and its depiction as a tree (see Figure 1.3). You will notice that the first column of the output always contains the type of the items as they were parsed sequentially:

```
pi xml version="1.0" encoding="UTF-8"
data \n
decl DOCTYPE book PUBLIC "-//OASIS//DTD DocBook..."\n "http://www.oasis-open..."
data \n\n
begin BOOK
attrib id
value hello-world
```

attrib lang
value en
data \n \n
begin BOOKINFO
data \n
begin TITLE
data Hello, world
end TITLE
... *etc.* ...

This is in accordance with the guiding principles explained in the header of the parser script. Note that the description in Figure 2.1 is incomplete. More details will be provided below.

The script parses the XML data and saves each parsed item in two arrays:

- type[3] indicates the type of the 3rd parsed XML data item. This may be any of
  - "**error**" when an invalid item has been parsed or another error has occurred. In this case, item[3] contains the text of the error message.
  - "**begin**" when an opening tag has been parsed.
  - "**end**" when a closing tag has been parsed. In the case of type[3] containing "begin" or "end", item[3] contains the name of the tag.
  - "**attrib**" when an attribute's name has been parsed.
  - "**value**" when an attribute's value has been parsed. In the case of type[3] containing "attrib" or "value", item[3] contains the attribute's name or value.
  - "**data**" when the data between opening and closing tags has been parsed.
  - "**cdata**" when character data has been parsed.
  - "**comment**" when a comment has been parsed.
  - "**pi**" when a processing instruction has been parsed.
- item[3] contains data depending on type[3], as distinguished in the item list above.

While you proceed reading this book, you will notice that the basic idea is similar to what gawk-xml does. Especially the approach described in Section 8.2 [gawk-xml Core Language Interface Summary], page 88, as *Concise Interface - Reduced set of variables shared by all events* will look familiar. The script as it is was not designed to be a modular building block. Any application will *not* simply include the xmlparser.awk file, but copy it textually and modify the copy. Look into the original script once more and have a closer look at the final END pattern. You will find suggestions for several useful applications inside the END pattern.

```
##############################################################################
#
# xmlparse.awk - A simple XML parser for awk
#
# Author:  Steve Coile <scoile@csc.com>
#
# Version:  1.0 (20011017)
#
# Synopsis:
#
# awk -f xmlparse.awk [FILESPEC]...
#
# Description:
#
# This script is a simple XML parser for (modern variants of) awk.
# Input in XML format is saved to two arrays, "type" and "item".
#
# The term, "item", as used here, refers to a distinct XML element,
# such as a tag, an attribute name, an attribute value, or data.
#
# The indexes into the arrays are the sequence number that a
# particular item was encountered.  For example, the third item's
# type is described by type[3], and its value is stored in item[3].
#
# The "type" array contains the type of the item encountered for
# each sequence number.  Types are expressed as a single word:
# "error" (invalid item or other error), "begin" (open tag),
# "attrib" (attribute name), "value" (attribute value), "end"
# (close tag), and "data" (data between tags).
#
# The "item" array contains the value of the item encountered
# for each sequence number.  For types "begin" and "end", the
# item value is the name of the tag.  For "error", the value is
# the text of the error message.  For "attrib", the value is the
# attribute name.  For "value", the value is the attribute value.
# For "data", the value is the raw data.
#
# WARNING: XML-quoted values ("entities") in the data and attribute
# values are *NOT* unquoted; they are stored as-is.
#
###############################################################################
```
Figure 2.1: Usage explained in the header of `xmlparser.awk`

1. By checking for the occurence of an error with

   ```
   if (type[idx] == "error") {
   ...
   }
   ```

   it is quite easy to implement a script that checks for well-formedness of some XML data.

2. Several attempts have been made to introduce a *simplified* XML that is easier to parse by shell scripts. Simplication of the XML and output in a convenient line-by-line format can be implemented with the following code fragment inside an `END` pattern. It demonstrates how to go through all parsed items sequentially and handle each of the types appropriately.

   ```
   for ( n = 1; n <= idx; n += 1 ) {
     if        ( type[n] == "attrib"  ) {
     } else if ( type[n] == "begin"   ) {
     } else if ( type[n] == "cdata"   ) {
     } else if ( type[n] == "comment" ) {
     } else if ( type[n] == "data"    ) {
     } else if ( type[n] == "decl"    ) {
     } else if ( type[n] == "end"     ) {
     } else if ( type[n] == "error"   ) {
     } else if ( type[n] == "pi"      ) {
     } else if ( type[n] == "value"   ) {
     }
   }
   ```

3. One application of the framework just mentioned is an outline script like the one in Figure 3.2. Producing an outline output like the one in Figure 3.1 is a matter of a few lines in AWK if you modify the `xmlparser.awk` script. Notice that this is done after the complete XML data has been read. So, at the moment of processing, the complete XML data is somehow saved in AWK's memory, imposing some limit on the size of the data that can be processed.

   ```
   XMLDEPTH=0
   for (n = 1; n <= idx; n += 1 ) {
     if (       type[n] == "attrib") { printf(" %s=", item[n] )
     } else if (type[n] == "value" ) { printf("'%s'", item[n] )
     } else if (type[n] == "begin" ) { printf("\n%*s%s", 2*XMLDEPTH,"", item[n])
                                       XMLDEPTH ++
     } else if (type[n] == "end"   ) { XMLDEPTH -- }
   }
   ```

   If you compare the output of this application with Figure 3.1 you will notice only two differences. The first is the newline character before the very first tag; the second is the names of the tags. The `xmlparser.awk` script saves the names of the tags in uppercase letters, the exact tag name cannot be revovered without changing the internals of the XML parsing mechanism.

2.2 Jan Weber's getXML script

In 2005, Jan Weber posted a similar XML parser to the newsgroup comp.lang.awk (`news://comp.lang.awk`). You can use Google to search for the script `getXML` and copy it into a file. Unfortunately, Jan tried to make the script as short as possible and often put several statements on one line. Readability of the script has suffered severely and if you intend to analyse the script, be prepared that some editing may be necessary to understand it. Again, while you're editing the parser script, have a look at the comments. Jan has commented the one central function of the script from a user's perspective as follows (see Figure 2.2). The basic approach was taken over from the `xmlparser.awk` script. But there were several constraints Jan tried to satisfy in writing his XML parser:

1. The function `getXML` allows to read multiple XML files in parallel.
2. As a consequence, each XML event happens upon returning from the `getXML` function, similar to the `getline` mechanism of AWK (see Figure 2.3). Furthermore, the user application reads files in the `BEGIN` action of the AWK script, not in the `END` action.
3. The exact names of tags and attributes are preserved, no change in case is done by the XML parser.
4. Parameter passing resembles the approach described in Section 8.2 [`gawk-xml` Core Language Interface Summary], page 88, as *Concise Interface - Reduced set of variables shared by all events* much more closely. Most importantly, attribute names and values are passed along with the tag they belong to. So, granularity of events is more coarse and user-friendly.
5. While the `xmlparser.awk` script stored the complete XML data into two arrays during the parsing process, `getXML.awk` passes one XML event at a time back to the calling application, avoiding the unwanted waste of memory. This means, parsing large XML files becomes possible (although it doesn't make too much sense).
6. This XML parser runs with the `nawk` implementation of the AWK language that comes with the Solaris Operating System. As a consequence, this XML parser is probably the most portable of all parsers described in this book.

Again, we will demonstrate the usage of this XML parser by implementing an outline script like the one in Figure 3.2. Change the file `getXML` and replace the existing `BEGIN` action with the script in Figure 2.3. Invoke the new outline parser for the first time with

```
awk -f getXML docbook_chapter.xml
```

Compare the output to the original file's content (see Figure 1.2), its depiction as a tree (see Figure 1.3) and to the output of the original `outline` tool that comes with the `expat` parser (see Figure 3.1). The result is almost identical to Figure 3.1, except for one minor detail: The very first line is a blank line here.

```
##
# getXML( file, skipData ): # read next xml-data into XTYPE,XITEM,XATTR
# Parameters:
#   file       -- path to xml file
#   skipData   -- flag: do not read "DAT" (data between tags) sections
# External variables:
#   XTYPE      -- type of item read, e.g. "TAG"(tag), "END"(end tag), "COM"(comment),
#   XITEM      -- value of item, e.g. tagname if type is "TAG" or "END"
#   XATTR      -- Map of attributes, only set if XTYPE=="TAG"
#   XPATH      -- Path to current tag, e.g. /TopLevelTag/SubTag1/SubTag2
#   XLINE      -- current line number in input file
#   XNODE      -- XTYPE, XITEM, XATTR combined into a single string
#   XERROR     -- error text, set on parse error
# Returns:
#    1         on successful read: XTYPE, XITEM, XATTR are set accordingly
#    ""        at end of file or parse error, XERROR is set on error
# Private Data:
#   _XMLIO     -- buffer, XLINE, XPATH for open files
##
```

Figure 2.2: Usage of Jan Weber's `getXML` parser function

But some implementation details are noteworthy. Here, granularity of items is different: All attributes are reported along with their tag item. This results from a design decision: The `getXML` function uses several variables to pass larger amounts of data back to the caller. Finally a detail that did not become so obvious in this example. Notice the second parameter of the `getXML` function (`skipData`). Jan introduced an option that allows skipping textual data in between tags (mixed content).

```
#!/usr/bin/nawk -f

BEGIN {
  XMLDEPTH=0
  while ( getXML(ARGV[1],1) ) {
    if ( XTYPE == "TAG" ) {
      printf("\n%*s%s", 2*XMLDEPTH, "", XITEM)
      XMLDEPTH++
      for (attrName in XATTR)
        printf(" %s='%s'", attrName, XATTR[attrName])
    } else if ( XTYPE == "END" ) {
      XMLDEPTH--
    }
  }
}
```

Figure 2.3: Outlining an XML file with Jan Weber's `getXML` parser

2.3 A portable subset of `gawk-xml`

Jan Webers's portable script in the previous section was a significant advance over Steve Coile's script. Handling of XML events feels much more like it does in the `gawk-xml` API. But after some time of working with the script, the differences between it and the `gawk-xml` API become a bit annoying to remember. As a consequence, we took Jan's script, copied it into a new script file `getXMLEVENT.awk` and changed its inner working so as to minimize differences to the `gawk-xml` API. If you intend to use the script as a template for your own work, search for the file `getXMLEVENT.awk` in the following places:

- The `gawk-xml` distribution file contains a copy in the `awklib/xml` directory.
- If `gawk-xml` has already been installed on your host machine, a copy of the file should be in the directory of shared source (which usually resides at a place like `/usr/share/awk/` on GNU/Linux machines).

The file `getXMLEVENT.awk` as it is serves well if you want to start writing a script from scratch. It already contains an *event-loop* in the `BEGIN` pattern of the script. Just take the main body of the *event-loop* (the `while` loop) and change those parts that react on incoming events of the XML event stream.

But in the remainder of this section, we will assume that we already *have* a script and we intend to port it. Attempting to describe the approach in the most useful way, we will go through two typical use-cases of the `getXMLEVENT.awk` template file. First we look at the necessary steps for taking an existing script written for `gawk-xml` and making it portable for use on Solaris machines (to name just the worst case scenario). Secondly, we go the other way round: take an existing portable script and describe the necessary steps for converting it into an `gawk-xml` script.

2.3.1 Converting a script from `gawk-xml` into portable subset

The general approach in porting a script that uses `gawk-xml` features to a portable script is always the same. No matter if we port the original outline script (see Figure 3.2) or if we take a non-trivial application like the DTD generator (see Section 7.6 [Generating a DTD from a sample file], page 70). Now we proceed through the following series of steps.

1. We always start by first copying the template file `getXMLEVENT.awk` into a new file (`dtdgport.awk` in the case of the DTD generator).
2. Near the top of the new script file, remove the main body of the original event loop.
3. Replace the original event loop with the pattern-action pairs from the application. In the case of the DTD generator, take the first part of the source code (Figure 7.19) and insert the `XMLSTARTELEM` action into the event loop.
4. Append the `END` pattern of Figure 7.19 verbatim after the event loop.
5. Append the second part of the application (containing function declarations in Figure 7.20) verbatim.
6. Take the resulting application source file and try if it really works in the expected way. Compare the resulting output to Figure 7.18. You will find that the resulting output (a DTD) is indeed exactly the same.

```
awk -f dtdgport.awk docbook_chapter.xml
```

It is amazing how simple and effective it is to turn an `gawk-xml` script into a portable script. After all, you should never forget about the limitations of the portable script.

This tiny little XML parser is far from being a complete XML parser. Most notably, it misses the ability to read files with multi-byte characters and other Unicode encoding details. Experience tells us that sooner or later your tiny little parser will stumble across a customer- supplied XML file with special characters in it (copyright marks, typographic dashes, european accent characters, or even chinese characters). Then the need arises to port the script back to the full `gawk-xml` environment with its full XML parsing capability. When you eventually reach this point, continue reading the next subsection and you will find advice on porting your script back to `gawk-xml`.

2.3.2 Converting a script from portable subset into `gawk-xml`

Conversion of scripts from the portable subset to full `gawk-xml` is even easier. This ease derives from the similarity of the portable subset's event-loop with the API in *Concise Interface - Reduced set of variables shared by all events* as described in the Section 8.2 [`gawk-xml` Core Language Interface Summary], page 88. The main point in porting is replacing the invocation of `getXMLEVENT` with `getline`. Step through the following task list and you will soon arrive at an application that supports all subtleties of the XML data.

1. Copy the application source code file into a new source code file.
2. In the new source code file, insert `@load "xml"` at the top of the file.
3. In the `BEGIN` pattern, convert the condition in the `while` statement of the event-loop.

   ```
   while (getXMLEVENT(ARGV[1])) {
   ```

   gets transformed into

   ```
   while (getline > 0) {
   ```

4. Leave the rest of the `BEGIN` pattern with its event-loop unchanged.
5. Remove the functions `getXMLEVENT`, `unescapeXML`, and `closeXMLEVENT`.
6. Take the resulting application source file and try if it really works in the expected way. Compare the resulting output.

3 XML Core Language Extensions of gawk

In Section 1.2 [How to traverse the tree with gawk], page 8, we have concentrated on the tree structure of the XML file in Figure 1.3. We found the two patterns `XMLSTARTELEM` and `XMLENDELEM` which help us following the process of tree traversal. In this chapter we will find out what the other XML-specific patterns are. All of them will be used in example scripts and their meaning will be described informally.

3.1 Checking for well-formedness

One of the advantages of using the XML format for storing data is that there are formalized methods of checking correctness of the data. Whether the data is written by hand or it is generated automatically, it is always advantageous to have tools for finding out if the new data obeys certain rules (is a tag misspelt ? another one missing ? a third one in the wrong place ?).

These mechanisms for checking correctness are applied at different levels. The lowest level being *well-formedness*. The next higher levels of correctness-check are the level of the DTD (see Section 7.6 [Generating a DTD from a sample file], page 70) and (even higher, but not required yet by standards) the Schema. If you have a DTD (or Schema) specification for your XML file, you can hand it over to a *validation* tool, which applies the specification, checks for conformance and tells you the result. A simple tool for validation against a DTD is `xmllint` (`http://xmlsoft.org/xmllint.html`), which is part of `libxml` and therefore installed on most GNU/Linux systems. Validation against a Schema can be done with more recent versions of `xmllint` or with the `xsv` (`http://www.ltg.ed.ac.uk/~ht/xsv-status.html`) tool.

There are two reasons why validation is currently not incorporated into the `gawk` interpreter.

1. Validation is not trivial and only DTD-validation has reached a proper level of standardization, support and stability.
2. We want a tool that can process all well-formed XML files, not just a tool for processing clean data. A good tool is one that you can rely on and use for fixing problems. What would you think of a car that rejected to drive outside just because there is some mud on the street and the sun isn't shining ?

Here is a script for testing well-formedness of XML data. The real work of checking well-formedness is done by the XML parser incorporated into `gawk`. We are only interested in the result and some details for error diagnostic and recovery.

```
@load "xml"
END {
  if (XMLERROR)
    printf("XMLERROR '%s' at row %d col %d len %d\n",
            XMLERROR, XMLROW, XMLCOL, XMLLEN)
  else
    print "file is well-formed"
}
```

As usual, the script starts with switching `gawk` into XML mode. We are not interested in the content of the nodes being traversed, therefore we have no action to be triggered for

a node. Only at the end (when the XML file is already closed) we look at some variables reporting success or failure. If the variable XMLERROR ever contains anything other than 0 or the empty string, there is an error in parsing and the parser will stop tree traversal at the place where the error is. An explanatory message is contained in XMLERROR (whose contents depends on the specific parser used on this platform). The other variables in the example contain the line number and the column in which the XML file is formed badly.

3.2 Printing an outline of an XML file

When working with XML files, it is sometimes necessary to gain some oversight over the structure an XML file. Ordinary editors confront us with a view such as in Figure 1.2 and not a pretty tree view such as in Figure 1.3. Software developers are used to reading text files with proper indentation like the one in Figure 3.1.

```
book lang='en' id='hello-world'
  bookinfo
    title
  chapter id='introduction'
    title
    para
    sect1 id='about-this-book'
      title
      para
    sect1 id='work-in-progress'
      title
      para
```

Figure 3.1: XML data (DocBook file) as a tree with proper indentation

Here, it is a bit harder to recognize hierarchical dependencies among the nodes. But proper indentation allows you to oversee files with more than 100 elements (a purely graphical view of such large files gets unbearable). Figure 3.1 was inspired by the tool `outline` that comes with the Expat (`http://expat.sourceforge.net`) XML parser. The `outline` tool produces such an indented output and we will now write a script that imitates this kind of output.

```
@load "xml"
XMLSTARTELEM {
  printf("%*s%s", 2*XMLDEPTH-2, "", XMLSTARTELEM)
  for (i=1; i<=NF; i++)
    printf(" %s='%s'", $i, XMLATTR[$i])
  print ""
}
```

Figure 3.2: `outline.awk` produces a tree-like outline of XML data

The script `outline.awk` in Figure 3.2 looks very similar to the other scripts we wrote earlier, especially the script `max_depth.awk`, which also traversed nodes and remembered the depth of the tree while traversing. The most important differences are in the lines with the `print` statements. For the first time, we don't just check if the XMLSTARTELEM variable

contains a tag name, but we also print the name out, properly indented with a `printf` format statement (two blank characters for each indentation level).

At the end of the description of the `max_depth.awk` script in Figure 1.4 we already mentioned the variable `XMLDEPTH`, which is used here as a replacement of the `depth` variable. As a consequence, bookkeeping with the `depth` variable in an action after the `XMLENDELEM` is not necessary anymore. Our script has become shorter and easier to read.

The other new phenomenon in this script is the associative array `XMLATTR`. Whenever we enter a markup block (and `XMLSTARTELEM` is non-empty), the array `XMLATTR` contains all the attributes of the tag. You can find out the value of an attribute by accessing the array with the attribute's name as an array index. In a well-formed XML file, all the attribute names of one tag are distinct, so we can be sure that each attribute has its own place in the array. The only thing that's left to do is to iterate over all the entries in the array and print name and value in a formatted way. Earlier versions of this script really iterated over the associative array with the `for (i in XMLATTR)` loop. Doing so is still an option, but in this case we wanted to make sure that attributes are printed in exactly the same oder that is given in the original XML data. The exact order of attribute names is reproduced in the fields `$1 .. $NF`. So the `for` loop can iterate over the attributes *names* in the fields `$1 .. $NF` and print the attribute *values* `XMLATTR[$i]`.

Please note that, staring with `gawk` 4.2 which supports version 2 of the API, the `XMLATTR` values are considered to be user input and are eligible for the strnum attribute. So if the values appear to be numeric, `gawk` will treat them as numbers in comparisons. This feature was not available prior to version 2 of the `gawk` API.

3.3 Pulling data out of an XML file

The script we are analyzing in this section produces exactly the same output as the script in the previous section. So, what's so different about it that we need a second one ? It is the programming style which is employed in solving the problem at hand. The previous script was written so that the pattern `XMLSTARTELEM` is positioned within the *pattern*. This is ordinary AWK programming style, but it is not the way users of other programming languages were brought up with. In a procedural language, the software developer expects that he himself determines control flow within a program. He writes down what has to be done first, second, third and so on. In the *pattern-action* model of AWK, the novice software developer often has the oppressive feeling that

- he is not *in control*
- events seem to crackle down on him from nowhere
- data flow seems chaotic and invariants don't exist
- assertions seem impossible

This feeling is characteristic for a whole class of programming environments. Most people would never think of the following programming environments to have something in common, but they have. It is the absence of a static control flow which unites these environments under one roof:

- In GUI frameworks like the X Window system, the main program is a trivial *event loop* – the main program does nothing but wait for events and invoke event-handlers.

- In the Prolog programming language, the main program has the form of a *query* – and then the Prolog interpreter decides which rules to apply to solve the query.
- When writing a compiler with the `lex` and `yacc` tools, the main program only invokes a function `yyparse()` and the exact control flow depends on the input source which controls invocation of certain rules.
- When writing an XML parser with the Expat (`http://expat.sourceforge.net`) XML parser, the main program registers some callback handler functions, passes the XML source to the Expat parser and the detailed invocation of callback function depends on the XML source.
- Finally, AWK's *pattern-action* encourages writing scripts that have no main program at all.

Within the context of XML, a terminology has been invented which distinguishes the procedural *pull* style from the event-guided *push* style. The script in the previous section was an example of a *push*-style script. Recognizing that most developers don't like their program's control flow to be pushed around, we will now present a script which pulls one item after the other from the XML file and decides what to do next in a more obvious way.

```
@load "xml"
BEGIN {
  while (getline > 0) {
    switch (XMLEVENT) {
      case "STARTELEM": {
        printf("%*s%s", 2*XMLDEPTH-2, "", XMLSTARTELEM)
        for (i=1; i<=NF; i++)
          printf(" %s='%s'", $i, XMLATTR[$i])
        print ""
      }
    }
  }
}
```

One XML event after the other is pulled out of the data with the `getline` command. It's like feeling each grain of sand pour through your fingers. Users who prefer this style of reading input will also appreciate another novelty: The variable `XMLEVENT`. While the *push-style* script in Figure 3.2 used the event-specific variable `XMLSTARTELEM` to detect the occurrence of a new XML element, our *pull-style* script always looks at the value of the same universal variable `XMLEVENT` to detect a new XML element. We will dwell on a more detailed example in Figure 7.14.

Formally, we have a script that consists of one `BEGIN` pattern followed by an action which is always invoked. You see, this is a corner case of the *pattern-action* model which has been reduced so wide that its essence has disappeared. Instead of the patterns you now see the cases of `switch` statement, embedded into a `while` loop (for reading the file item-wise). Obviously, we have explicite conditionals now, instead of the implicite ones we used formerly. The actions invoked within the `case` conditions are the same we have seen in the *push* approach.

3.4 Character data and encoding of character sets

All of the example scripts we have seen so far have one thing in common: they were only interested in the tree structure of the XML data. None of them treated the words between the tags. When working with files like the one in Figure 1.2, you are sometimes more interested in the words that are embedded in the nodes of Figure 1.3. XML terminology calls these words *character data*. In the case of a DocBook file one could call these words which are interspersed between the tags the *payload* of the whole document. Sometimes one is intersted in freeing this payload from all the useless stuff in angle brackets and extract the character data from the file. The structure of the document may be lost, but the bare textual content in ASCII is revealed and ready for importing it into an application software which does not understand XML.

```
Hello, world



Introduction

This is the introduction. It has two sections


About this book

This is my first DocBook file.




Warning

This is still under construction.
```

Figure 3.3: Example of some textual data from a DocBook file

You may wonder where the blank lines between the text lines come from. They are part of the XML file; each line break in the XML outside the tags (even the one after the closing angle bracket of a tag) is character data. The script which produces such an output is extremely simple.

```
@load "xml"
XMLCHARDATA  { printf $0 }
```

Figure 3.4: `extract_characters.awk` extracts textual data from an XML file

Each time some character data is parsed, the `XMLCHARDATA` pattern is set to 1 and the character data itself is stored into the variable `$0`. A bit unusual is the fact that the text itself is stored into `$0` and not in `XMLCHARDATA`. When working with text, one often needs the text split into fields like AWK does it when the interpreter is not in XML mode. With the words stored in fields `$1 ... $NF`, we now have found a way to refer to isolated words again; it would be easy to extend the script above so that it counts words like the script `wc.awk` did.

Most texts are not as simple as Figure 3.3. Textual data in computers is not limited to 26 characters and some punctuation marks anymore. On all keyboards we have various kinds

of brackets (`<`, [and {) and in Europe we have had things like the ligature (Æ) or the umlaut (ü) for centuries. Having thousands of symbols is not a problem in itself, but it became a problem when software applications started representing these symbols with different bytes (or even byte sequences). Today we have a standard for representing all the symbols in the world with a byte sequence – Unicode (`http://www.unicode.org/versions/Unicode4.0.0`). Unfortunately, the accepted standard came too late. Earlier standardization efforts had created ways of representing subsets of the complete symbol set, each subset containing 256 symbols which could be represented by one byte. These subsets had names which are still in use today (like ISO-8859-1 or IBM-852 or ISO-2022-JP). Then came the programming language Java with a `char` data type having 16 bits for each character. It turned out that 16 bits were also not enough to represent all symbols. Having recognized the fixed 16 bit characters as a failure, the standards organizations finally established the current Unicode standard. Today's Unicode character set is a wonderful catalog of symbols – the book mentioned above needs more than a 1000 pages to list them all.

And now to the ugly side of Unicode:

- The names of the 8 bit character sets are still in use and have to be supported by XML parsers and the software built upon them.
- Symbols in the Unicode catalog have an unambiguous number, but their number may be encoded in many different ways with varying numbers of bytes per character.
- When displaying a text, you have to decide which encoding you want to use; if the text is encoded differently, you will see strange symbols that you have never dreamed of.

Notice that the *character set* and the *character encoding* are very different notions. The former is a set in the mathematical sense while the latter is a way of mapping the number of the character into a byte sequence of varying length. To make things worse: The use of these terms is not consistent – neither the XML specification nor the literature distinguishes the terms cleanly. For example, take the citation from the excellent O'Reilly book XML in a Nutshell (`http://www.oreilly.com/catalog/xmlnut3/`) in chapter 5.2 (`http://safari.oreilly.com/0596002920/xmlnut2-CHP-5-SECT-2`):

> 5.2 The Encoding Declaration
>
> Every XML document should have an *encoding declaration* as part of its XML declaration. The encoding declaration tells the parser in which character set the document is written. It's used only when other metadata from outside the file is not available. For example, this XML declaration says that the document uses the character encoding US-ASCII:
>
> ```
> <?xml version="1.0" encoding="US-ASCII" standalone="yes"?>
> ```
>
> This one states that the document uses the Latin-1 character set, though it uses the more official name ISO-8859-1:
>
> ```
> <?xml version="1.0" encoding="ISO-8859-1"?>
> ```
>
> Even if metadata is not available, the encoding declaration can be omitted if the document is written in either the UTF-8 or UTF-16 encodings of Unicode. UTF-8 is a strict superset of ASCII, so ASCII files can be legal XML documents without an encoding declaration.

Several times a character set name is assigned to an encoding declaration – the book does it and the XML samples do it too. Only in the last paragraph the usage of terms is clean: UTF-8 is the default way of encoding a character into a byte sequence.

After this unpleasant excursion into the cultural history of textual data in occidental societies, let's get back to gawk and see how the concepts of the encoding and the character set are incorporated into the language. Three variables are all that you need to know, but each of them comes from a different context. Take care that you recognize the difference between the XML document, gawk's internal data handling and the influence of an environment variable from the shell environment setting the *locale*.

- XMLATTR["ENCODING"] is a pattern variable that (when non-empty and XMLDECLARATION is triggered) contains the name of the character encoding in which the XML file was originally encoded. This information comes from the first line of the XML file (if the line contains the usual XML header). There is no use in overwriting this variable, the variable is meant to tell you what's in the XML data and nothing happens when you change XMLATTR["ENCODING"].
- XMLCHARSET is the variable to change if you want to see the XML data converted to a character set of your own choice. When you set this variable, the gawk interpreter will remember the character set of your choice. But this choice will take effect only upon opening the next file. A change of XMLCHARSET will not influence XML data from a file that has already been opened earlier for reading.
- LANG is an environment variable of your operating system. It tells the gawk interpreter which value to use for XMLCHARSET on initial startup when nothing has been said about it in the user's script. In the absence of any setting for LANG, US-ASCII is used as the default encoding. Up to now, we have always talked about the encoding and character set of the data to be processed. Remember that the source code of your program is also written in some character set. It is usually the LANG character set that is used while writing programs. Imagine what happens when you have a program containing a character from your native character set, for which there is no encoding in the character set used at run-time. The alert reader will notice how consequent gawk is in following the Unicode tradition of mixing up character encoding and character set.

After so much scholastic reasoning, you might be inclined to presume that character sets and encodings are hardly of any use in real life (except for befuddling the novice). The following example should dispel your doubts. In real life, circumstance transcending sensible reasoning could require you to import the text in Figure 3.3 into a Microsoft Windows application. Contemporary flavours of Microsoft Windows prefer to store textual data in UTF-16. So, a script for converting the text to UTF-16 would be a nice tool to have – and you already have such a tool. The script extract_characters.awk in Figure 3.4 will do the job, if you tell the gawk interpreter to use the UTF-16 encoding when reading the DocBook file. Two alternatives ways of reaching this target arise:

- Change the script and insert a line setting XMLCHARSET to UTF-16. After invocation, the gawk interpreter will now print the same data as in Figure 3.3, but converted to UTF-16.

  ```
  BEGIN { XMLCHARSET="utf-16" }
  ```
- Do not change the script, but before invoking the gawk interpreter, set the environment variable LANG to UTF-16.

The result will be the same in both cases, provided your operating system supports these character sets and encodings. In real life, it is probably a better idea to avoid the second of

these approaches because it requires changes (and possibly side-effects) at the level of the command line shell.

3.5 Dealing with DTDs

Earlier in this chapter we have seen that gawk does not validate XML data against a DTD. The declaration of a document type in the header of an XML file is an optional part of the data, not a mandatory one. If such a declaration is present (like it is in Figure 1.2), the reference to the DTD will not be resolved and its contents will not be parsed. However, the presence of the declaration will be reported by gawk. When the declaration starts, the variable XMLSTARTDOCT contains the name of the root element's tag; and later, when the declaration ends, the variable XMLENDDOCT is set to 1. In between, the array variable XMLATTR will be populated with the values of the public identifier of the DTD (if any) and the value of the system's identifier of the DTD (if any). Other parts of the declaration (elements, attributes and entities) will not be reported.

```
@load "xml"
XMLDECLARATION {
  version    = XMLATTR["VERSION"        ]
  encoding   = XMLATTR["ENCODING"       ]
  standalone = XMLATTR["STANDALONE"     ]
}
XMLSTARTDOCT {
  root       = XMLSTARTDOCT
  pub_id     = XMLATTR["PUBLIC"         ]
  sys_id     = XMLATTR["SYSTEM"         ]
  intsubset  = XMLATTR["INTERNAL_SUBSET"]
}
XMLENDDOCT {
  print FILENAME
  print "  version    '" version    "'"
  print "  encoding   '" encoding   "'"
  print "  standalone '" standalone "'"
  print "  root   id '" root   "'"
  print "  public id '" pub_id "'"
  print "  system id '" sys_id "'"
  print "  intsubset '" intsubset "'"
  print ""
  version = encoding = standalone = ""
  root = pub_id = sys_id = intsubset ""
}
```

Figure 3.5: db_version.awk extracts details about the DTD from an XML file

Most users can safely ignore these variables if they are only interested in the data itself. But some users may take advantage of these variables for checking requirements of the XML data. If your data base consists of thousands of XML file of diverse origins, the public identifier of their DTDs will help you gain an oversight over the kind of data you have to handle and over potential version conflicts. The script in Figure 3.5 will assist you in

analyzing your data files. It searches for the variables mentioned above and evaluates their content. At the start of the DTD, the tag name of the root element is stored; the identifiers are also stored and finally, those values are printed along with the name of the file which was analyzed. After each DTD, the remembered values are set to an empty string until the DTD of the next file arrives.

In Figure 3.6 you can see an example output of the script in Figure 3.5. The first entry is the file we already know from Figure 1.2. Obviously, the first entry is a DocBook file (English version 4.2) containing a `book` element which has to be validated against a local copy of the DTD at CERN in Switzerland. The second file is a `chapter` element of DocBook (English version 4.1.2) to be validated against a DTD on the Internet. Finally, the third entry is a file describing a project of the GanttProject application. There is only a tag name for the root element specified, a DTD does not seem to exist.

```
data/dbfile.xml
  version    ''
  encoding   ''
  standalone ''
  root   id  'book'
  public id  '-//OASIS//DTD DocBook XML V4.2//EN'
  system id  '/afs/cern.ch/sw/XML/XMLBIN/share/www.oasis-open.org/docbook/xmldtd-4.2/docbookx.dtd'
  intsubset  ''

data/docbook_chapter.xml
  version    ''
  encoding   ''
  standalone ''
  root   id  'chapter'
  public id  '-//OASIS//DTD DocBook XML V4.1.2//EN'
  system id  'http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd'
  intsubset  ''

data/exampleGantt.gan
  version    '1.0'
  encoding   'UTF-8'
  standalone ''
  root   id  'ganttproject.sourceforge.net'
  public id  ''
  system id  ''
  intsubset  ''
```

Figure 3.6: Details about the DTDs in some XML files

You may wish to make changes to this script if you need it in daily work. For example, the script currently reports nothing for files which have no DTD declaration in them. You can easily change this by appending an action for the `END` rule which reports in case all the variables `root`, `pub_id` and `sys_id` are empty. As it is, the script parses the entire XML file, although the DTD is always positioned at the top, before the root element. Parsing the root element is unnecessary and you can improve the speed of the script significantly if you tell it to stop parsing when the first element (the root element) comes in.

```
XMLSTARTELEM { nextfile }
```

3.6 Sorting out all kinds of data from an XML file

If you have read this book sequentially until now, you have understood how to read an XML file and treat it as a tree. You also know how to handle different character encodings and DTD declarations. This section is meant to give you an overview of what other patterns there are when you work with XML files. The overview is meant to be complete in the sense that you will see the name of every pattern involved and an example of usage. Conceptually, you will not see much new material, this is only about some new variables for passing information from the XML file. Here are the new patterns:

- XMLPROCINST contains the name of a processing instruction, while $0 contains its contents.

- XMLCOMMENT indicates an XML comment. The comment itself is in $0.

- XMLDECLARATION indicates that the XML version number from the first line of the XML file can be read from `XMLATTR["VERSION"]`.

- XMLUNPARSED indicates a text that did not fit into any other category. Its contents is in $0.

The following script is meant to demonstrate all XML patterns and variables. It can help you while you are debugging other scripts because this script will show you everything that is in the XML file and how it is read by `gawk`.

```
@load "xml"
# Set XMLMODE so that the XML parser reads strictly
# compliant XML data.  Convert characters to Latin-1.
BEGIN         { XMLMODE=1 ; XMLCHARSET = "ISO-8859-1" }
# Print an outline of nested tags and attributes.
XMLSTARTELEM  {
  printf("%*s%s", 2*XMLDEPTH-2, "", XMLSTARTELEM)
  for (i=1; i<=NF; i++)
    printf(" %s='%s'", $i, XMLATTR[$i])
  print ""
}
# Upon closing tag, XMLPATH still holds the tag name.
XMLENDELEM    { printf("%s %s\n", "XMLENDELEM", XMLPATH) }
# XMLEVENT holds the name of the current event.
XMLEVENT { print "XMLEVENT", XMLEVENT, XMLNAME, $0    }
# Character data will not be lost.
XMLCHARDATA   { print "XMLCHARDATA", $0               }
# Processing instruction and comments instructions will be reported.
XMLPROCINST   { print "XMLPROCINST", XMLPROCINST, $0  }
XMLCOMMENT    { print "XMLCOMMENT", $0                }
# CDATA sections are used for quoting verbatim text.
XMLSTARTCDATA { print "XMLSTARTCDATA"                 }
# CDATA blocks have an end that is reported.
XMLENDCDATA   { print "XMLENDCDATA"                   }
# The very first event holds the version info.
XMLDECLARATION {
  version    = XMLATTR["VERSION"   ]
  encoding   = XMLATTR["ENCODING"  ]
  standalone = XMLATTR["STANDALONE"]
}
# DTDs, if present, are indicated as such.
XMLSTARTDOCT {
  root = XMLSTARTDOCT
  print "XMLATTR[PUBLIC]", XMLATTR["PUBLIC"]
  print "XMLATTR[SYSTEM]", XMLATTR["SYSTEM"]
  print "XMLATTR[INTERNAL_SUBSET]", XMLATTR["INTERNAL_SUBSET"]
}
# The end of a DTD is also indicated.
XMLENDDOCT     { print "root", root                   }
# Unparsed text occurs rarely.
XMLUNPARSED    { print "XMLUNPARSED", $0              }
# XMLENDDOCUMENT occurs only with XML data that is not
# strictly compliant to standards (multiple root elements).
XMLENDDOCUMENT { print "XMLENDDOCUMENT"               }
# At the end of the file, you can check if an error occurred.
END { if  (XMLERROR)
        printf("XMLERROR '%s' at row %d col %d len %d\n",
               XMLERROR, XMLROW, XMLCOL, XMLLEN)
}
```

Figure 3.7: The script `demo_pusher.awk` demonstrates all variables of `gawk-xml`

4 Some Convenience with the xmllib library

All the variables that were added to the AWK language to allow for reading XML files show you *one* event at a time. If you want to rearrange data from several nodes, you have to collect the data during tree traversal. One example for this situation is the name of the parent node which is needed several time in the examples of Chapter 7 [Some Advanced Applications], page 51.

Stefan Tramm has written the `xmllib` library because he wanted to simplify the use of `gawk` for command line usage (one-liners). His library comes as an ordinary script file with AWK code and is automatically included upon invocation of `xmlgawk`. It introduces new variables for easy handling of character data and tag nesting. Stefan contributed the library as well as the `xmlgawk` wrapper script.

4.1 Introduction Examples

The most used AWK script is something like this:

```
$ awk '/matchrx/ { print $3, $1 } foo.dat
```

which assumes a line at a time approach and the division of a line (record) into words (fields), where only some fields are printed for records that match. With xmlgawk this does not change drastically, the approach is now one XML token at a time:

```
$ xmlgawk '/on-loan/ { grep() }' books.xml
```

which prints the complete XML subtree, where "on-loan" matches either characterdata, some part of a start- or endelement or some part of an attributname or -value. The function grep() provided in the xmllib.awk does all the magic for you. If you need a simple prettyprinter for an XML stream (because there are perhaps no new lines in the file), then you can use this:

```
$ xmlgawk 'SE { grep(4) }' books.xml
```

The number "4" gives the indention. The variable SE is set on every startelement, including the root element. This is an ideal command line idiom. Faster (in CPU time) xmlgawk solutions are possible, but whats the difference between 100msec or 1 second for a quick check? The second most anticipated usage is searching through parts of XML documents and printing the results in a nicer human readable form:

```
$ xmlgawk '
    EE == "title"  { t = CDATA }
    EE == "author" { w = CDATA }
    EE == "book" && ATTR[PATH"@publisher"] == "WROX" { print "author:", w, "title:", t
  ' books.xml
```

This script memorizes every <author> and <title> and prints them only, when a <book> has the attribute "publisher" with the value "WROX". The variable EE is set with the name of an endelement, The variable PATH contains all 'open' startelements before the current one in the document. The array ATTR contains all XML Attributes of every startelement in PATH. Here is a little example to make it clearer:

```
$ xmlgawk '
    SE { print "SE", SE
         print "   PATH", PATH
```

```
              print "   CDATA", CDATA
              XmlTraceAttr(PATH)
            }
         EE { print "EE", EE
              print "   PATH", PATH
              print "   CDATA", CDATA
            }
       ' books.xml
     SE books
        PATH /books
        CDATA
     SE book
        PATH /books/book
        CDATA
     ATTR[/books/book@on-loan]="Sanjay"
     ATTR[/books/book@publisher]="IDG books"
     SE title
        PATH /books/book/title
        CDATA
     EE title
        PATH /books/book/title
        CDATA XML Bible
     SE author
        PATH /books/book/author
        CDATA
     EE author
        PATH /books/book/author
        CDATA Elliotte Rusty Harold
     EE book
        PATH /books/book
        CDATA
```

The variable CDATA contains the character data right before the start- or endelement, which is very convenient in the above examples and in daily life.

4.2 Main features

The main ideas are:

- make character data available preceding start- and endelements
- provide the current path (parse stack, nesting level)
- make all startelement attributes of the complete path available
- provide helper functions for output
- provide help for grep-like tools
- provide debug support

The following sections are devoted to the above topics.

Character Data (CDATA)

The variable CDATA collects the characters of all XMLCHARDATA events. At an XMLSTARTELEM or XMLENDELEM event the CDATA variable is trimmed (by calling the function trim()), that means leading and trailing whitespace ([:space:]) characters are removed.

Please, keep in mind to use the idiom 'print quoteamp(CDATA)' in your code, where the output is again XML or (X)HTML.

Start- and End-elements (SE, EE, PATH, ATTR[])

The variable SE has the same content and behaviour as XMLSTARTELEM, but it is much faster to type (EE does the same for XMLENDELEM).

The variable PATH contains all currently 'open' startelements. It is like a parse stack and allows checks for the context of a current element. Elements are delimited by slashes `"/"`. If PATH ist not empty, it begins with a `"/"`.

The ATTR array stores every attribute of 'open' startelements. This is sometimes very convenient, because you can simply 'look back' for already seen attributes. Attributenames are separated by an at-sign `"@"` from its element path, eg:

```
/books/book@publisher
```

The helper function XmlTraceAttr prints all attributes for the specified path (if no path argument is given, the function defaults to PATH).

Comments (CM)

CM contains the trim-ed comment string in XMLCOMMENT, and $0 holds the completely reconstructed comment.

All comments in a character data section will be seen by the user program before the accumulated CDATA variable delivers the characters.

Processing Instructions (PI)

All processing instruction are available via PI (which has the same content as XMLPROCINST). $0 contains the completely reconstructed processing instruction.

The very first procinst is specially handled by expat and den XML core extension. xmllib.awk takes care of this and delivers the very first procinst as a normal procinst via PI.

Real Character Data (XmlCDATA)

In the very seldom case you have to process real character data section, the variable XmlCDATA delivers the untrimmed characters between a XMLSTARTCDATA and a XMLENDCDATA token. These characters are also appended to CDATA, so you will get every character within CDATA at the next start or end element.

grep function

The grep function is build, to print a complete subtree, starting at a startelement (XML-STARTELEM) token. Therefore grep cannot print comments before and behind the root element.

If grep is given a numerical argument, grep prettyprints the XML subtree and uses the value as the number of spaces for indention. If no argument is given, the subtree is printed as in the source document.

XmlStartElement and XmlEndElement functions

The helper functions return nice formatted strings for the tail of PATH. These functions are used in the grep function, but can also be used by end user programs.

XmlPathTail function

Delivers the current element name from PATH. It needs two parameters, the path and the delimiter character. If no path is supplied PATH will be used, if no delimiter is supplied "/" will be used.

XmlTraceAttr function

When debugging a xmlgawk script it is sometimes very welcome to have a simple functions, which prints all attributes. This is exactly what XmlTraceAttr does. The optional parameter is the path of the startelement for which the attributes should be printed (the default is PATH).

Simple String manipulation functions

xmllib.awk provides three additional little but useful functions:

```
# remove leading and trailing [[:space:]] characters
function trim(str)
{
    sub(/^[[:space:]]+/, "", str)
    if (str) sub(/[[:space:]]+$/, "", str)
    return str
}

# quote function for character data escape & and <
function quoteamp(str)
{
    gsub(/&/, "\\&amp;", str)
    gsub(/</, "\\&lt;", str)
    return str
}

# quote function for attribute values
#  escape every character, which can
#  cause problems in attribute value
```

```
#  strings; we have no information,
#  whether attribute values were
#  enclosed in single or double quotes
function quotequote(str)
{
    gsub(/&/, "\\&amp;", str)
    gsub(/</, "\\&lt;", str)
    gsub(/"/, "\\&quot;", str)
    gsub(/'/, "\\&apos;", str)
    return str
}
```

Minor Issues

The grep() and XmlStartelement() functions do NOT return the exact same string as seen in the input, the strings are semantically identical but completely reconstructed. xmlgawk gives you an 80% solution fast, if you want more, use another tool (and more time).

xmllib.awk passes every token from the gawk-xml core-extension through to the user program. This means, that you can use NR and FNR in your code (especially in rules FNR==1), but remember the count XML tokens now.

All variable and function names beginning with the prefix 'XML' are reserved for the GAWK XML core and prefix 'Xml' for xmllib.awk. If you want to prefix a name with 'xml' in your programs use all lower case.

For convenience purposes some names in the xmllib.awk have shorter names (variables all uppercase, functions all lowercase):

- SE (identical setting as XMLSTARTELEMENT)
- EE (identical setting as XMLSTARTELEMENT)
- CM (identical setting as XMLCOMMENT)
- PI (identical setting as XMLPROCINST)
- CDATA (collected and trimed XMLCHARDATA)
- ATTR (complete attribute stack)
- PATH (complete element path)
- grep()
- quoteamp()
- quotequote()

4.3 Usage of xmllib.awk

The following sections give more elaborate examples for the xmlgawk programming. At first we concentrate on search tools, then we focus on converters and template instantiations. The last sections gives an example how classical configuration files can be replaced by XML files, which opens the brave new XML world to old shell script(er)s – new tricks to an old dog.

Most examples use the following `books.xml` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<books>
  <book publisher="IDG books" on-loan="Sanjay">
    <title>XML Bible</title>
    <author>Elliotte Rusty Harold</author>
  </book>
  <book publisher="Addison-Wesley">
    <title>The Mythical Man Month</title>
    <author>Frederick Brooks</author>
  </book>
  <book publisher="WROX">
    <title>Professional XSLT 2nd Edition</title>
    <author>Michael Kay</author>
  </book>
  <book publisher="Prentice Hall" on-loan="Sander" >
   <title>Definitive XML Schema</title>
   <author>Priscilla Walmsley</author>
  </book>
  <book publisher="APress">
   <title>A Programmer's Introduction to C#</title>
   <author>Eric Gunnerson</author>
  </book>
</books>
```

Ad hoc Queries (grep-like tools)

At first some one-liners:

```
# print all books from the publisher WROX
$ xmlgawk 'XMLATTR["publisher"]=="WROX" {grep(4)}' books.xml

# print complete information for every loaned book
$ xmlgawk 'XMLATTR["on-loan"] {grep(2)}' books.xml

# print loaner name and loaned book title only
$ xmlgawk 'EE=="title" && l=ATTR["/books/book@on-loan"] { \
                print l, "loaned", CDATA }' books.xml

# print all book titles containing the word "Professional"
#   to print "&" in titles as "&amp;", use quoteamp()
$ xmlgawk 'EE=="title" && CDATA~/Professional/ { print PATH ":", quoteamp(CDATA) }' bo
```

Formatter and Converter (sed-like tools)

The complexity of formatter or converter tools depends on the output format. The simpler the better – comma-separated-value-files aren't dead and wont be dead in 20 years...

If the output format will be XML, we speak of a formatter and if it will be something different, we speak of a converter. Converters can generate CSV-, SQL-, or proprietary format files out of XML input.

Formatters are like prettyprinters or extended grep-like tools. The main question you have to answer is whether you need a nice human readable indented formatting or just one line of characters, or something in between.

In both cases you have to take care of the character set encoding you want to generate: ASCII, ISO-8859, UTF-8, ...

Here will follow the extensive Jabber XML-Configfile manipulation script (in productive use at the employer of one author).

Comparison to XSLT

At the moment, template instantiation mechanisms like XSLT are en vogue. We will give a short example why this is so, and what we can do with shell and xmlgawk.

The examples are taken from the very good pages of Anders Moeller (take a look at `http://www.brics.dk/~amoeller/XML/`).

Here you see the proposed XSLT script:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns="http://www.w3.org/1999/xhtml">;
 <xsl:template match="nutrition">
   <html xmlns="http://www.w3.org/1999/xhtml">;
     <head>
       <link href="../style.css"
             rel="stylesheet"
             type="text/css"/>
     </head>
     <body>
       <table border="1">
         <tr>
           <th>Dish</th>
           <th>Calories</th>
           <th>Fat</th>
         <th>Carbohydrates</th>
         <th>Protein</th>
        </tr>
        <xsl:apply-templates
          select="dish"/>
     </table>
    </body>
   </html>
 </xsl:template>
 <xsl:template match="dish">
   <tr>
     <td><xsl:value-of
         select="@name"/></td>
     <td><xsl:value-of
         select="@calories"/></td>
```

```
      <td><xsl:value-of
          select="@fat"/>%</td>
      <td><xsl:value-of
          select="@carbohydrates"/>%</td>
      <td><xsl:value-of
          select="@protein"/>%</td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

A straightforward translation into gawk looks like this:

```
xmlgawk '
BEGIN              { print "<html xmlns=\"http://www.w3.org/1999/xhtml\"><head>;"
                     print "<link href=\"../style.css\ rel=\"stylesheet\" type=\"text/c
                     print "</head><body><table border=\"1\">"
                     print "<tr><th>Dish</th><th>Calories</th>"
                     print "<th>Fat</th><th>Carbohydrates</th>"
                     print "<th>Protein</th></tr>"
                   }
EE == "title"      { print "<tr><td>" CDATA "</td>" }
SE == "nutrition" { print "<td>" XMLATTR["calories"] "</td>"
                     print "<td>" XMLATTR["fat"] "%</td>"
                     print "<td>" XMLATTR["carbohydrates"] "%</td>"
                     print "<td>" XMLATTR["protein"] "%</td></tr>"
                   }
END                { print "</table></body></html>" }
' recipes.xml
```

As you can see, the script is filled with print statements and full of \-escapes in the strings. It it really annoying and error-prone to write the print strings and the escapes. This is – in the eyes of the authors – the main reason, that XSLT is used. You take the original HTML, XHTML or XML and insert afterwards the logic. In plain AWK (or Perl or Tcl) it is the other way round – write the logic and insert the template (with print and escapes).

This is the place, where the good old Unix shell with HERE-documents can help out. Take a look at the following solution:

```
#!/bin/bash
cat <<EOT
<html
  xmlns="http://www.w3.org/1999/xhtml">;
  <head>
    <link href="../style.css"
          rel="stylesheet"
          type="text/css"/>
  </head>
  <body>
    <table border="1">
```

```
      <tr>
        <th>Dish</th>
        <th>Calories</th>
        <th>Fat</th>
        <th>Carbohydrates</th>
        <th>Protein</th>
     </tr>
$(xmlgawk '
EE == "title"     { print "     <tr>"
                    print "       <td>" CDATA "</td>"
                  }
SE == "nutrition" { print "       <td>" XMLATTR["calories"]       "</td>"
                    print "       <td>" XMLATTR["fat"]           "%</td>"
                    print "       <td>" XMLATTR["carbohydrates"] "%</td>"
                    print "       <td>" XMLATTR["protein"]       "%</td>"
                    print "     </tr>"
                  }
' recipes.xml)
  </table>
  </body>
</html>
EOT
```

You still have to write some print statements, which seems a reasonable compromise between both worlds (template- vs. logic driven).

5 DOM-like access with the xmltree library

Even with the `xmllib`, random access to nodes in the tree is not possible. There are a few applications which need access to parent and child elements and sometimes even remote places in the tree. That's why Manuel Collado wrote the `xmltree` library.

Manuel's `xmltree` reads an XML file at once and stores it entirely. This approach is called the DOM approach. Languages like XSL inherently assume that the DOM is present when executing a script. This is, at once, the strength (random access) and the weakness (holding the entire file in memory) of these languages.

Manuel contributed the `xmltree` library.

FIXME: This chapter has not been written yet.

6 Problems from the newsgroups comp.text.xml and comp.lang.awk

This chapter is a collection of XML related problems which were posted in newsgroups on the Internet. After a citation of the original posting and a short outline of the problem, each of these problems is followed by a solution in `gawk-xml`. Although we take care to find an exact solution to the original problem, we are not really interested in the details of any of these problems. What we *are* interested in is a demonstration of how to attack problems of this kind in general. The *raison d'être* for this chapter is manifold:

- The problems being posted in newsgroups often represent daily work much better than academic textbook examples.
- The development of Open Source Software is driven by the needs occurring in real life. Adapting a tool to user's needs is much more rewarding than adapting it to ideological criteria.
- Such problems are a welcome test bed for evaluating the adequacy of a tool. We will learn about the bright sides and also about the not-so-bright sides of `gawk-xml`.
- Each kind of problem-solving tool will only prove its utility when the user has acquired some basic skills and techniques to use it. We will demonstrate some of these in passing.

6.1 Extract the elements where i="Y"

The original poster of this problem wanted to find all tags which have an attribute of a specific kind (`i="Y"`) and produce the value of another attribute as output. He described the problem as follows with an input/output relationship:

```
suppose i have:

<a>
  <b i="Y" j="aaaa"/>
  <c i="N" j="bbbb"/>
  <d i="Y" j="cccc"/>
  <e i="N" j="dddd"/>
  <f i="N" j="eeee"/>
  <g i="Y" j="ffff"/>
</a>

and i want to extract the elements where i="Y" such that i get something like
<x>
  <y>1. aaaa</y>
  <y>2. cccc</y>
  <y>3. gggg</y>
</x>

how would i get the numbering to work across the different elements?
```

He probably had XML data from an input form with two fields. The first field containing the answer to an alternative choice (Y/N) and the second field containing some description. The goal was to extract the specific description for all positive answers (`i="Y"`). All the

output data had to be embedded into nested tags (x and y). The nesting of the tags explains the print commands in the BEGIN and the END patterns of the following solution.

```
@load "xml"
BEGIN { print "<x>" }
XMLSTARTELEM {
  if (XMLATTR["i"] == "Y")
    print "  <y>" ++n ". " XMLATTR["j"] "</y>"
}
END { print "</x>" }
```

An XMLSTARTELEM pattern triggers the printing of the y output tags. But only if the attribute i has the value Y will an output be printed. The output itself consists of the value of the attribute j and is embedded into y tags.

If you try the script above on the input data supplied by the original poster, you will notice that the resulting output differs slightly from the desired output given above. There is obviously a typo in the third item of the output (gggg instead of ffff).

Problems of this kind (input data is XML and output dats is also XML) are usually solved with the XSL language. From this example we learn that gawk-xml is an adequate tool for reading the input data. But producing the tagged structure of the output data (with simple print commands) is not as elegant as some users like it.

6.2 Convert XMLTV file to tabbed ASCII

This problem differs from the previous one in the kind of output data to be produced. Here we produce tabbed ASCII output from an XML input file. The original poster of the question had XML data in the XMLTV format (http://xml.coverpages.org/xmltv.html). XMLTV is a format for storing your knowledge about which TV program (or TV program*me* in British English) will be broadcast at which time on which channel. The original poster gives some example data (certainly not in the most readable form).

```
To help me get my head around XMLGAWK can someone solve the following.
I have a XMLTV data file from which I want to extract certain data and
write to a tab-delimited flat file.

The XMLTV data is as follows:

<?xml version="1.0" encoding="UTF-8"?>
<tv><programme start="20041218204000 +1000" stop="20041218225000
+1000" channel="Network TEN Brisbane"><title>The
Frighteners</title><sub-title/><desc>A psychic private detective, who
consorts with deceased souls, becomes engaged in a mystery as members
of the town community begin dying mysteriously.</desc><rating
system="ABA"><value>M</value></rating><length
units="minutes">130</length><category>Horror</category></programme><programme
start="20041218080000 +1000" stop="20041218083000 +1000"
channel="Network TEN Brisbane"><title>Worst Best
Friends</title><sub-title>Better Than Glen</sub-title><desc>Life's
like that for Roger Thesaurus - two of his best friends are also his
```

```
worst enemies!</desc><rating
system="ABA"><value>C</value></rating><length
units="minutes">30</length><category>Children</category></programme></tv>

The flate file needs to be as follows:

channel<tab>programme
start<tab>length<tab>title<tab>description<tab>rating value

So the first record would read:

Network TEN Brisbane<tab>2004-12-18 hh:mm<tab>130<tab>The
Frighteners<tab>A psychic private detective, who consorts with
deceased souls, becomes engaged in a mystery as members of the town
community begin dying mysteriously.<tab>M
```

So, he wants an ASCII output line for each node of kind `programme`. The proper outline of his example input looks like this:

```
tv
  programme channel='Network TEN Brisbane' start='20041218204000 +1000' stop='20041218
    title
    sub-title
    desc
    rating system='ABA'
      value
    length units='minutes'
    category
  programme channel='Network TEN Brisbane' start='20041218080000 +1000' stop='20041218
    title
    sub-title
    desc
    rating system='ABA'
      value
    length units='minutes'
    category
```

Printing the desired output is not as easy as in the previous section. Here, much data is stored as character data in the nodes and only a few data items are stored as attributes. In `gawk-xml` it is much easier to work with attributes than with character data. This sets `gawk-xml` apart from XSL, which treats both kinds of data in a more uniform way.

In the action after the `BEGIN` pattern we can see how easy it is to produce tabbed ASCII output (i.e. separating output fields with `TAB` characters): just set the `OFS` variable to `"\t"`. Another easy task is to collect the information about the channel and the start time of a program on TV. These are stored in the attributes of each `programme` node. So, upon entering a `programme` node, the attributes are read and their content stored for later work. Why can't we print the output line immediately upon entering the node ? Because the other data bits (length, title and description) follow later in nested nodes. As a consequence, data collection is completed only when we are *leaving* the `programme` node.

Therefore, the printing of tabbed output happens in the action after the `XMLENDELEM == "programme"` pattern.

```
@load "xml"
BEGIN { OFS= "\t" }
XMLSTARTELEM  == "programme" {
  channel = XMLATTR["channel"]
  start   = XMLATTR["start"]
  data    = ""
}
XMLCHARDATA                { data  = $0    }
XMLENDELEM  == "desc"      { desc  = data  }
XMLENDELEM  == "length"    { leng  = data  }
XMLENDELEM  == "title"     { title = data  }
XMLENDELEM  == "value"     { value = data  }
XMLENDELEM  == "programme" {
  print channel, substr(start,1,4) "-" substr(start,5,2) "-" substr(start,7,2) " " \
        substr(start,9,2) ":" substr(start,11,2), leng, title, desc, value
  desc = leng = title = value = ""
}
```

What's left to do is collecting character data. Each time we come across some character data, we store it in a variable `data` for later retrieval. At this moment we don't know yet what kind of character data this is. Only later (when leaving the `desc`, `length`, `title` or `value` node) can we assign the data to its proper destination. This kind of *deferred* assignment of character data is typical for XML parsers following the *streaming* approach: they see only one data item at a time and the user has to take care of storing data bits needed later. XML Transformation languages like XSL don't suffer from this shortcoming. In XSL you have random access to all information in the XML data. It is up to the user to decide if the problem at hand should be solved with a streaming parser (like `gawk-xml`) or with a DOM parser (`http://www.w3.org/TR/REC-DOM-Level-1/`) (like XSL). If you want to use `gawk-xml` and still enjoy the comfort of easy handling of character data, you should use the `xmllib` (see Chapter 4 [Some Convenience with the xmllib library], page 31) or the `xmltree` (see Chapter 5 [DOM-like access with the xmltree library], page 41) library described elsewhere.

6.3 Finding the minimum value of a set of data

Up to now we have seen examples whose main concern was finding and re-formatting of XML input data. But sometimes reading and printing is not enough. The original poster of the following example needs the shortest value of attribute `value` in all `month` tags. He refers explicitly to a solution in XSL which he tried, mentioning some typical problem he had with XSL templates.

```
I'm trying to find the minimum value of a set of data (see below).
I want to compare the lengths of these attribute values and display
the lowest one.

This would be simple if I could re-assign values to a variable,
but from what I gather I can't do that.  How do I keep track of the
```

```
lowest value as I loop through?  My XSL document only finds the length
of each string and prints it out (for now).  I can write a template
that calls itself for recursion, but I don't know how to keep the
minimum value readially available as I go through each loop.

Thanks,

James

XML Document
=============================
<calendar name="americana">
<month value="January"/>
<month value="February"/>
<month value="March"/>
<month value="April"/>
<month value="May"/>
<month value="June"/>
<month value="July"/>
<month value="August"/>
<month value="September"/>
<month value="October"/>
<month value="November"/>
<month value="December"/>
</calendar>
```

The solution he looks for is the value `May`. Simple minds like ours simply go through the list of `month` tags from top to bottom, always remembering the shortest value found up to now. Having finished the list, the remembered value is the solution. Look at the following script and you will find that it follows the path of our simple-minded approach.

```
@load "xml"
XMLSTARTELEM  == "month" {
  # Initialize shortest
  if (shortest == "")
    shortest = XMLATTR["value"]
  # Find shortest value
  if (length(XMLATTR["value"]) < length(shortest))
    shortest = XMLATTR["value"]
}
END { print shortest }
```

A solution in XSL is not as easy as this. XSL is a functional language, as such being mostly free from programming concepts like the *variable*. It is one of the strengths of functional languages that they are mostly free from side-effects and global variables containing values are (conceptually speaking) side-effects. Therefore, a solution in XSL employs the use of so-called *templates* which invoke each other recursively.

Examples like this shed some light on the question why XSL is so different from other languages and therefore harder to learn for most of us. As can be seen from this simple

example, the use of recursion is unavoidable in XSL. Even for the simplest of all tasks. As a matter of fact, thinking recursively is not the way most software developers prefer to work in daily life. Ask them. When did *you* use recursion for the last time in your C or C++ or AWK programs ?

6.4 Updating DTD to agree with its use in doc's

A few months after I wrote Section 7.6 [Generating a DTD from a sample file], page 70, someone posted a request for a similar tool in the newsgroup comp.text.xml (`news://comp.text.xml`).

```
A few years ago my department defined a DTD for a projected class of
documents. Like the US Constitution, this DTD has details that are
never actually used, so I want to clean it up. Is there any tool that
looks at existing documents and compares with the DTD they use?

[I can think of other possible uses for such a tool, so I thought
someone might have invented it. I have XML Spy but do not see a feature
that would do this.]
```

What the original poster needs is a tool for reading a DTD and finding out if the sample files actually use all the parts of the DTD. This is not exactly what the DTD generator in Section 7.6 [Generating a DTD from a sample file], page 70, does. But it would be a practical solution to let the DTD generator produce a DTD for the sample files and compare the produced DTD with the old original DTD file.

Someone else posted an alternative solution, employing a bunch of tools from the Unix tool set:

```
I did this as part of a migration from TEI SGML to XML. Basically:

a) run nsgmls over the documents and produce ESIS
b) use awk to extract the element type names
c) sort and uniq them
d) use Perl::SGML to read the DTD and list the element type names
e) sort them
f) caseless join the two lists with -a to spit out the non-matches

If you're not using a Unix-based system, I think Cygwin can run these tools.
```

Whatever solution you prefer, these tools serve the user well on the most popular platforms available.

6.5 Working with XML paths

Most programming languages today offer some support for reading XML files. But unlike `gawk-xml`, most other languages map the XML file to a tree-like memory-resident data structure. This allows for convenient access of all elements of the XML file in any desired order; not just sequentially one-at-a-time like in `gawk-xml`. One user of such a language came up with a common problem in the newsgroup comp.text.xml (`news://comp.text.xml`) and asked for a solution. When reading the following XML data, notice the two

`item` elements containing structurally similar sub-elements. Each `item` has a `PPrice` and a `BCQuant` sub-element, containg price and quantity of the `items`. The user asked

```
I have an XML like this:

<?xml version="1.0" encoding="UTF-8"?>
<invoice>
        <bill>
            <BId>20</BId>
            <CId>73</CId>
            <BDate>2006-01-10</BDate>
            <BStatus>0</BStatus>
        </bill>
    <billitems>
        <item>
            <PName>Kiwi</PName>
            <PPrice>0.900</PPrice>
            <PId>1</PId>
            <BCQuant>15</BCQuant>
        </item>
        <item>
            <PName>Apfel</PName>
            <PPrice>0.500</PPrice>
            <PId>3</PId>
            <BCQuant>10</BCQuant>
        </item>
    </billitems>
</invoice>

Now I want to have the sum of /invoice/billitems/item/BCQuant * /invoice/billitems/ite

(=total price)
```

His last sentence sums it all up: He wants the total cost over all `items`, yielding the summation of the product of `PPrice` and `BCQuant`. He identifies the variables to be multiplied with *paths* which resemble file names in a Unix file system. The notation

```
/invoice/billitems/item/BCQuant * /invoice/billitems/item/PPrice
```

is quite a convenient way of addressing variables in an XML document. Some programming languages allow the user to apply this notation directly for addressing variables. For users of these languages it is often hard to adjust their habits to `gawk-xml`'s way of tackling a problem. In `gawk-xml`, it is *not* possible to use such paths for direct access to variables. But it is possible to use such paths in AWK patterns for matching the current location in the XML document. Look at the following solution and you will understand how to apply paths in `gawk-xml`. The crucial point to understand is that there is a predefined variable `XMLPATH` which always contains the path of the location which is currently under observation. The very first line of the solution is the basis of access to the variables `PPrice` and `BCQuant`. Each time some character data is read, the script deposits its content into an associative array `data` with the *path* name of the variable as the index into the array. As a consequence,

this associative array `data` maps the variable name (`/invoice/billitems/item/BCQuant`) to its value (`15`), but only for the short time interval when one XML element `item` is being read.

```
@load "xml"
XMLCHARDATA { data[XMLPATH] = $0 }
XMLENDELEM == "item" {
  sum += data["/invoice/billitems/item/BCQuant"] * \
         data["/invoice/billitems/item/PPrice" ]
}
END { printf "Sum = %.3f\n",sum }
```

The summation takes places each time when the reading of one element `item` is completed; when `XMLENDELEM == "item"`. At this point in time the quantity and the price have definitely been stored in the array `data`. After completion of the XML document, the summation process has ended and the only thing left to do is printing the result.

This simple technique (mapping a path to a value with `data[XMLPATH] = $0`) is the key to *later* accessing data *somewhere else in the tree*. Notice the subtle difference between languages like XSL which store the complete XML document in a tree (DOM (`http://www.w3.org/TR/REC-DOM-Level-1/`)) and `gawk-xml`. With `gawk-xml` only those parts of the tree are stored in precious memory which are really necessary for random access. The only inconvenience is that the user has to identify these parts himself and store the data explicitly. Other languages will do the storage implicitly (without writing any code), but the user has abandoned control over the granularity of data storage.

After a detailed analysis you might find a serious limitation in this simple approach. It only works for a character data block inside a markup block when there is no other tag inside this markup block. In other words: Only when the node in the XML tree is a terminal node (a leaf, like number 3, 5, 6, 8, 9, 11, 12 in Figure 1.3 and Figure 1.2), will character data be stored in `data[XMLPATH]` as expected. If you are also interested in accessing character data of *non-terminal* nodes in the XML tree (like number 2, 4, 7, 10), you will need a more sophisticated approach:

```
@load "xml"
XMLSTARTELEM { delete          data[XMLPATH]    }
XMLCHARDATA  { data[XMLPATH] = data[XMLPATH] $0 }
```

The key difference is that the last line now successively *accumulates* character data blocks of each non-terminal node while going through the XML tree. Only after starting to read another node of the same kind (same tag name, a *sibling*) will the accumulated character data be cleared. Clearing is really necessary, otherwise the character data of all nodes of same kind and depth would accumulate. This kind of accumulation is undesirable because we expect character data in one `data[XMLPATH]` to contain only the text of one node and not the text of other nodes at the same nesting level. But you are free to adapt this behavior to your needs, of course.

7 Some Advanced Applications

Unlike the previous chapter, this chapter really provides complete application programs doing non-trivial work. Inspite of the sophisticated nature of the tasks, the source code of some of these applications still fits onto one page. But most of the source code had to be split up into two or three parts.

7.1 Copying and Modifying with the `xmlcopy.awk` library script

XML data is traditionally associated with Internet applications because this data looks so similar to the HTML data used on the Internet. But after more than 10 years of use, the XML data format has been found to be useful in many more areas, which have different needs. For example, data measured periodically in remote locations is nowadays often encoded as XML data. It is not only the improved readability of XML data (as opposed to proprietary binary formats of the past), but also the tree structure of tagged XML data that is so pleasant for users. If you need to add one more measured data type to your format, no problem, the measuring device just adds an XML attribute or another tag to the XML data tree. The application software reading the data can safely ignore the additional data type and is still able to read the *new* data format. So far so good. But then a device sends us data like the one in Figure 7.1.

```
<?xml version="1.0"?>
<MONITORINGSTATIONS>
   <C_CD>ES</C_CD>
   <DIST_CD>ES100</DIST_CD>
   <NAME>CATALU&#xD1;A</NAME>
   <MONITORINGSTATION>
     <EU_CD>ES_1011503</EU_CD>
     <MS_CD>1011503</MS_CD>
     <LON>0,67891</LON>
     <LAT>40.98765</LAT>
     <PARAMETER>Particulate Matter &lt; 10 &#xB5;m</PARAMETER>
     <STATISTIC>Days with maximum value &gt; 100 ppm</STATISTIC>
     <VALUE>10</VALUE>
     <URL>http://www.some.domain.es?query=1&amp;argument=2</URL>
   </MONITORINGSTATION>
</MONITORINGSTATIONS>
```

Figure 7.1: The file `remote_data.xml` contains data measured in a remote location

If you skim over the data, you might find three places that look odd:

- Inside the **NAME** tag, the capital letter N with tilde in CATALUÑA.
- Inside the **LON** tag, the comma used as decimal separator for a floating point number.
- Inside the **PARAMETER** tag, the μ symbol encoded as *µ*

What we need is a script that rectifies these quirks on the XML data while leaving the tree structure untouched. To be more precise: We need a script that

- Converts all the funny *special characters* to the character encoding we need.

- Replaces the comma with a decimal point, but only in the numbers inside a **LON** tag.
- Copies everything else verbatim as it was in the original XML data.

You can find a solution in Figure 7.2. It begins with an include statement that includes a library script named `xmlcopy.awk`. In the second line, we set the desired character encoding before any data is read. This is obvious. But why is the encoding name (`"ISO-8859-1"`) copied into the `XMLATTR["ENCODING"]` variable ? Didn't we learn in Section 3.4 [Character data and encoding of character sets], page 23, that `XMLATTR["ENCODING"]` reflects the encoding of the *input* data and that there is no use in overwriting this read-only variable ? That's true, the `gawk-xml` reader simply ignores anything we write into `XMLATTR["ENCODING"]`. But in a moment you will see that a function inside the library script evaluates this *faked encoding* variable. The fourth line of the script is obvious again: Inside character data of a **LON** tag, the comma is replaced with a decimal point. And finally, the last line contains the invocation of the `XmlCopy()` function.

```
@include xmlcopy
BEGIN                               { XMLCHARSET          = "ISO-8859-1" }
XMLATTR["VERSION"]                  { XMLATTR["ENCODING"] = XMLCHARSET   }
XMLCHARDATA && XMLPATH ~ /\/LON$/  { gsub(",", ".")                     }
{ XmlCopy() }
```

Figure 7.2: The script `modify.awk` slightly modifies the XML data

All the magic of evaluating the faked encoding name and copying everything is done inside the library script. Just like the `getXMLEVENT.awk` that was explained in Section 2.3 [A portable subset of `gawk-xml`], page 17, this library script may be found in one of several places:

- The `gawk-xml` distribution file contains a copy in the `awklib/xml` directory.
- If `gawk-xml` has already been installed on your host machine, a copy of the file should be in the directory of shared source (which usually resides at a place like `/usr/share/awk/` on GNU/Linux machines).

Have a look at your copy of the `xmlcopy.awk` library script. Notice that the script contains nothing but the declaration of the `XmlCopy()` function. Also notice that the function gets invoked only after all manipulations on the data have been done. The result of a successful run can be seen in Figure 7.3. Shortly after opening the input file, an `XMLEVENT` of type `DECLARATION` occurs, but there is no `XMLATTR["ENCODING"]` variable set because the input data doesn't contain such a declaration. That's the place where our script in Figure 7.2 comes in and sets the declaration in the right moment. So, the `XmlCopy()` function will happily print an encoding name.

```
gawk -f modify.awk remote_data.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<MONITORINGSTATIONS>
   <C_CD>ES</C_CD>
   <DIST_CD>ES100</DIST_CD>
   <NAME>CATALUÑA</NAME>
   <MONITORINGSTATION>
     <EU_CD>ES_1011503</EU_CD>
     <MS_CD>1011503</MS_CD>
     <LON>0.67891</LON>
     <LAT>40.98765</LAT>
     <PARAMETER>Particulate Matter &lt; 10 µm</PARAMETER>
     <STATISTIC>Days with maximum value > 100 ppm</STATISTIC>
     <VALUE>10</VALUE>
     <URL>http://www.some.domain.es?query=1&amp;argument=2</URL>
   </MONITORINGSTATION>
</MONITORINGSTATIONS>
```

Figure 7.3: The output data of `modify.awk` is slightly modified

7.2 Reading an RSS news feed

The Internet is a convenient source of news data. Most of the times we use a browser to read the HTML files that are transported via the HTTP protocol to us. But sometimes there is no browser at hand or we don't want the news data to be visualized immediately. A news ticker displaying news headings in a very small window is an example. For such cases, a special news format has been established, the RSS format (`http://www.rss-specifications.com`). The protocol for transporting the data is still HTTP, but now the content is not HTML anymore, but XML with a simple structure (see Figure 7.4 for an example). The root node in the example tells us that we have received data structured according to version 0.91 of the RSS specification. Node number 2 identifies the news channel to us; augmented by its child nodes 3, 4 and 5, which contain title, link and description of the source. But we are not interested in these; we are interested in the titles of the news items. And those are contained in a sequence of nodes like node 6 (only one of them being depicted here).

What we want as textual output is a short list of news titles – each of them numbered, titled and linked like in Figure 7.5. How can we collect the data for each news item while traversing all the nodes and how do we know *when* we have finished collecting data of one item and we are ready to print ? The idea is to wait for the end of an item such as node 6. Notice that the tree is parsed *depth-first*, so when leaving node 6 (when pattern `XMLENDELEM == "item"` triggers), its child nodes 7 and 8 have already been parsed earlier. The most recent data in nodes 7 and 8 contained the title and the link to be printed. You may ask *How do we access data of a node that has already been traversed earlier ?* The answer is that we store textual data in advance (when XMLCHARDATA triggers). At that moment we don't know yet if the stored data is title or link, but when `XMLENDELEM == "title"` triggers, we know that the data was a title and we can remember it as such. I know this sounds complicated and you definitely need some prior experience in AWK or event-based programming to grasp it.

If you are confused by these explanations, you will be delighted to see that all this mumbling is contained in just 4 lines of code (inside the `while` loop). It is a bit surprising that these 4 lines are enough to select all the news items from the tree and ignore nodes 3, 4 and 5. How do we manage to ignore node 3, 4 and 5 ? Well, actually we don't ignore them. They are `title` and `link` nodes and their content is stored in the variable `data`. But the content of nodes 3 and 4 never gets printed because printing happens only when leaving a node of type `item`.

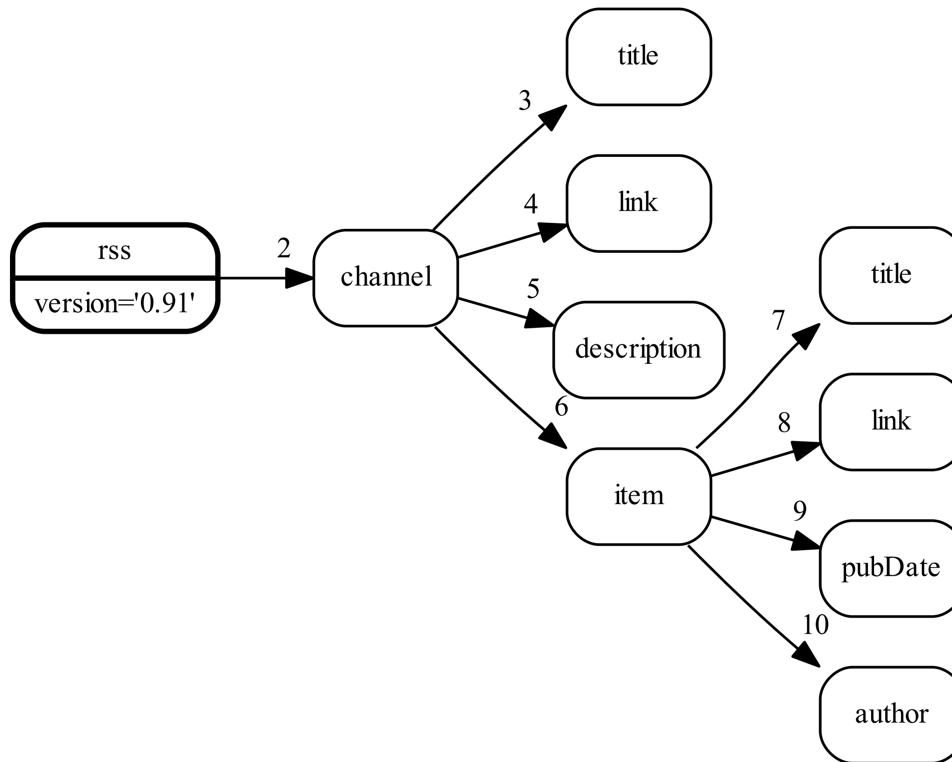

Figure 7.4: This is an example node structure of XML data from an RSS news feed

```
1.      Playing the waiting game        http://www.theinquirer.net/?article=18979
2.      Carbon-dating the Internet      http://www.theinquirer.net/?article=18978
3.      LCD industry walking a margin tightrope http://www.theinquirer.net/?article=18977
4.      Just how irritating can you get?        http://www.theinquirer.net/?article=18976
5.      US to take over the entire Internet     http://www.theinquirer.net/?article=18975
6.      AMD 90 nano shipments 50% by year end   http://www.theinquirer.net/?article=18974
```

Figure 7.5: These are news titles from an RSS news feed

It turns out that traversing the XML file is the easiest part. Retrieving the file from the Internet is a bit more complicated. It would have been wonderful if the news data from the Internet could have been treated as XML data at the moment it comes pouring in hot off the rumour mill (`http://www.theinquirer.net/feeds/rss`). But unfortunately, the XML data comes with a header, which does not follow the XML rules – it is an HTTP header. Therefore, we first have to swallow the HTTP header, then read all the lines from the news feed as ASCII lines and store them into a temporary file. After closing the tem-

porary file, we can re-open the file as an XML file and traverse the news nodes as described above.

```
@load "xml"
BEGIN {
  if (ARGC != 3) {
    print "get_rss_feed - retrieve RSS news via HTTP 1.0"
    print "IN:\n    host name and feed as a command-line parameter"
    print "OUT:\n    the news content on stdout"
    print "EXAMPLE:"
    print "    gawk -f get_rss_feed.awk www.TheInquirer.Net inquirer.rss"
    print "JK 2004-10-06"
    exit
  }
  host = ARGV[1]; ARGV[1] = ""
  feed = ARGV[2]; ARGV[2] = ""
  # Switch off XML mode while reading and storing data.
  XMLMODE=0
  # When connecting, use port number 80 on host
  HttpService = "/inet/tcp/0/" host "/80"
  ORS = RS = "\r\n\r\n"
  print "GET /" feed " HTTP/1.0" |& HttpService
  HttpService                    |& getline Header
  # We need a temporary file for the XML content.
  feed_file="feed.rss"
  # Make feed_file an empty file.
  printf "" > feed_file
  # Put each XML line into feed_file.
  while ((HttpService |& getline) > 0)
    printf "%s", $0 >> feed_file
  close(HttpService)  # this is optional  since connection is empty
  close(feed_file)    # this is essential since we re-read the file

  # Read feed_file (XML) and print a simplified summary (ASCII).
  XMLMODE=1
  XMLCHARSET="ISO-8859-1"
  # While printing, use \n as line separator again.
  ORS="\n"
  while ((getline < feed_file) > 0) {
    if (XMLCHARDATA            ) { data  =   $0 }
    if (XMLENDELEM  == "title" ) { title = data }
    if (XMLENDELEM  == "link"  ) { link  = data }
    if (XMLENDELEM  == "item"  ) { print ++n ".\t" title "\t" link }
  }
}
```

You can find more info about the data coming from RSS news feeds in the fine article *What is RSS* (`http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html`). Digging

deeper into details you will find that there are many similar structural definitions which all call themselves RSS, but have differing content. Our script above was written in such a way to make sure that the script understands all different RSS sources, but this could only be achieved at the expense of leaving details out.

There is another problem with RSS feeds. For example, Yahoo also offers RSS news feeds. But if you use the script above for retrieval, Yahoo will send HTML data and not proper XML data. This happens because the RSS standards were not properly defined and Yahoo's HTTP server does not understand our request for RSS data.

7.3 Using a service via SOAP

In Section 7.2 [Reading an RSS news feed], page 53, we have seen how a simple service on the Internet can be used. The request to the service was a single line with the name of the service. Only the response of the server consisted of XML data. What if the request itself contains several parameters of various types, possibly containing textual data with newline characters or foreign symbols ? Classical services like Yahoo's stock quotes have found a way to pass tons of parameters by appending the parameters to the `GET` line of the HTTP request. Practice has shown that such overly long `GET` lines are not only *awk*ward (which we could accept) but also insufficient when object oriented services are needed. The need for a clean implementation of object oriented services was the motivation behind the invention of the SOAP protocol (`http://www.w3.org/TR/soap`). Instead of compressing request parameters into a single line, XML encoded data is used for passing parameters to SOAP services. SOAP still uses HTTP for transportation, but the parameters are now transmitted with the `POST` method of HTTP (which allows for passing data in the body of the request, unlike the `GET` method).

In this section we will write a client for a SOAP service. You can find a very short and formalized description of the Barnes & Noble Price Quote service (`http://www.ibm.com/developerworks/library/x-tipjaxrpc/`) on the Internet. The user can send the ISBN of a book to this service and it will return him some XML data containing the price of the book. You may argue that this example service needs only one parameter and should therefore be implemented without SOAP and XML. This is true, but the SOAP implementation is good enough to reveal the basic principles of operation. If you are not convinced and would prefer a service which really exploits SOAP's ability to pass structured data along with the request, you should have a look at a list on the Internet, which presents many publicly available SOAP services (`http://www.soapclient.com/XmethodsServices.html`). I urge you to look this page up, it is really enlightening what you can find there. Anyone interested in the inner working of more complex services should click on the *Try it* link of an example service. Behind the *Try it* link is some kind of debugger for SOAP requests, revealing the content of request and response in Pseudocode, raw, tree or interactive XML rendering. I have learned much from this *SOAPscope*.

The author of the Barnes & Noble Price Quote service (Mustafa Basgun) has also written a client for his service. In a fine article on the Internet, the author described how he implemented a GUI-based client interface with the help of *Flash MX*. From this article, we take the following citation, which explains some more details about what SOAP is:

> Simple Object Access Protocol (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based

> protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.

All of the parts he mentions can be seen in the example in Figure 7.6. The example shows the rendering (as a degenerated tree) of a SOAP request in XML format. The root node is the envelop mentioned in the citation. Details on how to process the XML data (Schema and encoding) are declared in the attributes of the root node. Node number 3 contains the remote procedure call `getPrice` that we will use to retrieve the price of a book whose ISBN is contained in the character data of node number 4. Notice that node 4 contains not only the ISBN itself but also declares the data type `xs:string` of the parameter passed to the remote procedure `getPrice` (being the ISBN).

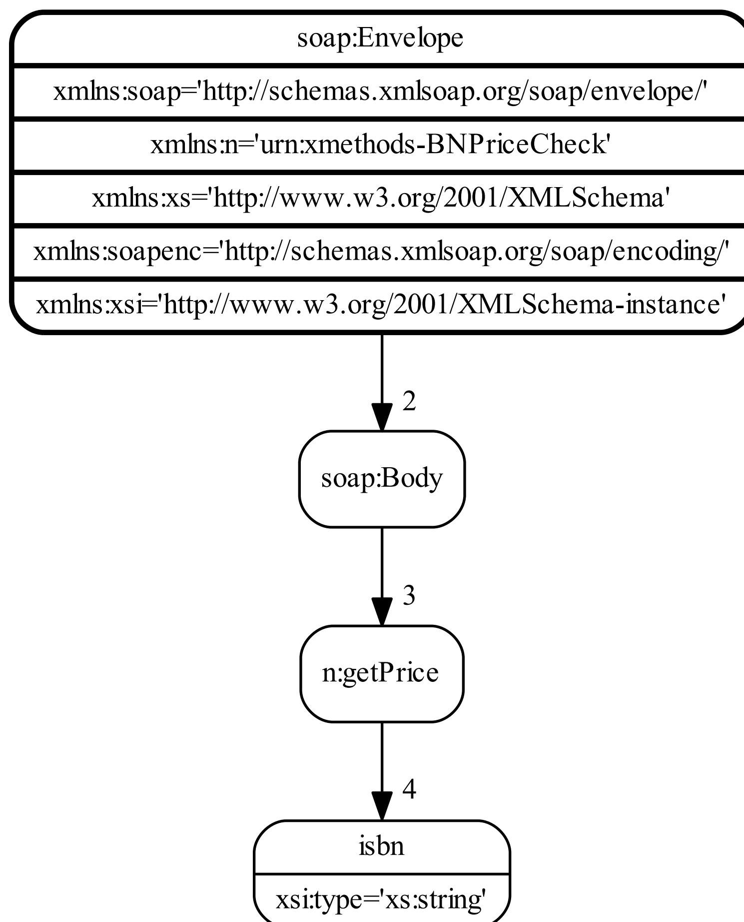

Figure 7.6: Request for a book price in SOAP format

Before we start coding the SOAP client, we have to find out how the response to this request will look like. The tree in Figure 7.7 is the XML data which comes as a response and that we have to traverse when looking for the price of the book. A proper client would

analyze the tree thoroughly and first watch out for the type of node encountered. The structure of Figure 7.7 will only be returned if the request was successful; if the request had failed, we would be confronted with different kinds of nodes describing the failure. It is one of the advantages of SOAP that the response is not a static number or string, but a tree with varying content. Error messages are not simply numbered in a cryptic way, they come as XML elements with specific tag names and character data describing the problem. But at the moment, we are only interested in the successful case of seeing node 4 (tag `return`), being of type `xsd:float` and containing the price as character data.

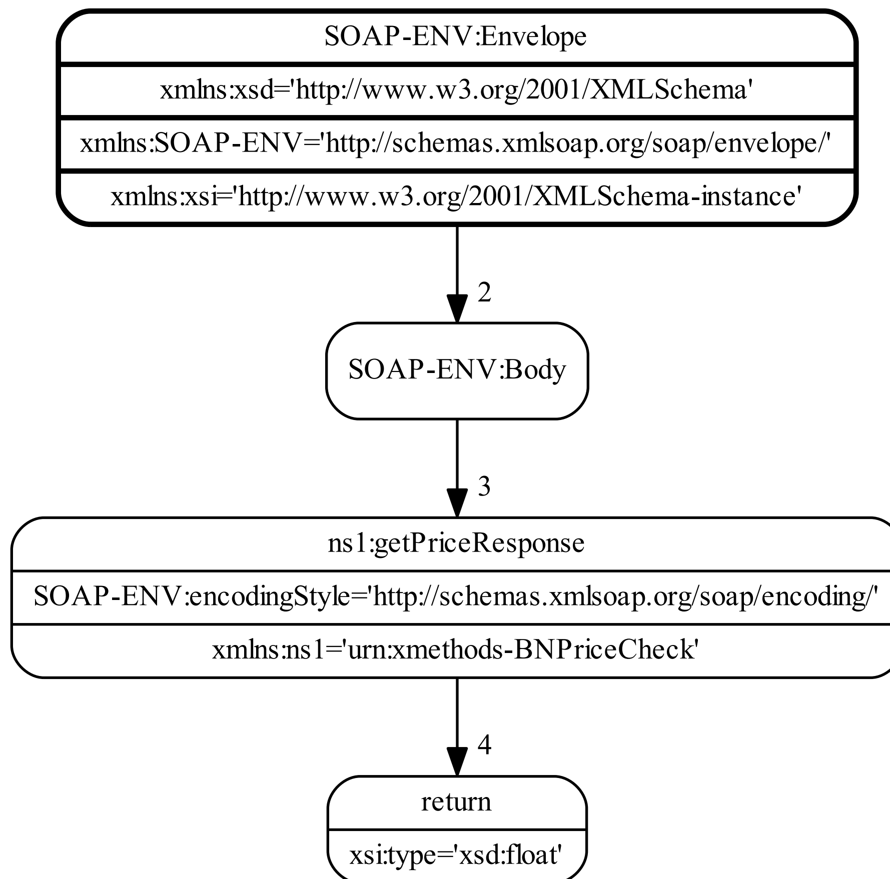

Figure 7.7: Response to a book price request in SOAP format

The first part of our SOAP client in Figure 7.8 looks very similar to the RSS client. The `isbn` is passed as a parameter from the command line while the `host` name and the service identifier `soap` are fixed. Looking at the variable `response`, you will recognize the tree from Figure 7.6. Only the `isbn` is not fixed but inserted as a variable into the XML data that will later be sent to the SOAP server.

```
@load "xml"
BEGIN {
  if (ARGC != 2) {
    print "soap_book_price_quote - request price of a book via SOAP"
    print "IN:\n    ISBN of a book as a command-line parameter"
    print "OUT:\n    the price of the book on stdout"
    print "EXAMPLE:"
    print "    gawk -f soap_book_price_quote.awk 0596000707"
    print "JK 2004-10-17"
    exit
  }
  host = "services.xmethods.net"   # The name of the server to contact.
  soap = "soap/servlet/rpcrouter"  # The identifier of the service.
  isbn = ARGV[1]; ARGV[1] = ""
  # Switch off XML mode while reading and storing data.
  XMLMODE=0
  # Build up the SOAP request and integrate "isbn" variable.
  request="\
    <soap:Envelope xmlns:n='urn:xmethods-BNPriceCheck'           \
       xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'     \
       xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'  \
       xmlns:xs='http://www.w3.org/2001/XMLSchema'                \
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>     \
       <soap:Body>                                                \
          <n:getPrice>                                            \
             <isbn xsi:type='xs:string'>" isbn "</isbn>           \
          </n:getPrice>                                           \
       </soap:Body>                                               \
    </soap:Envelope>"
```

Figure 7.8: First part of `soap_book_price_quote.awk` builds up a SOAP request

The second and third part of our SOAP client resemble the RSS client even more. But if you compare both more closely, you will find some interesting differences.

- The variable `ORS` is not used anymore because we handle the header line differently here.
- The header itself now begins with HTTP's `POST` method and not the `GET` method.
- There are more header lines sent. Most SOAP services require the client to specify the content type and the content length because the HTTP servers hosting the service are used to receiving this kind of information.

```
    # When connecting, use port number 80 on host.
    HttpService = "/inet/tcp/0/" host "/80"
    # Setting RS is necessary for separating header from XML reply.
    RS = "\r\n\r\n"
    # Send out a SOAP-compliant request. First the header.
    print "POST  " soap " HTTP/1.0"                  |& HttpService
    print "Host: " host                              |& HttpService
    print "Content-Type: text/xml; charset=\"utf-8\"" |& HttpService
    print "Content-Length: " length(request)         |& HttpService
    # Now separate header from request and then send the request.
    print "" |& HttpService
    print request |& HttpService
```

Figure 7.9: Second part of `soap_book_price_quote.awk` sends the SOAP request

Having sent the request, the only thing left to do is receiving the response and traversing the XML tree. Just like the RSS client, the SOAP client stores the XML response in a file temporarily and opens this file as an XML file. While traversing the XML tree, our client behaves very simple minded: character data is remembered and printed as soon as a node with tag name `return` occurs.

```
    # Receive the reply and save it.
    HttpService   |& getline Header
    # We need a temporary file for the XML content.
    soap_file="soap.xml"
    # Make soap_file an empty file.
    printf "" > soap_file
    # Put each XML line into soap_file.
    while ((HttpService |& getline) > 0)
      printf "%s", $0 >> soap_file
    close(HttpService)  # this is optional  since connection is empty
    close(soap_file)    # this is essential since we re-read the file

    # Read soap_file (XML) and print the price of the book (ASCII).
    XMLMODE=1
    while ((getline < soap_file) > 0) {
      if (XMLCHARDATA            ) { price  =   $0 }
      if (XMLENDELEM  == "return") { print "The book costs", price, "US$."}
    }
  }
```

Figure 7.10: Third and final part of `soap_book_price_quote.awk` reads the SOAP response

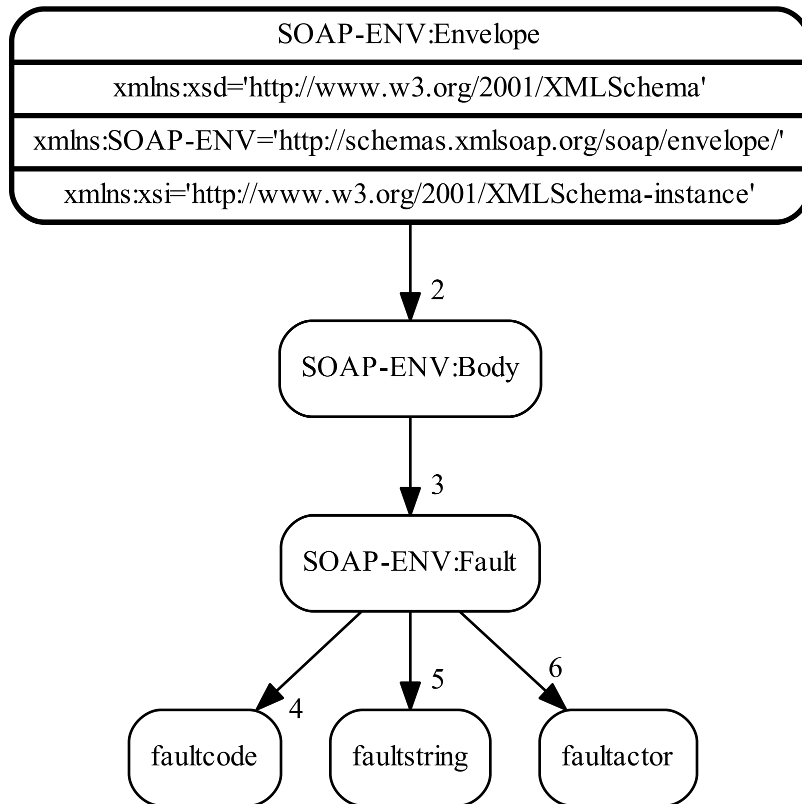

Figure 7.11: Response to a SOAP request in case of an error

What would happen in case of an error ? Each of the following cases would have to be handled in more detail if we were writing a proper SOAP client.

- When the service does not know the ISBN, it returns `-1.0` as the prices. This is not a problem for our client, but the user has to keep in mind to check for a negative price.
- When the network connection to the SOAP service cannot be established, then the client might *hang* for a very long time, doing nothing and finally terminate with an error, but without a textual answer. Or the client terminates immediately if, for example, a firewall rejects the request.
- When something is missing in the header of the HTTP request or the XML tree of the request is not complete, a proper SOAP response will be received. But you will find no `return` node in the response (and therefore the client will print nothing). Instead, the response will contain nodes of type `SOAP-ENV:Fault`, `faultcode`, `faultstring` and `faultactor`. This situation is depicted in Figure 7.11.
- When the service terminates abnormally, it will be unable to respond with a proper XML message. In this case, the hosting HTTP server often inserts a message in its own idiom (mostly HTML mumbling).

```
<h1>Error: 400</h1>
Error unmarshalling envelope: SOAP-ENV:VersionMismatch:
```

```
Envelope element must be associated with the
'http://schemas.xmlsoap.org/soap/envelope/' namespace.
```

I began each of the cases above with the word *When* because it is not the question *if* such a case will ever happen but only *when* it will happen. When writing software, it is always essential to distinguish cases that can happen (by evaluating return codes or by catching exceptions for example). But when writing software that connects to a network it is inevitable to do so; otherwise the networking software would be unreliable. So, most real world clients will be much longer than the one we wrote in this section. But in many cases it is not really complicated to handle error conditions. For example, by inserting the following single line at the end of the `while` loop's body, you can do a first step toward proper error handling.

```
if (XMLENDELEM  ~  /^fault/) { print XMLENDELEM ":", price }
```

When the client should ever receive a response like the one in Figure 7.11, it would print the detected messages contained in nodes 4, 5 and 6.

```
faultcode: SOAP-ENV:Protocol
faultstring: Content length must be specified.
faultactor: /soap/servlet/rpcrouter
```

7.4 Loading XML data into PostgreSQL

In the previous section we have seen how XML can be used as a data-exchange-format during a database query. Using XML in such database retrievals is quite commonplace today. But the actual storage format for the large databases in the background is usually not XML. Many proprietary solutions for database storage have established their niche markets over the decades. These will certainly not disappear just because a geeky new format like XML emerged out of the blue. As a consequence, the need for conversion of data between established databases and XML arises frequently in and around commercial applications. Fortunately, we have the query language SQL as an accepted standard for expressing the query. But unfortunately, the actual interface (the transport mechanism) for request and delivery of queries and results is not standardized at all. The rest of this section describes how this problem was solved by creating a GNU Awk extension for one specific database: PostgreSQL. All proprietary access mechanisms are encapsulated into an extension and an application script uses the extension. The problem at hand is to read a small contact database file (XML, Figure 7.13) and write the data into a PostgreSQL database with the help of two GNU Awk extensions:

'`xml`' for reading data from an XML file

'`pgsql`' for accessing the database interface of PostgreSQL

So it is not surprising that the application script in Figure 7.12 begins with loading both extensions. GNU Awk extensions like `pgsql` are usually implemented so that they make the underlying C interface (`http://www.postgresql.org/docs/8.0/interactive/libpq.html`) accessible to the writer of the application script.

```
@load "xml"
@load "pgsql"

BEGIN {
  # Note: should pass an argument to pg_connect containing PQconnectdb
  # options, as discussed here:
  #  http://www.postgresql.org/docs/8.0/interactive/libpq.html#LIBPQ-CONNECT
  # Or the parameters can be set in the environment, as discussed here:
  #  http://www.postgresql.org/docs/8.0/interactive/libpq-envars.html
  # For example, a typical call might be:
  #   pg_connect("host=pgsql_server dbname=my_database")
  if ((dbconn = pg_connect()) == "") {
    printf "pg_connect failed: %s\n", ERRNO > "/dev/stderr"
    exit 1
  }

  # these are the columns in the table
  ncols = split("name email work cell company address", col)

  # create a temporary table
  sql = "CREATE TEMPORARY TABLE tmp ("
  for (i = 1; i <= ncols; i++) {
    if (i > 1)
      sql = (sql",")
    sql = (sql" "col[i]" varchar")
  }
  sql = (sql")")
  if ((res = pg_exec(dbconn, sql)) !~ /^OK /) {
    printf "Cannot create temporary table: %s, ERRNO = %s\n",
           res, ERRNO > "/dev/stderr"
    exit 1
  }

  # create a prepared insert statement
  sql = ("INSERT INTO tmp ("col[1])
  for (i = 2; i <= ncols; i++)
    sql = (sql", "col[i])
  sql = (sql") VALUES ($1")
  for (i = 2; i <= ncols; i++)
    sql = (sql", $"i)
  sql = (sql")")
  if ((insert_statement = pg_prepare(dbconn, sql)) == "") {
    printf "pg_prepare(%s) failed: %s\n",sql,ERRNO > "/dev/stderr"
    exit 1
  }
}
```

Figure 7.12: First part of `testxml2pgsql.awk` connects to PostgreSQL

For example, the function pg_connect is just a wrapper around a C function with an almost identical name. This transparency is good practice, but not compulsory. Other GNU Awk extensions may choose to implement some opacity in the design of the interface.

Unsuccessful connection attempts are reported to the user of the application script before termination. After a successful connection (with pg_connect), the script tells PostgreSQL about the structure of the database. This structure is of course inspired by the format of the contact database in Figure 7.13. Each field in the database (name, email, work, cell, company address) is declared with an SQL statement that is executed by PostgreSQL. After creation of this table, PostgreSQL is expected to respond with an OK message, otherwise the attempt to create the table has to be aborted. Finally, a prepared insert statement tells PostgreSQL about details (fieldwidths) of the database.

```
<?xml version="1.0" encoding="utf-8"?>
<contact_database>

<contact>
<name>Joe Smith</name>
<phone type="work">1-212-555-1212</phone>
<phone type="cell">1-917-555-1212</phone>
<email>joe.smith@acme.com</email>
<company>Acme</company>
<address>32 Maple St., New York, NY</address>
</contact>

<contact>
<name>Ellen Jones</name>
<phone type="work">1-310-555-1212</phone>
<email>ellen.jones@widget.com</email>
<company>Widget Inc.</company>
<address>137 Main St., Los Angeles, CA</address>
</contact>

<contact>
<name>Ralph Simpson</name>
<phone type="work">1-312-555-1212</phone>
<phone type="cell">1-773-555-1212</phone>
<company>General Motors</company>
<address>13 Elm St., Chicago, IL</address>
</contact>

</contact_database>
```

Figure 7.13: The contact database to be stored with PostgreSQL

Now that the structure of the database is known to PostgreSQL, we are ready to read the actual data. Assuming that the script has been stored in a file testxml2pgsql.awk and the XML data in a file sample.xml, we can invoke the application like this:

```
gawk -f testxml2pgsql.awk < sample.xml
```

```
name|email|work|cell|company|address
Joe Smith|joe.smith@acme.com|1-212-555-1212|1-917-555-1212|Acme|32 Maple St., New York
Ellen Jones|ellen.jones@widget.com|1-310-555-1212|<NULL>|Widget Inc.|137 Main St., Los
Ralph Simpson|<NULL>|1-312-555-1212|1-773-555-1212|General Motors|13 Elm St., Chicago,
```

Notice that the data file is not passed as a parameter to the interpreter, but the data file is redirected (`< sample.xml`) to the standard input of the interpreter. This way of invocation will hide the file's name from the application script, but still allow the script to handle incoming XML data conveniently inside the curly braces of Figure 7.14, following the *pattern-action* model of AWK. You will also recognize that some fields which were empty in the XML file appear as `<NULL>` fields in the output of the script. Obviously, while reading the XML file, the application script in Figure 7.14 takes care which fields are filled with data and which are empty in a data record.

```
{
  switch (XMLEVENT) {
    case "STARTELEM":
      if ("type" in XMLATTR)
        item[XMLPATH] = XMLATTR["type"]
      else
        item[XMLPATH] = XMLNAME
      break
    case "CHARDATA":
      if ($1 != "")
        data[item[XMLPATH]] = (data[item[XMLPATH]] $0)
      break
    case "ENDELEM":
      if (XMLNAME == "contact") {
        # insert the record into the database
        for (i = 1; i <= ncols; i++) {
          if (col[i] in data)
            param[i] = data[col[i]]
        }
        if ((res = pg_execprepared(dbconn, insert_statement,
                                   ncols, param)) !~ /^OK /) {
          printf "Error -- insert failed: %s, ERRNO = %s\n",
                 res, ERRNO > "/dev/stderr"
          exit 1
        }
        delete item
        delete data
        delete param
      }
      break
  }
}
```

Figure 7.14: Second part of `testxml2pgsql.awk` transmits data to PostgreSQL

Most scripts you have seen so far follow the *pattern-action* model of AWK in the way that is described in Section 8.2 [`gawk-xml` Core Language Interface Summary], page 88, as *Verbose Interface*. The script in Figure 7.14 is different in that it employs the *Concise Interface*. Each XML event that comes in is analyzed inside a `switch` statement for its type. When the *Verbose Interface* would have called for an `XMLSTARTELEM` pattern in front of an action, the *Concise Interface* looks at the content of `XMLEVENT` and switches to the `STARTELEM` case for an action to be done. The action itself (collecting data from attribute `type` or the tag name) remains the same in both styles. The case of `CHARDATA` will remain hard to understand, unless you have a look at Section 6.5 [Working with XML paths], page 48, where the idea of collecting data from terminal XML nodes is explained. Remember that most of the data in Figure 7.13 is stored as character data in the terminal XML nodes. Whenever a `contact` node is finished, it is time to store the collected data into PostgreSQL and the variables have to be emptied for collecting data of the next `contact`. All this collecting and storing data will be repeated until there are no more XML events coming in. By then, PostgreSQL will contain the complete database.

```
END {
  if (dbconn != "") {
    # let's take a look at what we have accomplished
    if ((res = pg_exec(dbconn, "SELECT * FROM tmp")) !~ /^TUPLES /)
      printf "Error selecting * from tmp: %s, ERRNO = %s\n",
              res, ERRNO > "/dev/stderr"
    else {
      nf = pg_nfields(res)
      for (i = 0; i < nf; i++) {
        if (i > 0)
          printf "|"
        printf "%s", pg_fname(res, i)
      }
      printf "\n"
      nr = pg_ntuples(res)
      for (row = 0; row < nr; row++) {
        for (i = 0; i < nf; i++) {
          if (i > 0)
            printf "|"
          printf "%s",
                 (pg_getisnull(res,row,i) ? "<NULL>" : pg_getvalue(res,row,i))
        }
        printf "\n"
      }
    }
    pg_disconnect(dbconn)
  }
}
```

Figure 7.15: Final part of `testxml2pgsql.awk` reads back data from PostgreSQL

The third and final part of our application script in Figure 7.15 will help us verify that everything has been stored correctly. In order to do so, we will also see how a PostgreSQL database can be read from within our script. Whenever you have an `END` pattern in a script, the following action will be triggered after all data has been read; irrespective of the success of initializations done earlier. The situation is comparable to the `try (...) catch` sequence in the exception handling of programming languages of lesser importance. In such an "exception handler", there are very few assertions about the state of variables that you can rely on. Therefore, before using any variable, you have to check that it is valid. That's what's done first in Figure 7.15: Reading the database makes sense only when it was actually opened. If it *was* in fact opened, then it makes sense to transmit an SQL statement to PostgreSQL. After a successful transmission (and only then) the returned result can be split up into fields. All fields of all rows are printed, but only for those fields that are non-empty.

7.5 Converting XML data into tree drawings

While reading Chapter 1 [AWK and XML Concepts], page 5, you might have wondered how the DocBook file in Figure 1.2 was turned into the drawing of a tree in Figure 1.3. The drawing was not produced manually but with a conversion tool – implemented as a `gawk` script. The secret in finding a good solution for an imaging problem always is to find the right tool to employ. At the AT&T Labs, there is a project group working on GraphViz (`http://www.graphviz.org/`), an open source software package for drawing graphs. Graphs are data structures which are more general than trees, so they include trees as a special case and the `dot` tool (`http://www.graphviz.org/pdf/dotguide.pdf`) can produce nice drawings like the one in Figure 1.3. Before you go and download the source code of `dot`, have a look at your operating system's distribution media – `dot` comes for free with most GNU/Linux distributions.

But the question remains, how to turn an XML file into the image of a graph ? `dot` only reads textual descriptions (`http://www.graphviz.org/content/dot-language`) of graphs and produces Encapsulated PostScript files, which are suitable for inclusion into documents. These textual descriptions look like Figure 7.16, which contains the `dot` source code for the tree in Figure 1.3. So the question can be recast as how to convert Figure 1.2 into Figure 7.16 ? After a bit of comparison, you will notice that Figure 7.16 essentially has one `struct` line for each node (containing the node's name – the tag of the markup block) and one `struct` line for each edge in the tree (containing the number of the node to which it points). The very first `struct1` is a bit different. `struct1` contains the root node of the XML file. In the tree, this node has no number but it is framed with a bold line, while all the other nodes are numbered and are not framed in a special way. In the remainder of this section, we will find out how the script `outline_dot.awk` in Figure 7.17 converts an XML file into a graph description which can be read by the `dot` tool.

```
digraph G {
  rankdir=LR
  node[shape=Mrecord]
  struct1[label="<f0>book| lang='en'| id='hello-world' "];
  struct1 [style=bold];
  struct2[label="<f0>bookinfo "];
  struct1 -> struct2:f0 [headlabel="2\n\n"]
  struct3[label="<f0>title "];
  struct2 -> struct3:f0 [headlabel="3\n\n"]
  struct4[label="<f0>chapter| id='introduction' "];
  struct1 -> struct4:f0 [headlabel="4\n\n"]
  struct5[label="<f0>title "];
  struct4 -> struct5:f0 [headlabel="5\n\n"]
  struct6[label="<f0>para "];
  struct4 -> struct6:f0 [headlabel="6\n\n"]
  struct7[label="<f0>sect1| id='about-this-book' "];
  struct4 -> struct7:f0 [headlabel="7\n\n"]
  struct8[label="<f0>title "];
  struct7 -> struct8:f0 [headlabel="8\n\n"]
  struct9[label="<f0>para "];
  struct7 -> struct9:f0 [headlabel="9\n\n"]
  struct10[label="<f0>sect1| id='work-in-progress' "];
  struct4 -> struct10:f0 [headlabel="10\n\n"]
  struct11[label="<f0>title "];
  struct10 -> struct11:f0 [headlabel="11\n\n"]
  struct12[label="<f0>para "];
  struct10 -> struct12:f0 [headlabel="12\n\n"]
}
```

Figure 7.16: An example of a tree description for the `dot` tool

Before delving into the details of Figure 7.17, step back for a moment and notice the structural similarity between this `gawk` script and the one in Figure 3.2. Both determine the depth of each node while traversing the tree. In the `BEGIN` section of Figure 7.17, only the three `print` lines were added, which produce the three first lines of Figure 7.16. The same holds for the one `print` line in the `END` section of Figure 7.17, which only finalizes the textual description of the tree in Figure 7.16. As a consequence, all the `struct` lines in Figure 7.16 are produced while traversing the tree in the `XMLSTARTELEM` section of Figure 7.17.

Each time we come across a node, two things have to be done:

1. Insert the node into the drawing.
2. Insert an edge from its parent to the node itself into the drawing.

To simplify identification of nodes, the node counter `n` is incremented. Then `n` is appended to the `struct` and allows us to identify each node by name. Identifying nodes through the tag name of the markup block is not possible because tag names are not unique. At this stage we are ready to insert the node into the drawing by printing a line like this:

```
struct3[label="<f0>title "];
```

The label of the node is the right place to insert the tag name of the markup block (`XMLSTARTELEM`). If there are attributes in the node, they are appended to the label after a separator character.

```
@load "xml"
BEGIN {
  print "digraph G {"
  print "  rankdir=LR"
  print "  node[shape=Mrecord]"
}
XMLSTARTELEM {
  n ++
  name[XMLDEPTH] = "struct" n
  printf("%s", "  " name[XMLDEPTH] "[label=\"<f0>" XMLSTARTELEM)
  for (i in XMLATTR)
    printf("| %s='%s'", i, XMLATTR[i])
  print " \"];"
  if (XMLDEPTH==1)
    print "  " name[1], "[style=bold];"
  else
    print "  " name[XMLDEPTH-1], "->", name[XMLDEPTH] ":f0 [headlabel=\""n"\\n\\n\"]"
}
END { print "}" }
```

Figure 7.17: `outline_dot.awk` turns an XML file into a tree description for the `dot` tool

Now that we have a name for the node, we can draw an edge from its parent node to the node itself. The array `name` always contains the identifiers of the most recently traversed node of given depth. Since we are traversing the tree *depth-first*, we can always be sure that the most recently traversed node of a lesser depth is a parent node. With this assertion in mind, we can easily identify the parent by name and print a line from the parent node to the node.

```
struct2 -> struct3:f0 [headlabel="3\n\n"]
```

The root node (`XMLDEPTH==1`) is a special case which is easier to handle. It has no parent, so no edge has to be drawn, but the root node gets a special recognition by framing it with a bold line.

Now, store the script into a file and invoke it. The output of `gawk` is piped directly into `dot`. `dot` is instructed to store Encapsulated PostScript output into a file `tree.eps`. `dot` converts this description into a nice graphical rendering in the PostScript format.

```
gawk -f outline_dot.awk dbfile.xml | dot -Tps2 -o tree.eps
```

7.6 Generating a DTD from a sample file

We have already talked about validation in Section 3.1 [Checking for well-formedness], page 19. There, we have learned that `gawk` does not validate XML files against a DTD. So, does this mean we have to ignore the topic at all ? No, the use of DTDs is so widespread that everyone working with XML files should at least be able to read them. There are at least two good reason why we should take DTDs for serious:

1. If you are given an original DTD (and it is well written), you can learn much more from looking at the DTD than from browsing through a valid XML example file.
2. In real life, only very few of us will ever have to produce one. But when you are confronted with a large XML file (like the one attached to this posting (`http://lists.w3.org/Archives/Public/www-archive/2004Mar/0169.html`)) and you *don't* have a DTD, it will be hard for you to make sense out of it.

In such cases, you wish you had a tool like DTDGenerator – A tool to generate XML DTDs (`http://saxon.sourceforge.net/dtdgen.html`). This is a tool which takes a well-formed XML file and produces a DTD out of it (so that the XML file is valid against the DTD, of course). Let's take Figure 1.2 as an example. This is a DocBook file, which has a well-established DTD named in its header. Imagine the DocBook file was much longer and you had an application which required reading and processing the file. Would you go for the complete DocBook DTD and taylor your application to handle all the details in the DocBook DTD ? Probably not. It is more practical to start with a subset of the DTD which is good enough to describe the file at hand. A DTD Generator will produce a DTD which can serve well as a starting point. The DocBook file in Figure 1.2 for example can be described by the DTD in Figure 7.18. Unlike most DTDs you will find in the wild, this DTD uses indentation to emphasize the structure of the DTD. Attributes are always listed immediately after their element name and the sub-elements occuring in one element follow immediately below, but indented.

You should take some time and try to understand the relationship of the elements and attributes listed in Figure 7.18 and the example file in Figure 1.2. The first line of Figure 7.18 for example tells you that a `book` consists of a sequence of elements, which are either a `chapter` or a `bookinfo`. You can verify this by looking at the drawing in Figure 1.3. The next two lines tell you that a `book` has two mandatory attributes, `lang` and `id`. The rest of Figure 7.18 is indented and describes all other elements and their attributes in the same way. Elements that have no other elements included in them have `#PCDATA` in them.

```
<!ELEMENT book ( chapter | bookinfo )* >
<!ATTLIST book lang CDATA #REQUIRED>
<!ATTLIST book id CDATA #REQUIRED>
  <!ELEMENT chapter ( sect1 | para | title )* >
  <!ATTLIST chapter id CDATA #REQUIRED>
    <!ELEMENT sect1 ( para | title )* >
    <!ATTLIST sect1 id CDATA #REQUIRED>
      <!ELEMENT para ( #PCDATA ) >
      <!ELEMENT title ( #PCDATA ) >
  <!ELEMENT bookinfo ( title )* >
```

Figure 7.18: Example of a DTD, arranged to emphasize nesting structure

The rest of this section consists of the description of the script which produced the DTD in Figure 7.18. The first part of the scripts looks rather similar to Figure 7.17. Both scripts traverse the tree of nodes in the XML file and accumulate information in a very similar way (the array `name` and the variable `XMLDEPTH` for example). Three additional array variables in Figure 7.19 are responsible for storing the information needed for generating a DTD later.

1. `elem[e]` learns all element names and counts how often each element occurs. This is necessary for knowing their names and determining if a certain element occured always with a certain attribute.

2. `child[ep,ec]` learns which element is the child of which other element. This is necessary for generating the details of the `<!ELEMENT ...>` lines in Figure 7.18.

3. `attr[e,a]` learns which element has which attributes. This is necessary for generating the details of the `<!ATTLIST ...>` lines in Figure 7.18.

```
@load "xml"
# Remember each element.
XMLSTARTELEM {
  # Remember the parent names of each child node.
  name[XMLDEPTH] = XMLSTARTELEM
  if (XMLDEPTH>1)
    child[name[XMLDEPTH-1], XMLSTARTELEM] ++
  # Count how often the element occurs.
  elem[XMLSTARTELEM] ++
  # Remember all the attributes with the element.
  for (a in XMLATTR)
    attr[XMLSTARTELEM,a] ++
}

END { print_elem(1, name[1]) }   # name[1] is the root
```

Figure 7.19: First part of `dtd_generator.awk` — collecting information

Having completed its traversal of the tree and knowing all names of elements and attributes and also their nesting structure, the action of the `END` pattern only invokes a function which starts resolving the relationships of elements and attributes and prints them in the form of a proper DTD. Notice that `name[1]` contains the name of the root node of the tree. This means that the description of the DTD begins with the top level element of the XML file (as can be seen in the first line of Figure 7.18).

```
# Print one element (including sub-elements) but only once.
function print_elem(depth, element,   c, atn, chl, n, i, myChildren) {
  if (already_printed[element]++)
    return
  indent=sprintf("%*s", 2*depth-2, "")
  myChildren=""
  for (c in child) {
    split(c, chl, SUBSEP)
    if (element == chl[1]) {
      if (myChildren=="")
        myChildren = chl[2]
      else
        myChildren = myChildren " | " chl[2]
    }
  }
  # If an element has no child nodes, declare it as such.
  if (myChildren=="")
    print indent "<!ELEMENT", element , "( #PCDATA ) >"
  else
    print indent "<!ELEMENT", element , "(", myChildren, ")* >"
  # After the element name itself, list its attributes.
  for (a in attr) {
    split(a, atn, SUBSEP)
    # Treat only those attributes that belong to the current element.
    if (element == atn[1]) {
      # If an attribute occured each time with its element, notice this.
      if (attr[element, atn[2]] == elem[element])
        print indent "<!ATTLIST", element, atn[2], "CDATA #REQUIRED>"
      else
        print indent "<!ATTLIST", element, atn[2], "CDATA #IMPLIED>"
    }
  }
  # Now go through the child nodes of this elements and print them.
  gsub(/[\|]/, " ", myChildren)
  n=split(myChildren, chl)
  for(i=1; i<=n; i++) {
    print_elem(depth+1, chl[i])
    split(myChildren, chl)
  }
}
```

Figure 7.20: Second part of `dtd_generator.awk` — printing the DTD

The first thing this function does is to decide whether the element to be printed has already been printed (if so, don't print it twice). Proper indentation is done by starting each printed line with a number of spaces (twice as much as the indentation levels). Next comes the collection of all child nodes of the current element into the string `myChildren`. AWK's `split` function is used for breaking up the tuple of elements (parent and child) that make up an associative array index. Having found all children, we are ready to print the `<!ELEMENT ... >` line for this element of the DTD. If an element has no children, then it is a leaf of the tree and it is marked as such in the DTD. Otherwise all the children found are printed as belonging to the element.

Finding the right `<!ATTLIST ... >` line is coded in a similar way. Each attribute is checked if it has ever occured with the element and if so, it is printed. The distinction between an attribute that occurs always with the element and an attribute that occurs

sometimes with the element is the first stage of refinement in this generator. But if you analyze the generated DTD a bit, you will notice that the DTD is a rather coarse and liberal DTD.

- The elements are declared in such a way that their children are always allowed to occur in an arbitrary order.
- The elements which are leafs of the tree are always declared to be `#PCDATA`.
- The attributes are always declared to be `CDATA`.

Using the XSD Inference Utility http://msdn2.microsoft.com/en-us/library/aa302302.aspx

Feel free to refine this generator according to your needs. Perhaps, you can even generate a Schema file along the lines of Microsoft's *XSD Inference Utility*, see Using the XSD Inference Utility (`http://msdn2.microsoft.com/en-us/library/aa302302.aspx`). The rest of the function `print_elem()` should be good enough for further extensions. It takes the child nodes of the element (which were collected earlier) and uses the function recursively in order to print each of the children.

7.7 Generating a recursive descent parser from a sample file

It happens rather seldom, but sometimes we have to write a program which reads an XML file tag by tag and looks very carefully at the context of a tag and the character data embedded in it. Such programs detect the sequence, indentation and context of the tags and evaluate all this in an application specific manner, almost like a compiler or an interpreter does. These programs are called parsers. Their creation is not trivial and if you ever have to write a parser, you will be grateful to find a way of producing the first step of a parser automatically from an example file. Quite naturally, some commercial tools exist which promise to generate a parser for you. For example, the XMLBooster XMLBooster (`http://www.xmlbooster.com`) product generates not only a parser (in any language in any of the languages C, C++, C#, COBOL, Delphi, Java or Ada) but also convenient structural documentation and even a GUI for editing your specific XML files. The XMLBooster uses an existing DTD or Schema file to generate all these things. Unlike the XMLBooster, we will not assume that any DTD or Schema file exists for given XML data. We want our parser generator to take specific XML data as input and produce a parser for such data.

In the previous section Section 7.6 [Generating a DTD from a sample file], page 70, we already saw how an XML file was analyzed and a different file was generated, which contained the syntactical relationship between different kinds of tags. As we will see later, a parser can be created in a very similar way. So, in this section we will change the program from the previous section, leaving everything unchanged, except for the function `print_elem()`.

Once more, let's take Figure 1.2 (the DocBook file) as an example. A parser for DocBook files of this kind could begin like the program in Figure 7.21. In the `BEGIN` part of the parser, the very first tag is read by a function `NextElement()` which we will see later. If this very first tag is a `book` tag, then parsing will go on in a function named after the tag. Otherwise, the parser will assume that the root tag of the XML file was not the one expected and the parser terminates with an error message. In the function `parse_book` we see a loop, reading one tag after the other until the closing `book` tag is read. In between, each subsequent tag is checked against the set of allowed tags and another function for handling that tag is

invoked. Unexpected tag names lead to a warning message being emitted, but not to the termination of the parser.

The most important principle in this parser is that **for each tag name, one function exists for parsing tags of its kind**. These functions invoke each other while parsing the XML file (perhaps recursively, if the XML markup blocks were formed recursively). Each of these functions has a header with comments in it, naming the attributes which come with a tag of this name. Now, look at the `parse_book` function and imagine you had to generate such a function. Remember how we stored information about each kind of tag when we wrote the DTD generator. You will find that all the information needed about a tag is already available, we only have to produce a different kind of output here.

```
BEGIN {
  if (NextElement() == "book") {
    parse_book()
  } else {
    print "could not find root element 'book'"
  }
}

function parse_book() {
  # The 'book' node has the following attributes:
  # Mandatory attribute 'lang'
  # Mandatory attribute 'id'

  while (NextElement() && XMLENDELEM != "book") {
    if (XMLSTARTELEM == "chapter") {
      parse_chapter()
    } else    if (XMLSTARTELEM == "bookinfo") {
      parse_bookinfo()
    } else  {
      print "unknown element '" XMLSTARTELEM "' in 'book' line ", XMLROW
    }
  }
}
```

Figure 7.21: Beginning of a generated parser for a very simple DocBook file

Now that the guiding principle (recursive descent) is clear, we can turn to the details. The hardest problem in understanding the parser generator will turn out to be the danger of mixing up the kinds of text and data involved. Whenever you turn in circles while trying to understand what's going on, remember the kind of data you are thinking about:

- The **XML data** that has to be parsed by the *generated* parser.
- The **AWK data structures** for storing the tag relations.
- The **parser** *generator* that we are actually writing.
- The **parser** *generated* by our generator.

Traditional language parsers read their input text token by token. The work is divided up between a low-level character reader and a high-level syntax checker. On the lowest level,

a token is singled out by the *scanner*, which returns the token to the parser itself. In a *generated* parser for XML data, we don't need our own scanner because the scanner is hidden in the XML reader that we use. What remains to be generated is a function for reading the next token upon each invocation. This token-by-token reader in Figure 7.22 is implemented in the *pull-parser* style we have seen earlier. Notice that the function `NextElement()` implementing this reader remains the same in each generated parser. While reading the XML file with `getline`, the reader watches for any of the following events in the token stream:

- `XMLSTARTELEM` is a new tag to be returned.
- `XMLENDELEM` is the end of a markup block to be returned.
- `XMLCHARDATA` is text embedded into a markup block.
- `XMLERROR` is an error indicator, leading to termination.

Text embedded into a markup block is not returned as the function's return value but is stored into the global variable `data`. This function is meant to return the name of a tag — no matter if it is the beginning or the ending of a markup block. If the caller wants to distinguish between beginning or ending of a markup block, he can do so by watching if `XMLSTARTELEM` or `XMLENDELEM` is set. Only when the end of an XML file is reached will an empty string be returned. It is up to the caller to detect when the end of the token stream is reached.

```
@load "xml"
function NextElement() {
  while (getline > 0 && XMLERROR == "" && XMLSTARTELEM == XMLENDELEM)
    if (XMLCHARDATA) data = $0
  if (XMLERROR) {
    print "error in row", XMLROW ", col", XMLCOL ":", XMLERROR
    exit
  }
  return XMLSTARTELEM XMLENDELEM
}
```

Figure 7.22: The pull-style token reader; identical in all generated parsers

All the code you have seen in this section up to here was *generated* code. It makes no sense to copy this code into your own programs. What follows now is the *generator* itself. As mentioned earlier, the generator is identical to the `dtd_generator.awk` of the previous section — you only have to replace the function `print_elem()` with the version you see in Figure 7.23 and Figure 7.24. The beginning of the function `print_elem()` is easy to understand — it generates the function `NextElement()` as you have seen the function in Figure 7.22. We only need `NextElement()` generated once, so we generate it only when the root tag (`depth == 1`) is handled. Just like `NextElement()`, we also need the `BEGIN` pattern of Figure 7.21 only once, so it is generated immediately after `NextElement()`. What follows is the generation of the comments about XML attributes as you have seen them in Figure 7.21. This coding style should not be new to you if you have studied the `dtd_generator.awk`. Notice that each invocation of the `print_elem()` for non-root tags (`depth > 1`) produces one function (which is named after the tag).

```
function print_elem(depth, element,   c, atn, chl, n, i, myChildren) {
  if (depth==1) {
    print "@load \"xml\""
    print "function NextElement() {"
    print "  while (getline > 0 && XMLERROR == \"\" && XMLSTARTELEM == XMLENDELEM)"
    print "    if (XMLCHARDATA) data = $0"
    print "  if (XMLERROR) {"
    print "    print \"error in row\", XMLROW \", col\", XMLCOL \":\", XMLERROR"
    print "    exit"
    print "  }"
    print "  return XMLSTARTELEM XMLENDELEM"
    print "}\n"
    print "BEGIN {"
    print "  if (NextElement() == \"" element "\") {"
    print "    parse_" element "()"
    print "  } else {"
    print "    print \"could not find root element '" element "'\""
    print "  }"
    print "}\n"
  }
  if (already_printed[element]++)
    return
  print "function parse_" element "() {"
  print "  # The '" element "' node has the following attributes:"
  # After the element name itself, list its attributes.
  for (a in attr) {
    split(a, atn, SUBSEP)
    # Treat only those attributes that belong to the current element.
    if (element == atn[1]) {
      # If an attribute occured each time with its element, notice this.
      if (attr[element, atn[2]] == elem[element])
        print indent "  # Mandatory attribute '" atn[2] "'"
      else
        print indent "  # Optional  attribute '" atn[2] "'"
    }
  }
  print ""
```

Figure 7.23: The first part of `print_elem()` in `parser_generator.awk`

This was the first part of `print_elem()`. The second part in Figure 7.24 produces the body of the function (see function `parse_book()` in Figure 7.21 for a generated example). In the body of the newly generated function we have a `while` loop which reads tokens until the currently read markup block ends with a closing tag. Meanwhile each embedded markup block will be detected and completely read by another function. Tags of embedded markup blocks will only be accepted when they belong to a set of expected tags. The rest

of the function should not be new to you, it descends recursively deeper into the tree of embedded markup blocks and generates one function for each kind of tag.

```
    print "  while (NextElement() && XMLENDELEM != \"" element "\") {"
    myChildren=""
    for (c in child) {
      split(c, chl, SUBSEP)
      if (element == chl[1]) {
        if (myChildren=="")
          myChildren = chl[2]
        else
          myChildren = myChildren " | " chl[2]
        print "    if (XMLSTARTELEM == \"" chl[2] "\") {"
        print "      parse_" chl[2] "()"
        printf "    } else "
      }
    }
    if (myChildren != "") {
      print " {"
      printf "      print \"unknown element '\" XMLSTARTELEM \"'"
      print " in '" element "' line \", XMLROW\n    }"
      print "  }"
    } else {
      # If an element has no child nodes, declare it as such.
      print "    # This node is a leaf."
      print "  }"
      print "  # The character data is now in \"data\"."
    }
    print "}\n"
    # Now go through the child nodes of this elements and print them.
    gsub(/[\|]/, " ", myChildren)
    n=split(myChildren, chl)
    for(i=1; i<=n; i++) {
      print_elem(depth+1, chl[i])
      split(myChildren, chl)
    }
  }
```

Figure 7.24: The second part of `print_elem()` in `parser_generator.awk`

When the complete parser is generated from the example file, you have a commented parser that serves well as a starting point for further refinements. Most importantly, you will add code for evaluation of the XML attributes and for printing results. Although this looks like an easy start into the parsing business, you should be aware of some limitations of this approach:

- **A tag name** in XML may be any valid Unicode name. Since this name is used as a function name in the generated AWK source code, you will run into problems when there is a tag name containing an Umlaut character for example: Umlaut characters are

not allowed in AWK function names. You have to avoid using tag names in generated code and use a mapping from original tag names into enumerated/generated names instead.

- **Roundtrip Engineering** is a problem for every code-generation framework. After generating a parser, what happens if I want to add new tags as they are used in different example files ? This is a hard problem. Perhaps the best solution is to change the process of code generation by inserting manual changes to the generated parser *not* into the generated parser itself but into a PI (processing instruction) in the example file. The parser generator must take this PI and copy its content into the generated code.
- **Semantic constraints** on XML data are much more easily coded in Schema languages. A more advanced approach to the parser generation problem might be to take an existing Schema specification and compile it into a parser. When the Schema language at hand is itself written in XML, this may look like an easy solution. But when you look at it, this approach is a real compiler construction job with many pitfalls.

7.8 A parser for Microsoft Excel's XML file format

The previous two sections about generating text files from an XML example file were rather abstract and might have confused you. This section will be different. Here, we will put the program `parser_generator.awk` to work and see what it's good for. We will generate a parser for the kind of XML output that Microsoft's Excel application produces. Our starting point will be an XML file that we have retrieved from the Internet.

Before we put the parser generator to work, let's repeat once more that the parser generator consists of the source code presented in Figure 7.19, Figure 7.23 and Figure 7.24. Put these three fragments into a file named `parser_generator.awk`.

Now is the time to look for an XML file produced by Microsoft Excel that will be used by the generator. The example file should contain all relevant structural elements and attributes. Only these will be recognized by the generated parser later. On the Internet I looked for an example file that contained as much valid elements and attributes as possible. I found several file which are freely available and could serve well as templates, but none of them contained all kinds of elements and attributes. Two of the most complete were the following ones. Invoke these commannds and you will find the two files in your current working directory:

```
wget http://ruby.fgcu.edu/courses/CGS1100/excel/PastaMidwest.xml
wget http://ruby.fgcu.edu/courses/CGS1100/excel/Oklahoma2004.xml
```

If you have some examples of your own, pass their names to the parser generator along with the others like this:

```
gawk -f parser_generator.awk PastaMidwest.xml Oklahoma2004.xml > ms_excel_parser.awk
```

Now you will find a new file `ms_excel_parser.awk` in your current working directory. This is the recursive descent parser, ready to parse and recognize all elements that were present in the template files above. To prove the point, we let the new parser work on the template files and check if these obey the rules:

```
gawk -f ms_excel_parser.awk PastaMidwest.xml
gawk -f ms_excel_parser.awk Oklahoma2004.xml
gawk -f ms_excel_parser.awk xmltv.xml
could not find root element 'Workbook'
```

Obviously, the file `xmltv.xml` from Section 6.2 [Convert XMLTV file to tabbed ASCII], page 44, was the only file that did not obey the rules, which is not surprising. Each XML file exported from Microsoft Excel has a node of type `Workbook` as its root node. These `Workbook` nodes are parsed by the program `ms_excel_parser.awk` right at the beginning in the following function:

```
BEGIN {
  if (NextElement() == "Workbook") {
    parse_Workbook()
  } else {
    print "could not find root element 'Workbook'"
  }
}

function parse_Workbook() {
  # The 'Workbook' node has the following attributes:
  # Mandatory attribute 'xmlns:html'
  # Mandatory attribute 'xmlns:x'
  # Mandatory attribute 'xmlns'
  # Mandatory attribute 'xmlns:o'
  # Mandatory attribute 'xmlns:ss'

  while (NextElement() && XMLENDELEM != "Workbook") {
    if (XMLSTARTELEM == "Styles") {
      parse_Styles()
    } else     if (XMLSTARTELEM == "Worksheet") {
      parse_Worksheet()
    } else     if (XMLSTARTELEM == "ExcelWorkbook") {
      parse_ExcelWorkbook()
    } else     if (XMLSTARTELEM == "OfficeDocumentSettings") {
      parse_OfficeDocumentSettings()
    } else     if (XMLSTARTELEM == "DocumentProperties") {
      parse_DocumentProperties()
    } else  {
      print "unknown element '" XMLSTARTELEM "' in 'Workbook' line ", XMLROW
    }
  }
}
```

Figure 7.25: A generated code fragment from `ms_excel_parser.awk`

If the root node is *not* a node of type `Workbook` (like it's the case with the file `xmltv.xml`), then a report about a missing root element is printed. As you can easily see, a `Workbook` has several mandatory attributes. The generated parser could be extended to also check

the presence of these. Furthermore, a `Workbook` is a sequence of nodes of type `Styles`, `Worksheet`, `ExcelWorkbook`, `OfficeDocumentSettings` or `DocumentProperties`.

8 Reference of XML features

This chapter is meant to be a reference. It lists features in a precise and comprehensive way without motivating their use. First comes a section listing all builtin variables and environment variables used by the `gawk-xml` extension. Then comes a section explaining the two different ways that these variables can be used. Finally, we have several sections explaining libraries which were built upon the `gawk-xml` extension.

8.1 XML features built into the gawk interpreter

This section presents all variables and functions which constitute the XML extension of GNU Awk. For each variable one XML example fragment explains which XML code causes the pattern to be set. After this event has passed, the variable contains the empty string. So you *cannot* rely on a variable retaining a value until later, when the same kind of events sets a different value. Since we are *not* reading lines (but XML events), the variable `$0` is usually *not* set to any text value but to the empty string. Setting `$0` is seen as a side effect in XML mode and mentioned as such in this reference.

8.1.1 XMLDECLARATION: integer indicates begin of document

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

If an XML document has a header (containing the XML declaration), then the header will always precede all other kind of data — even comments, character data and processing instructions. Therefore the `XMLDECLARATION` (if there is one at all), will always be the very first event to be read from the file. When it has occurred, the `XMLATTR` array will be populated with the index items `VERSION`, `ENCODING`, and `STANDALONE`.

```
# The very first event holds the version info.
XMLDECLARATION {
  version    = XMLATTR["VERSION"   ]
  encoding   = XMLATTR["ENCODING"  ]
  standalone = XMLATTR["STANDALONE"]
}
```

Each of the entries in the `XMLATTR` array only exists if the respective item existed in the XML data.

8.1.2 XMLMODE: integer for switching on XML processing

This integer variable will not be changed by the interpreter. Its initial value is 0. The user sets it (to a value other than 0) to indicate that each file opened afterwards will be read as an XML file. Setting the variable to 0 again will cause the interpreter to read subsequent files as ordinary text files again.

‘`XMLMODE`’ = 0: Disable XML parsing for the next file to be opened

‘`XMLMODE`’ = 1: Enable XML parsing for the next file to be opened

‘`XMLMODE`’ = -1: Enable XML parsing, and enable concatenated XML documents

It is allowed to have several files opened at the same time, some of them XML files and others text files. After opening a file in one mode or the other, it is not possible to go on

reading the *same* file in the other mode by changing the value of `XMLMODE`. Many users need to read XML files which have multiple root elements. Such XML files are (strictly speaking) not really well-formed: Well-formed XML documents have only one root element. Setting `XMLMODE` to -1 tells the interpreter to accept XML documents with more than one root element.

The use of the line "@load "xml"" sets XMLMODE to -1 as a side-effect. The use of the command line option "-l xml' does the same. So, most users prefer the latter methods instead of setting `XMLMODE` directly. Invoking the GNU Awk interpreter by means of the `xmlgawk` script has the same side-effect.

8.1.3 `XMLSTARTELEM`: string holds tag upon entering element

```
<book id="hello-world" lang="en">
...
</book>
```

Upon entering a markup block, the XML parser finds a tag (**book** in the example) and copies its name into the string `XMLSTARTELEM` variable. Whenever this variable is set, you can take its value and store the value in another variable, but you *cannot* access the tag name of the enclosing (or the included) markup blocks. As a side effect, the associative array `XMLATTR` and the variable `$0` are filled. The variable `$0` holds the names of the attributes in the order of occurrence in the XML data. Attribute names in `$0` are separated by space characters. The variables `$1` through `$NF` contain the individual attribute names in the same order.

8.1.4 `XMLATTR`: array holds attribute names and values

```
<book id="hello-world" lang="en">
...
</book>
```

This associative is always empty, except when `XMLSTARTELEM`, or `XMLDECLARATION`, or `XMLSTARTDOCT` is true. In all these cases, `XMLATTR` is used for passing the values of several named attributes (in the widest sense) to the user.

- Upon setting of `XMLSTARTELEM`, the array `XMLATTR` is filled with the names of the attributes of the currently parsed element. Each attribute name is inserted as an index into the array and the attribute's value as the value of the array at this index. Notice that the array is also empty when `XMLENDELEM` is set.
- Upon setting of `XMLDECLARATION`, the array is filled with variables named `VERSION`, `ENCODING`, and `STANDALONE`, reflecting the parameters of the XML header. Each of these variables is optional.
- Upon setting of `XMLSTARTDOCT`, the array is filled with variables named `PUBLIC`, `SYSTEM`, and `INTERNAL_SUBSET`, reflecting the parameters of the same name within the DTD. Each of these variables is optional.

In the example we have `XMLSTARTELEM` and `XMLATTR` set to

```
# Print all attribute names and values.
XMLSTARTELEM    = "book"
XMLATTR["id"  ] = "hello-world"
XMLATTR["lang"] = "en"
```

```
$0              = "id lang"
$1              = "id"
$2              = "lang"
NF              = 2
```

8.1.5 XMLENDELEM: string holds tag upon leaving element

```
<book id="hello-world" lang="en">
  <bookinfo>
    <title>Hello, world</title>
  </bookinfo>
  ...
</book>
```

Upon leaving a markup block, the XML parser finds a tag (**book** in the example) and copies its name into the string `XMLENDELEM` variable. This variable is not as useless at it may seem at first sight. An action triggered by the pattern `XMLENDELEM` is usually the right place to process the character data (here `Hello, world`) that was accumulated inside an XML element (here `title`). If the XML element `book` contains a list of nested elements `bookinfo`, then the pattern `XMLENDELEM == "book"` may trigger an action that processes the list of `bookinfo` data, which was collected while parsing the `book` element. The array `XMLATTR` is empty at this time instant.

8.1.6 XMLCHARDATA: string holds character data

```
<title>Warning</title>

<para>This is still under construction.</para>
```

Any textual data interspersed between the markup tags is called *character data*. Each occurrence of character data is indicated by setting `XMLCHARDATA`. The actual data is passed in `$0` and may contain any text that is coded in the currently used character encoding. There are possibly `0` bytes contained in the data. The length of the data in bytes may differ from the number of characters reported as `length($0)`, for example in Japanesse texts. The character data reported in `$0` need not be byte-by-byte identical to the original XML data (because of a potentially different encoding). Line breaks are often contained in character data, like it is the case in the example above. All consecutive character data in the XML document will be passed in `$0` in one turn. Thus, line breaks may be contained in `$0`.

```
# Collect character data and report it at end of tagged data block.
XMLCHARDATA            { data  =   $0 }
XMLENDELEM  == "title" { title = data }
XMLENDELEM  == "link"  { link  = data }
XMLENDELEM  == "item"  { print "title", title, "contains", item, "and", link }
```

8.1.7 XMLPROCINST: string holds processing instruction target

```
<? echo ("this is the simplest, an SGML processing instruction\n"); ?>
```

Processing instructions begin with `<?` and end with `?>`. The name immediately following the `<?` is the *target*. The rest of the processing instruction is application specific. The target is passed to the user in `XMLPROCINST` and the content of the processing instruction is passed in `$0`.

```
# Find out what kind of processing instruction this is.
switch (XMLPROCINST) {
  case "PHP":             print "PI contains PHP source:",        $0 ; break
  case "xml-stylesheet": print "PI contains stylesheet source:", $0 ; break
}
```

8.1.8 XMLCOMMENT: string holds comment

```
<!-- This is a comment -->
```

Comments in an XML document look the same as in HTML. Whenever one occurs, the XML parser sets XMLCOMMENT to 1 and passes the comment itself in $0.

```
# Report comments.
XMLCOMMENT { print "comment:", $0 }
```

8.1.9 XMLSTARTCDATA: integer indicates begin of CDATA

```
<script type="text/javascript">
<![CDATA[
... unescaped script content may contain any character like < and "...
]]>
</script>
```

Character data is not allowed to contain a < character because this character has a special meaning as a tag indicator. The same is true for four other characters. All five characters have to be escaped (<) when used in an XML document. The CDATA section in an XML document is a way to avoid the need for escaping. A CDATA section starts with <![CDATA[and ends with]]>. Everything inside a CDATA section is ignored by the XML parser, but the content is passed to the user.

Upon occurrence of a CDATA section, XMLSTARTCDATA is set to 1 and $0 holds the content of the CDATA section. Notice that a CDATA section cannot contain the string]]>, therefore, nested CDATA sections are not allowed.

8.1.10 XMLENDCDATA: integer indicates end of CDATA

Whenever the XMLENDCDATA is set, the CDATA section has ended and the XML parser starts parsing the data as XML data again. The closing]]> of the CDATA section is not passed to the user.

8.1.11 LANG: env variable holds default character encoding

The operating system's environment at run-time of the GNU Awk interpreter has an environment variable LANG, which is part of the locale mechanism of the operating system. Its value determines the character encoding used by the interpreter. This value is visible to the user as the initial value of the XMLCHARSET variable.

```
# Print the character encoding of the user's environment.
BEGIN { print "LANG =", XMLCHARSET }
```

Sometimes the value of the LANG variable at the shell level is not copied verbatim into the XMLCHARSET; the operating system may choose to resolve aliases.

8.1.12 XMLCHARSET: string holds current character set

```
<?xml version="1.0" encoding="x-sjis-cp932"?>
```

This string is initially set to the current character set of the interpreter's environment (`nl_langinfo(CODESET)`). Although it is initially set by the interpreter, this string is meant to be set by the user when he needs data to be converted to a different character encoding. The XML header above, for example, is delivered in a Japanese encoding, and it may be necessary to convert the data to UTF-8 for other applications to read it.

```
# Set the character encoding so that XML data will be converted.
BEGIN { XMLCHARSET = "utf-8" }
```

Later, when XML files are opened by the interpreter, all XML data will be converted to the character set whose name was set by the user in `XMLCHARSET`. Notice that changes to `XMLCHARSET` will not take effect immediately, but only on the subsequent opening of any file. Such changes will affect only the file opened with the changed `XMLCHARSET` and not files opened prior to the change.

8.1.13 XMLSTARTDOCT: root tag name indicates begin of DTD

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [ <!ELEMENT greeting (#PCDATA)> ]>
<greeting>Hello, world!</greeting>
```

Valid XML data includes a reference to the DTD against which it should be validated. The `XMLSTARTDOCT` variable indicates the beginning of a DTD reference section. Such DTDs are either embedded into the XML data (like in the example above), or they are actual references to DTD files (like in the example below). In both cases, the name of the root tag of the XML data is copied into the variable `XMLSTARTDOCT`.

```
<?xml version='1.0'?>
<!DOCTYPE ListOfNames SYSTEM "Names.dtd">
<ListOfNames lang="English">
```

The distinction between both cases can be made by looking at the `XMLATTR` array. Once more, the `XMLATTR` array is used for passing the values of several named attributes (in the widest sense) to the user. If the array has an index `INTERNAL_SUBSET`, then the DTD is embedded into the XML data. Otherwise, the optional entries `PUBLIC` and `SYSTEM` will report the system identifier or the public identifier of the referenced DTD.

```
# Find out if DTD exists, and if it is embedded or external.
XMLSTARTDOCT {
  root = XMLSTARTDOCT
  if ("INTERNAL_SUBSET" in XMLATTR) {
    ...
  } else {
    public_id = XMLATTR["PUBLIC"]
    system_id = XMLATTR["SYSTEM"]
  }
}
```

In both cases, the DTD itself is *not* parsed by the XML parser. But any embedded DTD text is passed as *unparsed data* in the variable `XMLUNPARSED`.

8.1.14 XMLENDDOCT: integer indicates end of DTD

Whenever the variable XMLENDDOCT is set, the DTD section has ended and the XML parser continues parsing the data as tagged XML data again. The closing]> of the DTD section is not passed to the user.

8.1.15 XMLUNPARSED: string holds unparsed characters

Very few parts of the XML data go unparsed by the XML parser. Any embedded DTD of an XML document will be detected and reported as such (by setting XMLSTARTDOCT), but the DTD content itself is reported as unparsed data in XMLUNPARSED.

8.1.16 XMLERROR: string holds textual error description

This string is always empty. It is only set when the XML parser finds an error in the XML data. The string XMLERROR contains a textual description of the error. The contents of this text is informal and not guaranteed to be the same on all platforms. Whenever XMLERROR is non-empty, the variables XMLROW and XMLCOL contain the location of the error in the XML data.

8.1.17 XMLROW: integer holds current row of parsed item

This integer always contains the number of the line which is currently parsed. Initially, it is set to 0. Upon opening the first line of an XML file, it is set to 1 and incremented with each line in the XML data. The incremental reading of lines is done by the XML parser. Therefore, the notion of a line here has nothing to do with the notion of a *record* in AWK. The content of XMLROW does not depend on the setting of RS.

8.1.18 XMLCOL: integer holds current column of parsed item

This integer always contains the number of the column in the current line which is currently parsed. Initially, it is set to 0. Upon opening the first line of an XML file, it is set to 1 and incremented with each character in the XML data. At the beginning of each line it is set to 1. The incremental reading of lines is done by the XML parser. Therefore, the notion of a line here has nothing to do with the notion of a *record* in AWK. The content of XMLCOL does not depend on the setting of FS.

8.1.19 XMLLEN: integer holds length of parsed item

This integer always contains the number of bytes of the item which is currently parsed. Initially, it is set to 0. The number of bytes refers to bytes in the XML data originally parsed. It is not the same as the number of characters. After the optional conversion to the character encoding determined by XMLCHARSET the length in XMLLEN may also be different from the converted length of the XML data item.

8.1.20 XMLDEPTH: integer holds nesting depth of elements

This integer always contains the nesting depth of the element which is currently parsed. Initially, upon opening an XML file, it is set to 0. Upon entering the first element of an XML file, it is set to 1 and incremented with each further element (which has not yet been completed). Upon complete parsing of an element, the variable is decremented.

8.1.21 `XMLPATH`: string holds nested tags of parsed elements

Upon starting the interpreter and opening an XML file, this string is empty. With each `XMLSTARTELEM`, the new tag name is appended to `XMLPATH` with a `"/"` character in front of the new tag name. The `"/"` character (encoded according to the XML file's character encoding) serves as a separator between both. With each `XMLENDELEM`, the old tag name (and the leading `"/"`) is chopped off `XMLPATH`. The user may change `XMLPATH`, but any change to `XMLPATH` will be overwritten with the next XML data read.

8.1.22 `XMLENDDOCUMENT`: integer indicates end of XML data

This integer is always 0. It is only set when the XML parser finds the end of XML data.

8.1.23 `XMLEVENT`: string holds name of event

This string always contains the name of the event that is currently being processed. Valid names are `DECLARATION STARTDOCT ENDDOCT PROCINST STARTELEM ENDELEM CHARDATA STARTCDATA ENDCDATA COMMENT UNPARSED ENDDOCUMENT` . The names are closely related to the variables of the same name, that have an `XML` prefix. Any names coming with this event are passed in `XMLNAME`, `$0`, and `XMLATTR`. For details about which variable carries which information, See Figure 8.2.

8.1.24 `XMLNAME`: string holds name assigned to XMLEVENT

The variable `XMLNAME` is used for passing data when the variable `XMLEVENT` contains the specific event. For details about which variable carries which information, See Figure 8.2.

8.2 gawk-xml Core Language Interface Summary

The builtin variables of the previous section were chosen so that they bear analogy to the XML parser Expat's API. Most builtin variables reflect a "handler" function of Expat's API. If you have ever worked with Expat, you will feel at home with `gawk-xml`. The only question you will have is *how are parameters passed ?* This section answers your question with a tabular overview of variable names and details on parameter passing.

To be precise, there are actually two tables in this chapter.

8.2.1 Verbose Interface - One dedicated predefined variable for each event class: XML*eventname*

In the first table, you will find variable names that can stand by itself as *patterns* in a program, triggering an action that handles the respective kind of event. The first column of the table contains the variable's name and the second column contains the variable's value when triggered. All parameters that you can receive from the XML parser are mentioned in the remaining columns.

Event variable	**Value**	**$0**	`XMLATTR` **index (when supplied)**	`XMLATTR` **value**

`XMLDECLARATION`	1	—	`"VERSION"`
			`"ENCODING"`
			`"STANDALONE"`
				`"1.0"`
				enc. name
				`"yes"/"no"`

`XMLSTARTDOCT`	root element name	—	`"PUBLIC"`
			`"SYSTEM"`
			`"INTERNAL_ SUBSET"`
				public Id
				system Id
				1

`XMLENDDOCT`	1	—	—	—
`XMLPROCINST`	PI name	PI content	—	—
`XMLSTARTELEM`	elem. name	Ordered list of attribute names	given name(s)	given value(s)
`XMLENDELEM`	elem. name	—	—	—
`XMLCHARDATA`	1	text data	—	—
`XMLSTARTCDATA`	1	—	—	—
`XMLENDCDATA`	1	—	—	—
`XMLCOMMENT`	1	omment text	—	—
`XMLUNPARSED`	1	text data	—	—
`XMLENDDOCUMENT`	1	—	—	—

Figure 8.1: Variables for passing XML data in the *verbose interface*

8.2.2 Concise Interface - Reduced set of variables shared by all events

Now for the second table of variables. Some people don't like to remember all the different names in the table above. They prefer to remember only a minimum of two variable names. While the first variable (`XMLEVENT`) contains the kind of event that happened (`STARTELEM` for example), the second one (`XMLNAME`) passes details about it (like the name of the element).

All events and their parameters are passed in this manner. But sometimes there is more than just one parameter to be passed, then we have to rely on `$0` and `XMLATTR`, just like it was already described in the first table.

`XMLEVENT` **value**	`XMLNAME` **value**	**$0**	`XMLATTR` **index (when supplied)**	`XMLATTR` **value**
`"DECLARATION"`	—	—	`"VERSION"`	`"1.0"`
			`"ENCODING"`	enc. name
			`"STANDALONE"`	`"yes"/"no"`
`"STARTDOCT"`	root element name	—	`"PUBLIC"`	public Id
			`"SYSTEM"`	system Id
			`"INTERNAL_SUBSET"`	1
`"ENDDOCT"`	—	—	—	—
`"PROCINST"`	PI name	PI content	—	—
`"STARTELEM"`	elem. name	Ordered list of attribute names	given name(s)	given value(s)
`"ENDELEM"`	elem. name	—	—	—
`"CHARDATA"`	—	text data	—	—
`"STARTCDATA"`	—	—	—	—
`"ENDCDATA"`	—	—	—	—
`"COMMENT"`	—	comment text	—	—
`"UNPARSED"`	—	text data	—	—
`"ENDDOCUMENT"`	—	—	—	—

Figure 8.2: Variables for passing XML data in the concise interface

8.3 xmllib

FIXME: This section has not been written yet.

8.4 xmlbase

NAME

xmlbase - add some basic functionality to gawk-xml.

USAGE

```
@include "xmlbase"

result = XmlEscape(str)
result = XmlEscapeQuote(str)

XmlWriteError(message)
XmlCheckError()
```

DESCRIPTION

The *xmlbase* awk library adds some basic facilities to the gawk-xml extension.

Automatic error reporting
: The *xmlbase* library contains a rule that automatically invokes **XmlCheckError()** at **ENDFILE**.

XmlEscape(str)
: Returns the string argument with the basic XML metacharacters (`<`, `>`, &) replaced by their predefined XML escape sequences.

XmlEscapeQuote(str)
: Returns the string argument with all the XML metacharacters (`<`, `>`, &, `"`, ') replaced by their predefined XML escape sequences.

XmlWriteError(message)
: Prints a formatted diagnostic message showing **FILENAME**, **XMLROW**, **XMLCOL**, **XMLLEN** and the string argument.

XmlCheckError()
: If either **XMLERROR** or **ERRNO** have a non-null value, invokes *XmlWriteError()* on it. XMLERROR takes precedence over ERRNO. If ERRNO is used then it is cleared, to avoid duplicated error reports.

NOTES

The *xmlbase* library automatically loads the **xml** gawk extension.

LIMITATIONS

The error reporting facility may not suit everybody.

8.5 xmlcopy

NAME

xmlcopy - add token reconstruction facilities to gawk-xml.

USAGE

```
@include "xmlcopy"

result = XmlToken()
XmlCopy()

XmlSetAttribute(name, value)
XmlIgnoreAttribute(name)
```

DESCRIPTION

The *xmlcopy* awk library adds the ability to reconstruct the current XML token. The token can be modified before reconstruction.

Token reconstruction

XmlToken()
: Returns an XML string that represents the current token. The token is reconstructed from the predefined variables **XMLEVENT**, **XMLNAME**, **XMLATTR** and **$0**. They can have the original current token values or user modified ones.

XmlCopy()
: Writes an XML string that represents the current token, as returned by *XmlToken()*.

Token modification

XmlSetAttribute(name, value)
: Adds or replaces the (*name*, *value*) entry in **XMLATTR**. Adds *name* to **$0** if not already in it.

XmlIgnoreAttribute(name)
: Removes *name* from **$0**, so *XmlToken()* will ignore it. Keeps **XMLATTR** unchanged..

NOTES

The *xmlcopy* library includes the *xmlbase* library. Its functionality is implicitly available.

LIMITATIONS

When an XML declaration is reconstructed, the advertised encoding may not match the actual encoding.

8.6 xmlsimple

NAME

xmlsimple - add facilities for writing simple one-line scripts with the gawk-xml extension, and also simplify writing more complex scripts.

USAGE

```
@include "xmlsimple"

parentpath = XmlParent(path)
test = XmlMatch(path)
scopepath = XmlMatchScope(path)
ancestorpath = XmlMatchAttr(path, name, value, mode)

XmlGrep()
```

DESCRIPTION

The *xmlsimple* library facilitates writing simple one-line scripts based on the gawk-xml extension. Also provides higher-level functions that simplify writing more complex scripts. It is an alternative to the *xmllib* library. A key difference is that **$0** is not changed, so *xmlsimple* is compatible with awk code that relies on the *gawk-xml* core interface.

Short token variable names

To shorten simple scripts, *xmlsimple* provides two-letter named variables that duplicate predefined token-related core variables:

XD Equivalent to XMLDECLARATION.

SD Equivalent to XMLSTARTDOCT.

ED Equivalent to XMLENDDOCT.

PI Equivalent to XMLPROCINST.

SE Equivalent to XMLSTARTELEM.

EE Equivalent to XMLENDELEM.

TX Equivalent to XMLCHARDATA.

SC Equivalent to XMLSTARTCDATA.

EC Equivalent to XMLENDCDATA.

CM Equivalent to XMLCOMMENT.

UP Equivalent to XMLUNPARSED.

EOI Equivalent to XMLENDDOCUMENT.

Collecting character data

Character data items between element tags are automatically collected in a single **CHARDATA** variable. This feature simplifies processing text data interspersed with comments, processing instructions or CDATA markup.

CHARDATA
: Available at every **XMLSTARTELEMENT** or **XMLENDELEMENT** token. Contains all the character data since the previous start- or end-element tag.

Whitespace handling

The **XMLTRIM** mode variable controls whether whitespace in the **CHARDATA** variable is automatically trimmed or not. Possible values are:

XMLTRIM = 0
: Keep all whitespace

XMLTRIM = 1 (default)
: Discard leading and trailing whitespace, and collapse contiguous whitespace characters into a single space char.

XMLTRIM = -1
: Just collapse contiguous whitespace characters into a single space char. Keeps the collapsed leading or trailing whitespace.

Record ancestors information

The **ATTR** array variable automatically keeps the attributes of every ancestor of the current element, and of the element itself.

ATTR[**path@attribute**]
: Contains the value of the specified *attribute* of the ancestor element at the given *path*.

Example

While processing a `/books/book/title` element, `ATTR["/books/book@on-loan"]` contains the name of the book loaner.

Path related functions

A fixed path is a slash delimited list of direct child elements (`/name/name/...`). A path expression accepts also an asterisk (`*`) to match any name, and a double slash (`//`) to represent a descendant at any level. An absolute path starts with a slash (path from the root element). A relative path without a leading slash can start at any level (path from some ancestor).

XmlParent(path)
: Returns the path of the parent element. I.e., the *path* argument without the last `/name` part. The path argument is optional. If not given the **XMLPATH** is used.

XmlMatch(path)
: Tests whether the current **XMLPATH** matches the *path* expression argument, anchored at the end.

XmlMatchScope(path)

Returns the **XMLPATH** prefix not matched by the matching *path* expression argument. Returns a null value if there is no match.

XmlMatchAttr(path, name, value, mode)

Returns the path of the innermost ancestor that matches the *path* argument and also has a *name* attribute with the given *value*. The mode argument is optional. If non-null then the value is handled as a regular expression instead of a fixed value.

Grep-like facilities

XmlGrep()

If invoked at the **XMLSTARTELEM** event, causes the whole element subtree to be copied to the output.

NOTES

The *xmlsimple* library includes both the *xmlbase* and *xmlcopy* libraries. Their functionality is implicitly available.

LIMITATIONS

The path related functions only operate on elements. Comments, processing instructions or CDATA sections are not taken into account.

XmlGrep() cannot be used to copy tokens outside the root element (XML prologue or epilogue).

8.7 xmltree

NAME

xmltree - DOM-like facilities for gawk-xml. Its status is **experimental**. May change in the future.

USAGE

```
@include "xmltree"

XmlPrintElementStart(index)
XmlPrintElementEnd(index)
XmlPrintNodeText(index)

XmlPrintNodeTree(index)

n = XmlGetNodes(rootnode, path, nodeset)
value = XmlGetValue(rootnode, path)
```

DESCRIPTION

The *xmltree* awk library adds DOM-like facilities to the *gawk-xml* extension.

Automatic storage of the element tree

The *xmlbase* library contains rules that automatically store the document's element tree in memory. The tree contains a node for each:

- Element
- Attribute
- Text content fragment

Each node in the tree can be referenced by an integer node index. The root element node has an index of 1. Nodes are stored in lexicographical order.

Processing the tree in the END clause

The stored tree is not fully available until the end of the input file. The intended way of using the tree is to put all the processing code in the **END** clause.

Printing tree fragments

XmlPrintElementStart(index)
Prints the element's start tag, including the attributes. The index argument must point to an element node.

XmlPrintElementEnd(index)
Prints the element's end tag. The index argument must point to an element node.

XmlPrintNodeText(index)
Prints the text content of the node. The index argument must point to an attribute or text fragment node.

Selecting tree fragments

The *xmltree* library provides an XPath-like facility for querying or navigating the document tree.

n = XmlGetNodes(rootnode, path, nodeset)
Populates de *nodeset* integer array argument with the indexes of the nodes selected from the starting *rootnode* by the given *path* pattern. Returns the number of selected nodes.

value = XmlGetValue(rootnode, path)
Returns the text content of the set of nodes selected from the starting *rootnode* by the given *path* pattern. The content depends on the node kind:

Attribute node
The content is the attribute value.

Text fragment node
The content is the text fragment.

Element node
Concatenates the content of the descendant element and text fragment nodes. Attributes are excluded from the result.

The path expression language

path A relative path from one node to one of its descendants is denoted by a sequence of slash separated labels. The label of a child element is the element name. The label of an attribute node is the attribute name prefixed by the "@" sign. The label of a text content node is the string "#text". The path from one node to itself is an empty path. Examples: `book/title`, `recipe/ingredient/@calories`, `book/author/#text`.

path pattern
A sequence of selection steps `selector!condition!selector!condition....` Each step is a pair of contiguous "!" delimited fields of the expression.

selector Regular expression that will be matched against relative paths between nodes.

condition Like selectors, and may also have a trailing "/?" prefixed value pattern, also given as a regular expression.

selection step
A selection step selects descendant-or-self nodes whose relative path matches the selector, and in turn have some descendant-or-self node whose relative path and text content match the condition.

Examples:

`book!` –> selects all books.

`book!author` –> selects all books that have an author.

`book!author/?Kipling` –> selects all books written by Kipling.

`book!@onloan` –> selects all books that are loaned.

`book!@onloan!title!` –> selects the titles of all books that are loaned.

NOTES

The *xmltree* library includes both the *xmlbase* and the *xmlwrite* libraries. Their functionality is implicitly available.

LIMITATIONS

Currently only one XML input document is supported. And the stored node tree should not be modified.

The selection facility can only be used for descendants of a root node. Selectors for ascendant or sibling nodes are not supported.

8.8 xmlwrite

NAME

xmlwrite - gawk facilities for writing XML fragments or whole documents.

USAGE

```
@include "xmlwrite"

xwopen(filename[, options])
xwclose()

xwdeclaration(version, encoding, standalone)
xwstartdoct(root, pubid, sysid)
xwenddoct()

xwprocinst(name, string)
xwcomment(comment)

xwstarttag(name)
xwattrib(name, value)
xwendtag(name)

xwtext(string)
xwstartcdata()
xwendcdata()

xwunparsed(string)

xwdoctype(root, pubid, sysid, declarations)
xwstyle(type, uri)
xwelement(name, content)
xwcdata(string)
xwcopy()
```

DESCRIPTION

The *xmlwrite* library facilitates writing a XML document serially, piece by piece. A whole XML document can be composed this way. The composed document may be indented if desired. *xmlwrite* takes care of some peculiarities of the XML standard, like metacharacters escaping, whitespace handling, markup indentation, etc.

Output file and mode

xwopen(filename[, options])
Initializes output to the given file. The optional argument is an array of named options:

options["INDENT"]
Indent step (-1 = no indent), default = 2.

options["QUOTE"]
Preferred quote character (', "), default = (").

xwclose() Closes the current opened output file.

XML prologue

xwdeclaration(version, encoding, standalone)
Writes an XML declaration (`<?xml ... ?>`). All the arguments are optional.

xwstartdoct(root, pubid, sysid)
Writes the starting part of a DOCTYPE declaration (`<!DOCTYPE ...`). All the arguments are optional.

Internal DOCTYPE declarations, if any, may be inserted by subsequent *xwunparsed()* calls.

xwenddoct()
Writes the closing mark of the DOCTYPE declaration (`]>`).

Processing Instructions and Comments

xwprocinst(name, string)
Prints a Processing Instruction with the given name and contents (`<?name string?>`).

xwcomment(comment)
Prints a XML comment (`<!--comment-->`).

Elements and attributes

xwstarttag(name)
Prints the opening mark of an element start tag (`<name ...>`).

xwattrib(name, value)
Prints an attribute markup fragment (`name="value"`). Must be invoked immediately after the *xwstartag()* call.

xwendtag(name)
Prints an element closing tag (`</name>`). If the element is empty, just closes its collapsed markup (`/>`).

Character data

xwtext(string)
Writes the escaped text. If it is invoked inside a CDATA section, the text is written unescaped.

xwstartcdata()
Writes the opening mark of a CDATA section (`<![CDATA[`).

xwendcdata()
Writes the closing mark of a CDATA section (`]]>`).

Unparsed markup

xwunparsed(string)
Writes a text fragment literally. Can be used to directly insert special markup fragments.

Higher level convenience functions

xwdoctype(root, pubid, sysid, declarations)
Writes a complete DOCTYPE declaration with a single call. All the arguments are optional.

xwstyle(type, uri)
Writes a stylesheet processing instruction (`<?xsl-stylesheet type="text/type" href="uri"?>`).

xwelement(name, content)
Writes a complete simple element markup with a single call. Attributes are not supported. Nested child elements are not supported.

xwcdata(string)
Writes a complete CDATA section with a single call.

Integration with the XML extension

If the *xmlwrite* library and the *gawk-xml* extension are used together, then it is possible to directly copy XML input markup.

xwcopy() Writes the markup fragment equivalent to the current XML input token. Should be used instead of the *XmlCopy()* function provided by the *xmlcopy* library.

NOTES

xmlwrite is a standalone library that can be used independently of the *gawk-xml* extension (except the *xwcopy*() function).

LIMITATIONS

Improper use of the provided functions may produce non-wellformed markup.

The whole output document must be written with the provided functions. Mixing *xml-write* calls and direct print commands may produce corrupted markup.

It is not possible to write several output documents concurrently.

9 Reference of Books and Links

9.1 Good Books

Here is a commented list of books for those who intend to learn more about XML and AWK.

- The very first book about AWK was *The AWK Programming Language* by Aho, Kernighan and Weinberger. More than 20 years after its initial appearance, this book is still a highly appreciated source of information and inspiration. Some readers refer to this book as **TAPL** (`https://www.amazon.com/AWK-Programming-Language-Alfred-Aho/dp/020107981X`).
- *TAPL* was an expensive book and it was tied to the original AWK implementation that came with AT&T's Unix. When SunOS became popular, AWK's reputation was damaged by the way SunOS implemented the AWK interpreter (`oawk` and `nawk`). So there was a need for an implementation with open source code, support by developers and an appropriate documentation. The GNU Awk implementation is maintained by Arnold Robbins, who also serves as the author of **EAP3**, the third edition of the GNU Awk manual (`http://www.oreilly.com/catalog/awkprog3/index.html`). This inexpensive book is published by O'Reilly, but it is also distributed with the source distribution of GNU Awk. Along with this up-to-date source of information comes a reference card. Unfortunately, the reference card is not part of O'Reilly's book. No other publication is so precise about the subtle differences between the POSIX standard, the original AWK and GNU Awk.
- The success of other scripting languages in the aftermath of AWK's success and the rising of early GNU/Linux led to the misconception that AWK was being "replaced" by other languages. While stylish new scripting languages come and go every 5 years, AWK will *not* disappear or be "replaced". AWK is mentioned in the Single UNIX Specification (`http://en.wikipedia.org/wiki/Single_UNIX_Specification`) as one of the mandatory utilities of a Unix operating system. Only one other scripting language besides AWK enjoys this status as a "canonical Unix scripting language": the Bourne Shell. Few readers will actually need it, but if you want to know the exact specification of AWK, read the specification of **POSIX AWK** (`http://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html`).
- If you are just beginning to recognize the utility of AWK as a portable scripting language, you should read **Wikipedia's AWK entry** (`http://en.wikipedia.org/wiki/Awk`) as an introduction, overview and dispatcher for further sources.
- Just like GNU Awk, XML is an offspring of standards that can be traced back to the 1970s. Some recommended reading has already been mentioned earlier (see Section 1.3 [Looking closer at the XML file], page 9). As a single source, we would recommend the O'Reilly book **XML in a Nutshell** (`http://www.oreilly.com/catalog/xmlnut3/`). Again, Wikipedia's XML entry (`http://en.wikipedia.org/wiki/Xml`) provides a quick overview, introduction and collection of links and sources.

9.2 Links to the Internet

This section lists the URLs for various items discussed throughout this book.

- `gawk-xml` is based on Arnold Robbins' **GNU Awk** (`http://www.gnu.org/software/gawk/gawk.html`). Arnold's distribution is the one true GNU Awk. GNU Awk is meant to be a stable distribution, adhering to standards and used with many many Unix operating systems.
- `gawk-xml` is a functional extension of GNU Awk. It is distributed as part of the **gawkextlib project** (`http://sourceforge.net/projects/gawkextlib/`). The SourceForge project was initially named *xmlgawk* because original plans aimed at implementing an XML extension only. Later, the name of the distribution was changed to *xgawk* so that it reflected the emphasis on the e**x**tension, rather than on the XML parser. And finally the project was renamed as *gawkextlib* to make clear that it is a library of independent extensions.
- In Section 3.2 [Printing an outline of an XML file], page 20, we already mentioned that `gawk-xml`'s way of handling XML data was designed with the SAX API in mind. Other languages also have libraries implementing this API. To most developers, the Java implementation is the canonical implementation of the SAX API. Have a look at the chapter **XML Processing with Java** (`http://www.corewebprogramming.com/PDF/ch23.pdf`) in the book *Core Web Programming*. You will find an implementation of the script `outline.awk` that we presented in Figure 3.2. Comparing both implementations will reveal the strengths and weaknesses of a script solution and a compiled solution.
- The **XML standard's exact specification** (`http://www.w3.org/TR/2004/REC-xml-20040204/`) is open to everyone. But it is incomprehensible for casual users. A bit easier to consume and to digest is the *Introduction to the Annotated XML Specification* (`http://www.xml.com/axml/testaxml.htm`). Beginners are often struggling to gain an overview of all the optional and mandatory parts of an XML file. They will appreciate the *XML Syntax Quick Reference* (`http://www.mulberrytech.com/quickref/XMLquickref.pdf`). This two-page reference card was layed out in a nice graphical form and is noteworthy for presenting an overview which does *not* ignore the proper place of the DTD in the XML data.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

A

ASCII, tabbed 44
Attribute 9

B

Barnes & Noble Price Quote 56

C

`CDATA`, unescaped Character Data 84
Character Data 10, 23
character encoding 10, 23, 24, 52, 83, 84
character set 24
`chmod` 5
comp.lang.awk, newsgroup on the Internet 43
comp.text.xml, newsgroup on the Internet .. 10, 43
Cygwin 5, 48

D

database 62
`db_version.awk` 26
`demo_pusher.awk` 28
DocBook 6, 23, 27, 67, 70, 73
DOM, Document Object Model 1, 41, 50
`dot` 67
`dtd_generator.awk` 71
DTD, Document Type Definition ... 10, 19, 26, 48, 70, 85, 104

E

Element 10
`Expat, XML parser with SAX-like API` .. 1, 15, 20, 22, 88
`extract_characters.awk` 23

F

FDL, GNU Free Documentation License 105

G

gawkextlib 104
`get_rss_feed.awk` 55
`getXML` 15
`getXMLEVENT.awk` 17
Graphviz, open source graph drawing software 67

H

HTML 53
HTTP 53, 56

J

Java 104

L

`LANG`, environment variable 25, 84

M

`max_depth.awk` 8
Microsoft Excel 78
Microsoft Windows 1, 5, 25
Mixed Content 10
`modify.awk` 52
`modify.xml` 53

N

`nawk` 15, 103
`node_count.awk` 8

O

`outline, Expat application` 20
`outline.awk` 20, 104
`outline_dot.awk` 69
`outline_puller.awk` 21

P

parser, recursive descent 73, 78
POSIX 11, 103
`PostgreSQL` 62
PostScript 67, 69
Processing Instruction 10, 83

R

recursion 7, 47, 73, 77
`remote_data.xml` 51
RSS, Really Simple Syndication or Rich Site Summary 53

S

sample.xml 64
SAX, Simple API for XML 20, 104
Schema 10, 19, 57, 73, 78
soap_book_price_quote.awk 59
SOAP, Simple Object Access Protocol 56
SOAPscope 56
Solaris 15
SourceForge 104
SQL 62

T

Tag 9
target, of Processing Instruction 83
testxml2pgsql.awk 63
The Inquirer 54

U

Unicode 11, 24, 25, 77
Unix 5, 103
US-ASCII 25
UTF-16 .... 24, 25
UTF-8 .... 10, 24, 85

V

Valid 10
validation 19, 70

W

wc 5
wc.awk 5
Well-Formed 10, 82
well_formed.awk 19

X

xgawk 104
XML technology 10
XMLATTR 20, 21, 22, 26, 28, 44, 46, 47, 65, 69, 71, 81, 82, 85
XMLATTR["ENCODING"] 25, 26, 28, 52, 81
XMLATTR["INTERNAL_SUBSET"] 26, 28
XMLATTR["PUBLIC"] 26, 28, 85
XMLATTR["STANDALONE"] 26, 28, 81
XMLATTR["SYSTEM"] 26, 28, 85
XMLATTR["VERSION"] 26, 28, 81
XMLBooster 73
XMLCHARDATA 28, 83
XMLCHARSET 23, 25, 28, 84, 85, 86
XMLCOL 28, 86
XMLCOMMENT 28, 84
XMLDECLARATION 25, 28, 81, 82
XMLDEPTH 28, 69, 71, 86
XMLENDCDATA 28, 84
XMLENDDOCT 28, 86
XMLENDDOCUMENT 28, 87
XMLENDELEM 8, 28, 82, 83
XMLERROR 19, 28, 86
XMLEVENT 22, 28, 65, 87
xmlgawk 104
XMLLEN 28, 86
xmllint 19
XMLMODE 28, 81
XMLNAME 28, 87
xmlparse.awk 11
XMLPATH 28, 87
XMLPROCINST 28, 83
XMLROW 28, 86
XMLSTARTCDATA 28, 84
XMLSTARTDOCT 28, 82, 85, 86
XMLSTARTELEM 8, 28, 82
XMLTV 44
XMLUNPARSED 28, 85, 86
XSL 5, 41, 44, 45, 47, 50
xsv 19

Y

Yahoo 56