

School of Computer Science and Software Engineering
Monash University

Bachelor of Computer Science Honours (1608), Clayton Campus

Structured process input/output

Reformulating the UNIX command line environment to generate and process XML documents

Cameron McCormack (12793086)

clm@csse.monash.edu.au

Supervisor: Prof. John Hurst

ajh@csse.monash.edu.au

Declaration of Originality

I, Cameron Leigh McCormack, declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished works of others has been acknowledged in the text and a list of references is given in the bibliography.

Cameron Leigh McCormack
November 5, 2002

Abstract

One of the main tenets of the philosophy of the UNIX operating system is that is more effective to write small, simple programs that do specific tasks very well, as opposed to having one integrated program that does it all. Users compose these small programs at the command line, by way of a pipeline, to perform more complex computations. These simple programs have typically focussed on processing flat text files. Often, programs generate output that was destined to be displayed to the user and as such, the data is formatted for display. This necessitates the use of text processing programs to filter out irrelevant information and to select the appropriate data. However, this imposes an unwanted dependency on the formatting of the original output. It is clear that a separation of form and content must be made. This thesis explores the idea of using structured data for the input and output of some standard UNIX utilities. In particular, versions of these programs that parse XML input and generate an XML document as their output will be presented. An environment in which to execute these programs and format their output for display is also given.

Table of contents

1. Introduction	6
1.1. Previous research	6
1.2. Project outline	6
2. Existing systems	8
2.1. UNIX command line environment	8
2.1.1. User programs	8
2.1.2. Input/output redirection	8
2.1.3. Command pipelines	10
2.1.4. The shell	10
2.2. Problems	11
3. Structured input/output	13
3.1. Extensible Markup Language	14
4. Data generating programs	16
4.1. Generating XML output	16
4.2. Important programs	16
4.3. Implementation examples	17
4.3.1. ls	17
4.3.2. ps	20
4.3.3. df	21
5. Data filtering programs	22
5.1. Transforming the input	22
5.2. Important filters	23
5.3. Implementation examples	23
5.3.1. cat	23
5.3.2. grep	25
5.3.3. sort	27
6. User interface to the environment	29
6.1. The shell	29

6.2. Human readable output	29
7. Evaluating the model	31
7.1. Common applications	31
7.1.1. Determine filesystem with most free space	31
7.1.2. Determine which process has been running the longest	32
7.2. Issues with the model	33
8. Conclusion	34
8.1. Future work	34
Bibliography	35
Appendix A. Code	37
A.1. ls.pl	37
A.2. ps.pl	40
A.3. cat.cpp	41
A.4. sort.cpp	45

1. Introduction

The UNIX operating system (Ritchie, Thompson 1974) is an interactive time-sharing system developed in 1963 while its authors were working for Bell Laboratories. It was created with the aim of being an operating system for programmers. Since its inception, the UNIX command line environment has focussed on processing flat text files.

Flat text files are files whose data are organised into records and fields. Each record in the file is stored on one line. Each field within that record is separated from the other fields by a variable amount of horizontal whitespace. None of the UNIX systems developed over the last 30 years have moved away from the flat text file format that the UNIX command line holds to firmly. Gancarz (1995) notes that data files should contain only streams of bytes separated by newline characters. Authors of UNIX systems have thus far adhered to this.

The reason for the success of the flat text file format is its simplicity. It is indeed an easy to use, useful format in which to store some types of data. Thirty years, however, is a very long time in the world of computers, and requirements for data storage have changed.

Of interest is the possibility of reworking the UNIX command line environment to use some other, more structured file format instead of flat text. If programs generate and process a more user friendly file format, there is the possibility that commands constructed at the shell are easier to devise and understand.

1.1. Previous research

Even since the early days of UNIX, little research has been done into reworking the command line and even less into the idea that leaving behind the flat text file paradigm would be prudent. This is because keeping data stored in such a simple format has proven simple and powerful. Not all data fits neatly into the record/field organisation of the flat text file, however. It is for this reason that a more structured format for data files and process input and output, such as the Extensible Markup Language (XML) (Bray, et al 2000), needs to be investigated.

1.2. Project outline

The first stage of this project involved looking at existing UNIX systems' command line environments and determining the problems they have. This review of traditional UNIX data processing is given in section 2. Following this, a new model

was developed for process input and output, which is presented in section 3. An overview of XML is also given here.

The programs developed for the new command line environment are divided into three categories — data generating programs, data filtering programs and the interface to the environment. These are discussed in sections 4, 5 and 6, respectively.

Finally, in section 7, some example applications of the command line environment are given, comparing the techniques used in a traditional UNIX environment and those used in the new model.

2. Existing systems

2.1. UNIX command line environment

The command line environment of UNIX has four main aspects: user programs, input/output redirection, command pipelines and the shell. Together, these provide the user with a powerful means of composing and executing commands to manipulate data files.

2.1.1. User programs

The selection of programs available for the user to run is at the heart of the command line environment. From the outset, the focus of user programs was on file manipulation. The first user programs to be written for UNIX were simple file system commands. After having just implemented the file system, Thompson (1979) wrote a few small programs to manage it — programs to let the user create, copy, edit, list and delete files. These commands alone gave the user a simplistic electronic word processor, but not much more. As more user programs were added to the system, three general categories of program emerged: commands which performed a task with no output (such as copying or deleting a file), those which produced output (such as the directory listing program) and those which transformed output (filters, like the program to sort a file's contents).

The user programs that were developed for UNIX were all small programs that performed a single task, and only that task, well. Kernighan and Plauger (1976) expounded on that idea and coined the term *software tool*. By writing programs to deal with generalities rather than specifics, focussing them on performing a single task well, the user can build up a toolkit from which they can assemble commands to perform complex tasks.

These programs were all written with a certain data format in mind, namely, flat text. It was held that, to maximise portability and usefulness, data should be stored in plain ASCII (the American Standard Code for Information Interchange — a 7-bit code defining a set of alphabetic, numeric, punctuation and control characters), arranged into fields separated by horizontal whitespace and records separated by newline characters (Gancarz 1995).

2.1.2. Input/output redirection

One idea which increases the usefulness of the user programs in UNIX is that of input/output (I/O) redirection. General redirection of process input and output was

already available in Multics (Corbató, Vyssotsky 1965) and it was from here that the idea was borrowed. One of the tenets of UNIX philosophy that Gancarz (1995) mentions is that all programs should be written as filters. Processes under UNIX have the notion of a *standard input* and *standard output*. If a program has not been given a filename on the command line, the standard input is where a program typically reads its input from. The standard output, analogously, is where a program would write its output to. Any program that reads data from the standard input, transforms it somehow, then writes that data out to the standard output, is considered to be a filter.

Normally, both the standard input and output of a process is the terminal. Thus, program input comes from the keyboard and program output is written to the screen. In UNIX, both the standard input and output of a process can be redirected to some other file or device. With this, a program can be written to handle just a single input source (the standard input) and be able to handle any file implicitly, by having its input redirected before the program is run; similarly for a program's output. This redirection is performed by the *shell*, the command interpreter that elicits commands to run from the user. Programs which did not use the standard input and output, and instead relied on hard coded filenames, were definitely of limited use.

While Multics could also handle such redirection, the method used for enacting the redirection in UNIX was far more elegant. According to Ritchie (1979), Multics required the user to use an explicit command to reassign a program's output. For example, to run the **list** command and redirect its output to a file called **listing**, the following three commands were issued:

```
iocall attach user_output file listing
list
iocall attach user_output syn user_i/o
```

To accomplish the same task in UNIX, running the **ls** program to generate the listing, just a single command was needed:

```
ls > listing
```

Input redirection could be achieved with a similarly simple command:

```
ed < cmds
```

This command runs the **ed** editor, taking commands from the file **cmds**. The syntax of I/O redirection, using '**<**' and '**>**', has not changed since its original inclusion in UNIX in 1970.

2.1.3. Command pipelines

The mechanism for composing commands is the *command pipeline*. The concept, which was first seen in the UNIX operating system, is attributed to McIlroy (Ritchie 1979). In 1972 McIlroy proposed the concept of connecting a series of processes together such that the output of one process became the input of the next, the analogy used being that connecting these processes was like screwing together garden hoses. The command pipeline is what enables the reuse of the software tools available to the UNIX user to create complex commands in a single line.

The original syntax used for the pipe concept was similar to the I/O redirection syntax that already existed. A pipeline to sort a file and take the first 10 lines to send to the printer could be written as:

```
sort file >head>lpr
```

Thus, either a file or a command could be placed after a '>' character to indicate where the output of the previous command in the pipeline was to go. This syntax, while consistent with the redirection syntax, did allow ambiguous commands to be written. For example, with this syntax there was no way to specify that you wished the output of the **sort** command to be written to a file called **head**, as the shell would determine that **head** was a program and that **sort**'s output should be sent there instead. The syntax was therefore changed to use the vertical bar (or the *pipe symbol* as it has become known). The above command now could be written as:

```
sort file | head | lpr
```

It was at this time, after having implemented the pipeline concept, that the developers of UNIX came to the realisation that this simple extension of I/O redirection was the vehicle for program reuse (Kernighan, Plauger 1976).

2.1.4. The shell

The program that provides an interface to the command line environment is the shell. It is this program that repeatedly prompts the user for a command and then executes that command. The shell is perhaps the one aspect of the UNIX command line environment that has undergone a number of changes since its inception.

The original UNIX shell created at the start of 1970 was very simplistic (Ritchie 1979). All the shell did was read a single command from the terminal, load the specified program over the top of itself, and jump to that new program. The operating system relied on the user program to invoke the *exit* system call, which would load a fresh copy of the shell into memory and begin its execution. This was not a true multiprogrammed system, though. Only one process could run at one time.

Subsequent developments to the operating system included the *fork/exec* process control that still exists in today's UNIXes. The Berkeley time-sharing system developed in 1969 already incorporated this method of process control. With this, the UNIX shell could now handle background processes easily. Background processes are those started with a command such as:

```
program &
```

The ampersand indicates that the shell should execute the program in the background and immediately ask the user for another command to run.

From the release of UNIX First Edition in 1971 until the Sixth Edition in 1976, very little changed in the operation of the shell. However, during the development of the Seventh Edition, Bourne (1983) created his own shell, to be known as the Bourne shell. The Bourne shell brought to UNIX shell programming constructs such as iteration and selection that allowed the user to write complex shell scripts. From that point onward, most new shells developed for UNIX, such as the Kornshell (Bolsky, Korn 1995) and the Z shell (Falstad 2001), maintained a syntax compatible with the Bourne shell. One popular shell that was significantly different from the Bourne shell was the C shell, *csh*, written by Bill Joy for inclusion into the Berkeley Systems Distribution of UNIX (McKusick 1999). The C shell, as its name implies, lets the user run commands with a syntax reminiscent of the C programming language (Kernighan, Ritchie 1978).

Attempts to vastly redesign the UNIX shell have been made. One notable design is that of a functional shell, where commands are entered in a form similar to those used in a functional programming language. These shells, such as McDonald's (1987) *fsh* and the Scheme shell (Shivers, et al 2002), *scsh*, allow the user to compose commands as if they were functions.

2.2. Problems

While certainly an appropriate design decision for the early 1970s, the 7-bit ASCII flat text file format used by the UNIX user programs is not always the best today. ASCII is able to represent only a handful of Western, Latin-based languages. There was not much cause for writing with internationalisation and localisation in mind when the standard UNIX user programs were being created. Today, with global networks such as the Internet, languages other than English have to be considered. A character set such as Unicode (The Unicode Consortium 2000) or the Universal Character Set (UCS) (ISO 10646-1:2000) would certainly solve this problem if they were routinely used and had support from the operating system and its user programs.

The rigid record/field format of flat text files also does not accommodate all types of

data. A particular example would be information arranged hierarchically. Naturally such information could be stored in flat text, however an inelegant kludge such as recording a node's depth in the tree would have to be stored in its own field. Far better to keep the relationships between the hierarchical data evident due to the file format itself. A data file format such as XML could quite easily be used to store such data, being intrinsically hierarchical. XML also has the advantage of using named fields. A user reading an XML document could easily determine the use of certain fields by their names, whereas in a flat text file, such names are absent, and the user would have to rely on some external documentation. To be maximally useful all text generating programs would have to write their output in XML so they could be used in conjunction with XML filtering programs.

All of the shells mentioned earlier, which have been developed for UNIX, have been designed with the assumption that the majority of user programs that are to be composed use flat text for their input and output. The shells can let program output be written verbatim to the terminal, since flat text is eminently human readable. If, however, user programs are to be designed to read and write XML documents, program output is not immediately appropriate for the user to read. XML documents are readable by humans — they do not contain any illegible control codes as some binary formats do, but the information in these documents should be converted into a form that is easy for the user to read. Since the user will be interacting with the shell using a text based terminal, program output destined for the terminal should be transformed into plain text by the shell. Making this final transformation for the user the responsibility of the shell means that user programs do not have to worry about checking whether their output is to be piped into another program, redirected into a file or written to the terminal.

3. Structured input/output

To address the problems mentioned in the previous section, a new model for the UNIX command line environment was devised. This model describes the general format programs should accept on their standard input and produce on their standard output, as well as the techniques that are to be used for handling the presentation of this data.

In the new model, programs will use a more structured data format for their input and output. Specifically, they will read and write Extensible Markup Language (XML) documents. There are a number of advantages to using a markup language such as XML instead of flat text. The major superiority is the ease of parsing to extract particular information. Formatting output for display on the terminal is sometimes an information losing process. It is this formatting for display which often makes it difficult for the user to construct a command to select the relevant fields or records. If a program generates its output in XML, human factors such as "information overload" do not have to be considered. All of the information can be kept in the output document in a form that is easy to extract.

XML documents also are inherently self documenting. Records and fields (elements and attributes in the document, correspondingly) must all be named explicitly. This helps the user inspecting the document determine the nature of the contents of each field. Comments can also be placed in the XML documents without causing disruption to programs that parse them.

As the name suggests, XML documents are extensible. What this means is that a user or a program can annotate a document by adding extra attributes to an element, again without confusing existing programs which parse the document.

All conforming XML parsers also must be capable of reading in documents using the Unicode character set. Unicode, which encompasses a wide variety of scripts and symbols used around the world, allows a single document to include data in many languages using only a single encoding. A user can include, say, text written in both Traditional Chinese and in Hebrew, without worrying that the programs manipulating such documents could not handle them. Of course, the display of such languages is another issue.

Since programs which generate output now must not worry about formatting, this formatting must be deferred somehow. The formatting should be delayed for as long as possible, so that the original information is always available to be parsed. It makes sense, then, that formatting should take place just before the output is to be written to

the terminal for the user to read. Thus this model requires that the shell transform the XML document resulting from the last program in a pipeline into a form suitable for writing to the terminal.

3.1. Extensible Markup Language

XML documents are essentially data organised into a tree structure. The syntax of XML is reminiscent of the Standard Generalised Markup Language (SGML) (ISO 8879:1986), being derived from it. XML is basically a stricter form of SGML with a couple of additions.

Every well formed XML document is a rooted tree of elements. An element can contain other elements, as well as attributes and character data. A very simple XML document is shown below:

```
<datafile>
  <record field1="val1" field2="val2"/>
  <record field1="val1" field2="val2">
    <details>Some text</details>
  </record>
</datafile>
```

The **datafile** element is the root element of this document. It opens with the start tag **<datafile>** and closes with the end tag **</datafile>**. Note that the first **record** field in the document does not have an end tag. Since the element has no content, it is opened and closed in the one tag, by having the slash just before the closing angle bracket.

Every element can have attributes. These can be seen in the example document as **field1** and **field2** in the **record** elements. Every attribute must have a value, surrounded by either double or single quotes. Each element can also have character data as its content. In the example, the **details** element has the text "Some text" as its content.

Every element and attribute is also within a particular namespace. Namespaces are used in XML documents to help prevent name collisions, and also to designate the type of an element or document. A namespace is a Uniform Resource Identifier (Berners-Lee 1994). Each element in an XML document can be given a namespace, such that every element within that element (unless overridden by another namespace) will be in that namespace. This is achieved by using the special **xmlns** attribute on an element, as demonstrated below:

```
<courses xmlns="http://myuni.edu/">
  <subject>
    ...
  </subject>
```

```
</courses>
```

Now both the **courses** and **subject** elements are identified as coming from the namespace with URI **http://myuni.edu/**. The URI is not required to refer to a physical web page. It is simply used as a unique identifier. Instead of assigning a namespace to the **courses** element and all its descendents, the namespace can be given a prefix and used explicitly.

```
<myu:courses xmlns:myu="http://myuni.edu/">
  <myu:subject>
    ...
  </myu:subject>
</myu:courses>
```

The **myu** prefix is used to identify the namespace. Namespace prefixes are often used when more than one namespace is needed in the one document. Elements which do not have an associated namespace are said to be within the default namespace.

The Extensible Stylesheet Language Transformations (XSLT) specification (Clark 1999) was developed along side XML to be used as a means of specifying how XML documents can be translated into other XML documents or into plain text. XSLT stylesheets are XML documents themselves, and provide the user with a means of specifying declaratively how particular elements of an XML document are to be transformed.

A language to address parts of an XML document was developed for use with XSLT, called XPath (Clark, DeRose 1999). XPath locations look similar to UNIX pathnames in that components are separated by a slash. A component in an XPath location can be an element name, an attribute name preceded by an @ symbol or a predicate such as **text()** to refer to a text node. For example, the XPath location **/courses/subject** refers to all of the **subject** elements which are directly beneath the **courses** element at the top of the document. Square brackets can be used to specify a constraint on selecting particular nodes. So, **/courses/subject[@code='CSE1301']/@name** will select the name of the subject whose code attribute contains the string 'CSE1301'. Quite complex expressions can be built up with XPath to refer to parts of an XML document.

4. Data generating programs

The first category of programs in the new model for the UNIX command line environment is the data generating programs. These programs do not take any input, though they may have command line arguments. Whereas in the traditional UNIX command line environment such programs may format their output for human readability, in the new model these data generating programs must not. Instead, the output must be a well formed XML document.

4.1. Generating XML output

So that the output can be identified as having been generated by a particular program, each data generating program will produce an XML document with a namespace specific for that program. This namespace will be used to determine how to transform the output to a human readable form later.

In general, the structure of the output document will be the same. Namely, there will be an element for each record underneath the root element. The data filtering programs described later default to processing documents structured in this manner. Of course, not all documents generated will have this format. The filters have options to inspect different parts of the document to handle these cases.

Users are accustomed to specifying formatting options to data generating programs. For example, they will give the `-l` switch to the `ls` program to instruct it to give a long listing of the files. Since the new model requires the data generating programs not to act on these formatting flags, they should store this formatting preference as part of the resulting output document. Adding an attribute to the root element of the XML document is sufficient for recording the formatting option. Later on, when the document is to be transformed for output to the terminal, this attribute can be checked for and acted upon.

4.2. Important programs

Inspecting a typical UNIX system, the programs in the table below were identified as being important data generating programs which should be included in the new environment.

Program	Purpose
date	Reports the current date and time
df	Reports the disk space free on each mounted

	filesystem
du	Report disk usage
echo	Outputs some text
hostname	Reports the current host's name
ls	Lists file details
netstat	Reports currently open network connections
ps	Lists details of processes running on the system
uname	Reports operating system details

4.3. Implementation examples

Three examples are now given for new, XML outputting versions of the UNIX data generating processes.

4.3.1. ls

The traditional **ls** program takes a list of files on the command line and displays information about those files. If a file on the command line is a directory, information about the files inside that directory are given instead. If no files are given on the command line, just information about the files in the current directory is given.

Without any formatting options, **ls** outputs just a list of the filenames. If the output is destined for a terminal, these files would typically be lined up in columns to save space on the screen. If the output is going to a file or another process' standard input, the filenames would be listed one per line. Such output is good for parsing, as there is no real formatting going on here.

One of the most commonly used options, however, does force some formatting to be done. The **-l** option causes **ls** to output a long listing with details about each file, such as its permissions, size and last modified date. This information is arranged into fixed width columns, regardless of whether the output is destined for the terminal or another process' standard input. An example of this output is shown below:

```
total 8
-rw-r--r--  1 cameron cameron  253 Oct 22 23:40 phonebook
-rw-r--r--  1 cameron cameron  763 Sep  5  2001 resume
```

The first thing output is the number of disk blocks taken up by the files which are listed. Then comes a line for each file, detailing the permissions in symbolic form, number of hard links, owner, group, size, last modified time and name of the file. One particularly nasty problem with this output is that the custom for using

whitespace to separate fields has been broken. The last modified time actually has spaces embedded in the field. "Oct 22 23:40" should be just the one field of this record. This means that it is somewhat difficult to extract this information from the record. One way to do it would be to use the **cut** program, which can select the text from each record between two given columns. Obviously this is going to need the user to count the characters in this output to determine where the last modified time begins and where it ends. Really the user would prefer to issue a command to just "extract the last modified time" rather than "extract characters in columns 40 through 52".

It can also be seen that the **ls** program changes the format of the last modified date depending on the actual date. If the last modified date is within about ± 6 months of the current date, the month, day of month and time will be displayed. But if the last modified date falls outside this range, the month, day of month and year will be displayed instead. This is perhaps not such a bad idea when considering what the user wants to see. If a document has been changed over a year ago, it is more important to mention the year of modification than the time of day. For extract information, however, this is not so good. Ideally, the full time would be listed for every file, so that no information is lost.

The **ls** program also has another mode, enabled by using the **-R** switch. This causes **ls** to recurse through directories to list information about files in a whole directory tree. An example of the traditional **ls -R** output is listed below:

```
. :
documents

./documents:
personal
shared

./documents/personal:
phonebook
resume

./documents/shared:
housekeeping
```

The output for the recursive directory listing is split into sections. Each section is for one directory, and it starts with a header comprising the name of the directory followed by a colon. Each file in the directory is then listed after this header. A blank line separates each directory. This output format breaks the customary flat text format even more than the previous example did. Now instead of there being a file per line, any program parsing this output must be careful of the headers and the blank lines. For a human, this output format is fine, as it is a reasonable way to flatten the hierarchy of the directory tree for display on the terminal. The inherent

structure in the directory tree has been lost, though, and it is non-trivial to extract information about a file's position in the directory tree.

To overcome these shortcomings in the traditional `ls` program, the new `ls` will produce an XML document with an element for each file being listed. Each `file` element in the output will have a number of attributes, used to hold the details of that file. Also, each `file` element can contain other `file` elements, thereby preserving the structure of the directory tree.

Here is the above `ls` example with the new output format (some attributes have been elided for brevity):

```
<files xmlns="urn:ns:clm.xml-unix.ls" rec="true">
  <file name="documents" path="." uid="1000" gid="1000"
    size="4096" type="directory" mode="0755"
    mtime="914037321" human-mtime="Dec 19 1998">
    <file name="personal" path="documents" uid="1000" gid="1000"
      size="4096" type="directory" mode="0755" mtime="1022074845"
      human-mtime="May 22 23:40">
      <file name="phonebook" path="documents/personal" uid="1000"
        gid="1000" size="253" type="regular" mode="0644"
        mtime="1022074845" human-mtime="May 22 23:40"/>
      <file name="resume" path="documents/personal" uid="1000"
        gid="1000" size="763" type="regular" mode="0644"
        mtime="1022078771" human-mtime="May 23 00:46"/>
    </file>
    <file name="shared" path="documents" uid="1000" gid="1000"
      size="4096" type="directory" mode="0755" mtime="1022074911"
      human-mtime="May 22 23:41">
      <file name="housekeeping" path="documents/shared" uid="1000"
        gid="1000" size="0" type="regular" mode="0644"
        mtime="1022074911" human-mtime="May 22 23:41"/>
    </file>
  </file>
</files>
```

The output document uses the namespace `urn:ns:clm.xml-unix.ls`. The specific namespace that was chosen here does not matter, though it is important that it is unique. The namespace will be used by the shell after the `ls` program is run to determine the correct transformation to perform on this output to make it human readable. The `rec` attribute is used in the root element to inform the transformer that a recursive directory listing format is required when it comes time display it on the terminal.

The last modified time is now bundled up in a single attribute, to facilitate its extraction. In fact, for the last modified time, two attributes are used to represent it. The first, `mtime`, holds the UNIX time — this is the number of seconds since the UNIX epoch, 1 January 1970. The second attribute is `human-mtime`, and this holds the human readable form of the last modified time. It is this field which will be output to

the terminal in the directory listing, as it is much easier for a human to understand than the UNIX time.

4.3.2. ps

The **ps** program reports the status of processes running on the system. **ps**, while differing in behaviour across different UNIX systems, typically has two modes of process selection. By default, the **ps** command will only report processes which are running in the same session as itself. Usually this results in the processes which have been started under the current login shell being listed. The **-e** option instructs **ps** to list every process running. (There exist other criteria by which to select processes to list, such as by process ID or by user, however these are not as interesting as the major two mentioned, and can be reproduced using a filter on the data.)

Which properties of processes to report is the other main aspect of **ps**. By default, just four fields are shown — process ID, controlling terminal, running time and command. Using the **-f** flag asks for a full listing. A full listing comprises the username, process ID, parent process ID, percentage CPU usage, start time, controlling terminal, running time and command fields. If neither of these two alternatives is appropriate, the **-o** argument may be used to specify which fields should be displayed.

Two of these fields contain embedded spaces which make them harder to extract from the flat text output — the **lstart** field, which contains a long format date, and the **command** field, which holds the full command line for the process. Also, the **command** field and the **wchan** field, which contains the system call that process is currently running in, will be truncated if they are not the last column in the output.

A simple XML file is used to store all of the relevant information about a process in the new model **ps** program. An example output document from running **ps -f** is given below (very many attributes omitted):

```
<processes xmlns="urn:ns:clm.xml-unix.ps" sid="1925" full="true">
  <process pid="1" lstart='Mon Nov  4 10:59:05 2002' vsize='1264' uid='0'
    state='S' command='init' tty='?' time='00:00:04' sid='0' c='0'
    nice='0' ppid='0' user='root' wchan='select'/>
  <process pid="240" lstart='Mon Nov  4 10:59:24 2002' vsize='2156' uid='0'
    state='S' command='/sbin/dhclient-2.2.x -q eth0' tty='?'
    time='00:00:00' sid='240' c='0' nice='0' ppid='1' user='root'
    wchan='select'/>
  <process pid="244" lstart='Mon Nov  4 10:59:24 2002' vsize='1372' uid='1'
    state='S' command='/sbin/portmap' tty='?' time='00:00:00' sid='244'
    c='0' nice='0' ppid='1' user='daemon' wchan='poll'/>
  <process pid="347" lstart='Mon Nov  4 10:59:29 2002' vsize='2028' uid='0'
    state='S' command='/sbin/syslogd' tty='?' time='00:00:00' sid='347'
    c='0' nice='0' ppid='1' user='root' wchan='select'/>
  <process pid="350" lstart='Mon Nov  4 10:59:29 2002' vsize='1936' uid='0'
```

```

state='S' command='/sbin/klogd' tty='?' time='00:00:00' sid='350'
c='0' nice='0' ppid='1' user='root' wchan='syslog' />
</processes>

```

To signify that the "full" listing format is required, a **full** attribute is added to the root element of the document. Also, an **sid** attribute is added. This contains the process ID of the session leader of the **ps** process. This is helpful for the transformation process the shell will perform later, in determining which processes should be displayed.

Thus, although only some processes will be written to the terminal, all process details are included in the output. Selecting which processes to display is a formatting issue, and should be delayed until as late as possible.

4.3.3. df

The **df** program is a relatively simple tool to report the amount of free disk space on each of the currently mounted filesystems. Some systems, such as Linux, have a number of virtual filesystems which don't actually take up any space. These filesystems are by default not listed by **df**, unless the **-a** option is given.

df does not have much wrong with its output. It is straightforward and easy to parse. The only thing the user must be wary of is the header on the first line.

An XML version of the output of running **df -a** is shown below:

```

<filesystems xmlns="urn:ns:clm.xml-unix.df" all="true">
  <filesystem dev="/dev/hde2" blocks="29249536" free="1884160" use="94"
    mountpoint="/" />
  <filesystem dev="proc" blocks="0" free="0" use="0" mountpoint="/proc" />
  <filesystem dev="devpts" blocks="0" free="0" use="0" mountpoint="/dev/pts" />
  <filesystem dev="/dev/hde1" blocks="10481664" free="3760128" use="65"
    mountpoint="/winxp" />
  <filesystem dev="/dev/hdf3" blocks="28409856" free="1523712" use="95"
    mountpoint="/storage" />
  <filesystem dev="usb" blocks="0" free="0" use="0" mountpoint="/proc/bus/usb" />
</filesystems>

```

Like both **ls** and **ps**, a formatting option (in this case, **-a**) is handled by storing it as an attribute in the root element of the document.

5. Data filtering programs

The second category of programs in this new model of the UNIX command line environment is the data filtering programs. Filters are programs that take some data from standard input, manipulate it somehow, and then write that manipulated data to standard output. Since data generating programs now produce well formed XML documents on their standard output, the filtering programs will have to parse and manipulate that.

Parsing XML documents correctly is not a trivial task. Fortunately there are many freely available XML parsing libraries in existence. In the implementations presented in this project, the GNOME project's libxml2 library was used (www.xmlsoft.org).

5.1. Transforming the input

Once the input XML document has been parsed and a document tree constructed in memory, the filter can transform the document. At first glance, it may seem as if the majority of text filters in the traditional UNIX command line system can be converted to operate on XML documents without much paradigm shift. However, the fundamental difference in structure between flat text files and XML documents causes the XML filters to behave somewhat differently.

The most prominent difference in the behaviour of XML filters is that they need to know where in the document they will be operating. According to the model presented in this project, most XML documents generated by a data generating process will have an element for each record, whose parent element is the document element. This is a nice correspondence to the line-based records in a flat text file. XML filters, though, cannot rely on that structure being adhered to. Some processes will generate output with a different structure. XML data files on disk could have a completely different structure. The XML filters must therefore be able to handle the main data of an XML document being stored in a different location.

To achieve this, most XML filters presented here have a command line argument which instructs the filter to act upon certain elements in the document. By default, to concur with the description of a typical program output in section 3, the filters will act upon each element directly under the root element of the document.

5.2. Important filters

The programs presented in the following table have been identified as being important, and should be present in a complete implementation of an XML based UNIX command line environment.

Program	Purpose
cat	Concatenates files
grep	Searches for text which matches a regular expression
awk	Record/field matching scripting language
cut	Selects fields from a file
head	Selects the first records from a file
tail	Selects the last records from a file
sort	Sorts records in a file

5.3. Implementation examples

Presented now are three examples of XML filtering programs for the new UNIX command line environment.

5.3.1. cat

The **cat** program is a very simple one in the traditional UNIX command line environment. Its job is simply to concatenate files and write them out to standard output. The concatenation is trivial; since flat text files use line based records, the program needs only to write out each file in succession to standard output for them to be joined together.

Unfortunately, such simple behaviour will not work with XML documents. This is because outputting one document after another will not result in a well formed document. Well formed documents must contain exactly one root element. Thus, **cat** must do something else.

If it is assumed that records are stored as elements directly under the root element of the document, as outlined in the model description in section 3, then there is a fairly simple means of concatenating two XML documents together. All that must be done is to take the second level elements from the source documents and insert them just before the closing tag of the destination document's root element. For example, if the file **doc1.xml** contains:

```
<data>
  <record id="1"/>
  <record id="2"/>
</data>
```

and **doc2.xml** contains:

```
<data>
  <record id="3"/>
  <record id="4"/>
</data>
```

then running the command **cat doc1.xml doc2.xml** should result in the following document:

```
<data>
  <record id="1"/>
  <record id="2"/>
  <record id="3"/>
  <record id="4"/>
</data>
```

Note that the implementation here does not provide any checking to ensure both documents are in the same namespace or have the same root element. Traditional UNIX **cat** doesn't check for similar file formats either; it just outputs the files with no regard to if they "should" be allowed to be joined together.

If the important data is elsewhere in the source document, however, there needs to be some way of specifying which elements are to be selected. Similarly, if the destination document shouldn't have its root element appended to, there must be a way of specifying the insertion point for the new data. The **cat** program presented here has a command line argument **--from** which allows the user to give an XPath location specifying which elements from the source document are to be copied. Analogously, there exists a **--to** argument which should be followed by an XPath location indicating where in the destination document the source documents' elements are to be inserted.

The default **--from** XPath location is **/*/node()**, which refers to all of the nodes (that is, elements, text nodes and attributes), underneath the document's root element. The default **--to** XPath location is **/***, which refers to the root element of the destination document.

To demonstrate these options, consider the file **doc3.xml** to contain:

```
<html>
  <head>
    <title>Document title</title>
  </head>
  <body>
```

```
<p>This is from the first document.</p>
</body>
</html>
```

and **doc4.xml** to contain:

```
<html>
  <head>
    <title>Second document title</title>
  </head>
  <body>
    <p>This is from the second document.</p>
  </body>
</html>
```

If the command **cat --to /html/body doc3.xml --from '/html/body/*' doc4.xml** is then issued, the resulting document should be:

```
<html>
  <head>
    <title>Second document title</title>
  </head>
  <body>
    <p>This is from the first document.</p>
    <p>This is from the second document.</p>
  </body>
</html>
```

As encoding is an important issue with XML documents, unlike with traditionally 7-bit ASCII flat text files, the **cat** program also has a command line argument to specify which encoding should be used for output. Thus, to transcode an XML document to EBCDIC-US, the command **cat --encoding EBCDIC-US** is used. libxml2 can be compiled with support for the libiconv internationalisation library. Running the command **iconv -l** at the command line produces a list of encodings which **cat** should therefore be able to handle.

5.3.2. **grep**

The **grep** program is one of the most useful of the traditional UNIX command line environment filters. Its purpose is to select records (i.e., lines, in the flat text model) which match a given regular expression. As with **cat**, **grep** must be told where in the XML document it should be operating.

The **grep** program presented here takes two parameters to control where the search takes place. Firstly, it has a **--context** command line argument. This tells **grep** what each record of the XML document is. By default, this will be the XPath location **/*/***, the elements directly underneath the document's root element. Secondly, **grep** can be passed a **--node** argument. This controls, relative to each node that matches the context XPath location, which nodes should be compared against the regular

expression. The default node XPath location is `//text()`, that is any text node underneath the context node.

For the regular document structure mentioned earlier, the default XPath locations work well. For example, assuming that the file `doc5.xml` contains:

```
<data>
  <entry name="entry1">
    Hello there.
  </entry>
  <entry name="entry2">
    How <em>are</em> you?
  </entry>
  <entry name="entry3">
    I am well.
  </entry>
</data>
```

then issuing the command `grep 'a[^]' doc5.xml` will produce the following XML document:

```
<data>
  <entry name="entry2">
    How <em>are</em> you?
  </entry>
  <entry name="entry3">
    I am well.
  </entry>
</data>
```

To demonstrate the use of the `--context` and `--node` arguments, assume the file `doc6.xml` contains the following:

```
<data>
  <entries>
    <entry name="entry1">
      Hello there.
    </entry>
    <entry name="entry2">
      How <em>are</em> you?
    </entry>
    <entry name="entry3">
      I am well.
    </entry>
  </entries>
</data>
```

Then, the command `grep --context /data/entries/entry --node @name 'entry[12]' doc6.xml` can be used to select the first two entries:

```
<data>
  <entries>
    <entry name="entry1">
      Hello there.
    </entry>
```

```

    <entry name="entry2">
      How <em>are</em> you?
    </entry>
  </entries>
</data>

```

Notice that the two **entry** elements are returned still contained in the **entries** element. This is because the elements which match the regular expression are checked to see if they have a common parent, and since they do, they are placed within it in the output document.

5.3.3. sort

The traditional **sort** program rearranges the records in a file, sorting them by a particular field. The sort key is specified by giving a field number or range of characters. The **-r** command line argument is used to reverse the sort order, and **-n** is used to signify a numeric (rather than lexicographic) sort.

Just as with **cat** and **grep**, new version of **sort** defaults to sorting the elements directly beneath the document's root element. If this is not appropriate, a **--context** and **--node** command line argument can be given to select which elements are to be sorted, and on what they are to be sorted, respectively. By default, sorting is done just on the text nodes beneath each context node.

Below is a demonstration of how the new model **sort** works. If the file **doc7.xml** contains:

```

<foods>
  <staple id="4">Bread</staple>
  <fruit id="1">Apple</fruit>
  <fruit id="2">Starfruit</fruit>
  <staple id="3">Rice</staple>
</foods>

```

and the command **sort doc7.xml** is issued, the resulting XML document will be:

```

<foods>
  <fruit id="1">Apple</fruit>
  <staple id="4">Bread</staple>
  <staple id="3">Rice</staple>
  <fruit id="2">Starfruit</fruit>
</foods>

```

A different sort key can be specified. Running the command **sort --context '/foods/*' --node @id doc7.xml** results in:

```

<foods>
  <fruit id="1">Apple</fruit>
  <fruit id="2">Starfruit</fruit>
  <staple id="3">Rice</staple>

```

```
<staple id="4">Bread</staple>
</foods>
```

The **sort** command can also take an XPath expression (not just an XPath location) to sort on. This makes it possible to, for example, sort based on the name of the element being sorted. Executing **sort --context '/foods/*' --expr 'name()' doc7.xml** produces:

```
<foods>
  <fruit id="1">Apple</fruit>
  <fruit id="2">Starfruit</fruit>
  <staple id="4">Bread</staple>
  <staple id="3">Rice</staple>
</foods>
```

As can be seen from this example, the sort is stable. The two **fruit** elements are considered equal in the sorting, as are the two **staple** elements. Their relative ordering is the same as their relative ordering in the original document.

6. User interface to the environment

6.1. The shell

The final piece in the new model of the UNIX command line environment is the shell. The shell is the program which interacts with the user via the terminal, allowing the user to execute commands and view their output. The new shell behaves exactly like the Bourne shell in most respects. However, it also has the ability to transform the XML output of a program to a human readable format suitable for display on the terminal.

6.2. Human readable output

At the end of a command pipeline, an XML document will be produced. This document, while still readable, isn't intended for viewing by the user. Instead, a formatted representation of the data should be printed to the terminal. But to determine what sort of transformation should be applied to the output document, that document must be read in and parsed by the shell.

The shell uses a configuration file, named `/etc/transforms.xml` in the implementation given here, to decide which XSLT stylesheet should be used to transform the XML output document. An example of this file is given below:

```
<transforms>
  <transform namespace="urn:ns:clm.xml-unix.df">
    <target type="terminal" stylesheet="transforms/df-terminal.xsl"/>
  </transform>
  <transform namespace="urn:ns:clm.xml-unix.ps">
    <target type="terminal" stylesheet="transforms/ps-terminal.xsl"/>
  </transform>
  <transform namespace="urn:ns:clm.xml-unix.ls">
    <target type="terminal" stylesheet="transforms/ls-terminal.xsl"/>
  </transform>
</transforms>
```

The configuration file associates an XSL stylesheet whose output is appropriate for displaying on a terminal with each type of document that can be produced by the data generating programs. The namespace of the XML output document is compared against that stored in the **namespace** attribute of each **transform** element in the **transforms.xml** file. If a match is found, the relevant stylesheet is used to transform the output which is then written to the terminal. If no match is found, the XML output document is written to the terminal verbatim.

However, the user will not want this automatic transformation to occur all the time.

This is especially true when the user is building up a command pipeline. In a traditional UNIX system, pipelines are generally built up incrementally. First, the user would try one command. Then, upon inspecting the output of this command, they would pipe that output through a filter. If the new shell is ever converting the XML output documents into a human readable form, the user will have no chance at determining through what command the output should be piped. For this reason, the shell introduces a new character that instructs it to dump the XML output document to the terminal without converting it. This character, the caret (^) is placed at the end of the pipeline to ensure that the markup is kept.

To exemplify the behaviour of the shell, below is some sample output from the shell after entering some commands:

```
$ df
Filesystem          1k-blocks      Used Available Use% Mounted on
/dev/hde2           29249536  27398144   1851392  94% /
/dev/hde1           10481664    6721536   3760128  65% /winxp
/dev/hdf3           28409856   26886144   1523712  95% /storage
$ df ^
<filesystems xmlns="urn:ns:clm.xml-unix.df">
  <filesystem dev="/dev/hde2" blocks="29249536" free="1851392" use="94"
    mountpoint="/" />
  <filesystem dev="proc" blocks="0" free="0" use="0" mountpoint="/proc" />
  <filesystem dev="devpts" blocks="0" free="0" use="0" mountpoint="/dev/pts" />
  <filesystem dev="/dev/hde1" blocks="10481664" free="3760128" use="65"
    mountpoint="/winxp" />
  <filesystem dev="/dev/hdf3" blocks="28409856" free="1523712" use="95"
    mountpoint="/storage" />
  <filesystem dev="usb" blocks="0" free="0" use="0" mountpoint="/proc/bus/usb" />
</filesystems>
```

The presence of the caret after the command prevents the shell from using the XSLT stylesheet to produce the formatted output.

7. Evaluating the model

Presented in this section are some examples of the new command line environment in use, and a discussion of some problems that were encountered with the model.

7.1. Common applications

To determine the efficacy of the new model for the UNIX command line environment, some common tasks are performed in both the traditional model and the new model. The commands used to perform the tasks are then compared.

7.1.1. Determine filesystem with most free space

The goal of this task is to determine which filesystem has the most free disk space. In order to determine which filesystem has the most free space, the **df** command must of course be used. Both the traditional and the new **df** print the same output to the terminal, but of course underneath it is all different.

To select the filesystem with the largest amount of free space, the **sort** command can be used to arrange the records such that the last record is the one with the most free space. In the traditional system, the user must be mindful of the header **df** also outputs. Since it is part of the flat text output, it must be avoided. The **tail +2** command can be used to remove that first line. Subsequently, the records can be sorted. Since the traditional **sort** command uses field numbers to select the key, the **sort -n +3** command is used to sort the records numerically by the fourth column. Finally, the **tail -1** command can be used to select the last record from the sorted list. That record will be the one with the highest amount of free space.

In the new model, the header does not form part of **df**'s output, so it does not have to be avoided as must be done in the traditional system. The output of **df** can be piped straight into **sort**. Inspecting the output format of **df**, it is apparent that the **free** attribute contains the number of free blocks on a filesystem. Thus, the command **df**'s output must be piped through is **sort --numeric --node @free**. Once the records are sorted, the last one can be chosen as with traditional system, using **tail -1**.

In summary, the traditional UNIX command to find the filesystem with the most free space is:

```
df | tail +2 | sort -n +3 | tail -1
```

The command needed in the new model is

```
df | sort --numeric --node @free | tail -1
```

The new model command, while taking more keystrokes to type, uses one fewer pipeline stage to generate the output. Also, it is more intuitive than the traditional UNIX command. The traditional UNIX command requires a numeric reference to the sort key, while the new model command uses a symbolic key name. The **sort** command should thus be easier for the user to write.

7.1.2. Determine which process has been running the longest

This goal of this task is to display only the process ID and the command line of the process which has used the most CPU time.

Obviously, the **ps** command will need to be used in this command pipeline. To begin with in the traditional system, all processes are selected for display by the **ps** command. Also, the relevant fields (process ID, CPU time and command line) are also selected. As with the previous task, the header must be stripped off. Luckily, the GNU **ps** program being used has a **--no-headers** command line option. Thus the **ps** command used is **ps -e -o pid,time,command --no-headers**.

Once all of the records are available, they must be sorted based on their running time. This is done with the **sort** command, specifying the second field as the sort key. The command used is **sort +1**. The last record in the output will now be for the process with the longest CPU time. It can be selected with the **tail -1** command.

Finally, since the goal of this task is to display only the process ID and the command line, the CPU time must be removed. Unfortunately, since the command line could have spaces in its field, a field based solution (such as using **awk**) is not possible. Instead, **sed** can be used to match the time and remove it from the line. The command to do this is **sed 's///'**.

Building the pipeline in the new system starts similarly. However, when selecting which fields to display, we can omit the CPU time. This wasn't possible in the traditional model, since if the CPU time wasn't included in the initial **ps** output, its value could not be used as the sort key. With the new model, however, all of the information about each process is included in **ps**'s output document. It is just that the process ID and command line can be registered as the only fields to be displayed when the shell finally transforms the document for output to the terminal. Just like in the previous task, the header is part of the formatting and thus is not present in the XML output document of **ps**. Thus it doesn't need to be explicitly excluded. The initial command in the pipeline is therefore **ps -e -o pid,command**.

Looking at the raw XML output from the **ps** command, it can be seen that the **time**

attribute holds the CPU time field to be sorted on. So, the next command in the pipeline is **sort --node @time**. The last record in the sorted list, which has the greatest CPU time, can be selected with **tail -1**. Since the formatting options given to **ps** initially preclude the CPU time from being displayed, there is no need to remove it as in the traditional model.

Comparing the two models, the traditional model solution resulted in the command:

```
ps -e -o pid,time,command --no-headers | sort +1 | tail -1 | sed 's/ ..:...:..//'
```

while the new model solution produced the command:

```
ps -e -o pid,command | sort --node @time | tail -1
```

The new model gives the user a much cleaner and shorter command to produce the correct output. Again, the command is more intuitive (especially the **sort** command) than the equivalent traditional model command. The traditional model command needs more text processing to massage the data into an acceptable form.

7.2. Issues with the model

While developing the model, the idea that the user programs did not have read in and output complete, well formed XML documents was bandied about. Since some programs might extract individual attributes or just text from an XML document, the question of whether it made sense to allow them to be valid output and input to other programs was raised. An implementation of a model allowing such data was worked on, however it soon lost the simplicity of the original model. Some programs could perform sensible tasks with these sub-documents, while others had to reject incomplete documents.

A particular example which illustrates the problem is that of XInclude (Marsh, Orchard 2002). XInclude allows the user to include another XML document in the current document by referring to its URL. This inclusion is a direct substitution, in much the same way as a `#include` is performed in C. The problem with XInclude is that it allows the user to include a file which contains just a collection of elements without a single root element. This included file is not a well formed XML document. Thus the question of how such a file should be processed in the new model of the command line environment arose.

The idea of allowing the processing of invalid XML documents was then abandoned, and the XML processing programs returned to processing only complete XML documents.

8. Conclusion

During this project a comprehensive model for UNIX process input and output using XML documents was developed. This model specified the general format to be used for a process' standard input and output. The requirement that formatting should be kept separate from the content of a process' output was the driving force behind the rest of the model.

Once the model was devised, implementations of a few UNIX programs conforming to the new model were created. These programs, while not forming a complete system, demonstrated the concept the model is trying to convey.

Some sample tasks were also performed with both a traditional UNIX command line environment and the environment afforded by the new model. Comparing the commands used to perform the tasks in both models, the claim that the new model provides the user with a more intuitive in which to work, at least for the examples given, are vindicated.

8.1. Future work

A number of areas exist where further research can be undertaken. Firstly, there are a few programs which need to be written to complete the system. Also, the programs need to be set in a context where support for the XML file format is found throughout the system. Specifically, a UNIX-like operating system could be developed with native support for the XML file format. Configuration and most other files in the system would have to use XML too, for the programs to prove most useful.

Proper analysis of the interface to determine if users actually find the environment more intuitive than the traditional environment could be performed. This might involve surveying users' opinions of the environment and developing a better human-computer interface for it.

Finally, the model currently deals only with XML documents and not other file formats. The model could also be reviewed to determine if support for other file formats should be written in to the implemented programs or if they should be kept solely for processing XML documents. Having only one version of the `cat` program, for example, which could concatenate XML documents or flat text documents, could be advantageous.

Bibliography

Bensoussan, A., Clingen, C.T. & Daley, R.C. (1972), "The Multics Virtual Memory: Concepts and Design," *Communications of the ACM*, vol. 15, no. 5, pp. 308—318.

Berners-Lee, T. (1994, Jun.), "Uniform Resource Identifiers in WWW," The Internet Engineering Taskforce [online], Available: <http://www.ietf.org/rfc/rfc1630.txt> [Accessed 4 Nov 2002].

Bolsky, M.I. & Korn, D.G. (1995), *The new Kornshell, 2nd ed.*, Prentice Hall, Englewood Cliffs.

Bourne, S.R. (1983), *The UNIX system*, Addison-Wesley, Reading, Mass.

Bray, T., Sperberg-McQueen, C.M., Paoli, J. & Maler, E. (2000, Oct. 6), "Extensible Markup Language (XML) 1.0 (Second Edition)," The World Wide Web Consortium [online], Available: <http://www.w3.org/TR/REC-xml> [Accessed 30 July 2002].

Clark, J. (1999, Nov. 16), "XSL Transformations (XSLT)," The World Wide Web Consortium [online], Available: <http://www.w3.org/TR/1999/REC-xslt-19991116> [Accessed 2 May 2002].

Clark, J. & DeRose, S. (1999, Nov. 16), "XML Path Language (XPath)," The World Wide Web Consortium [online], Available: <http://www.w3.org/TR/1999/REC-xpath-19991116> [Accessed 5 Nov 2002].

Corbató, F.J. & Vyssotsky, V.A. (1965), "Introduction and Overview of the Multics System," *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 27, no. 1, pp. 185—196.

Daley, R.C. & Dennis, J.B. (1968), "Virtual memory, processes, and sharing in Multics," *Communications of the ACM*, vol. 11, no. 5, pp. 306—312.

Daley, R.C. & Neumann, P.G. (1965), "A General Purpose File System for Secondary Storage," *Proceedings of AFIPS Fall Joint Computer Conference*, vol. 27, no. 1, pp. 213—229.

Falstad, P. (2001, Oct. 24), "The Z Shell Manual," ZSH - THE Z SHELL [online], Available: http://zsh.sunsite.dk/Doc/zsh_a4.ps.gz [Accessed 30 July 2002].

Feiertag, R.J. & Organick, E.I. (1971), "The Multics input-output system," *Proceedings of the Third Symposium on Operating Systems Principles*, pp. 35—41.

Gancarz, M. (1995), *The UNIX Philosophy*, Digital Press, Boston.

International Organization for Standardization (2000), *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*, ISO 10646-1:2000, Geneva.

International Organization for Standardization (1986), *Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*, ISO 8879:1986, Geneva.

Kernighan, B.W. & Ritchie, D.M. (1978), *The C Programming Language*, Prentice Hall.

Kernighan, B.W. & Plauger, P.J. (1976), *Software Tools*, Addison-Wesley, Reading, Massachusetts.

Marsh, J. & Orchard, D. (2002, Sep. 17), "XML Inclusions (XInclude)," The World Wide Web Consortium [online], Available:
<http://www.w3.org/TR/2002/CR-xinclude-20020917> [Accessed 5 Nov 2002].

McDonald, C.S. (1987), "fsh — A Functional UNIX Command Interpreter," *Software — Practice and Experience*, vol. 17, no. 10, pp. 687—700.

McKusick, K. (1999), "Chapter 2 — Twenty Years of Berkeley Unix: From AT&T-Owned to Freely Redistributable," in *Open Sources: Voices from the Open Source Revolution*, DiBona, C., Ockman, S. & Stone, M. (eds.), O'Reilly & Associates, Sebastapol, California, pp. 31—46.

Ritchie, D.M. & Thompson, K. (1974), "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365—375.

Ritchie, D.M. (1979), "The Evolution of the UNIX Time-sharing System," in *Language Design and Programming Methodology*, Sydney, Australia, September 1979, Springer-Verlag.

Shivers, O., Carlstrom, B.D., Gasbichler, M. & Sperber, M. (2002, May), "Scsh Reference Manual," Scsh - The Scheme Shell [online], Available:
<ftp://ftp.scsh.net/pub/scsh/0.6/scsh-manual.ps.gz> [Accessed 30 July 2002].

The Unicode Consortium, (2000), *The Unicode standard, version 3.0*, Addison-Wesley, Reading, Mass.

Appendix A. Code

The code for a few of the implementations presented in this report is given here. The full code will be available from <http://www.csse.monash.edu.au/~clm/uni/honours/>.

A.1. ls.pl

```
#!/usr/bin/perl -w
# ls.pl
#
# A version of ls that outputs in XML.

use File::stat;
use Fcntl ':mode';

# Return just the filename from a pathname.
sub basename {
    my $f = shift;
    return '.' if !defined $f || $f eq '';
    return '/' if $f eq '/';
    return $f unless m[^\./(.*)];
    return $1;
}

# Return just the path from a pathname.
sub basepath {
    my $f = shift;
    return '.' if !defined $f || $f eq '';
    return '.' unless m[^\./(.*)/];
    return '/' if $1 eq '';
    return $1;
}

sub usage {
    print <<EOF;
    Usage: ls [--all] [--long] [--recursive] [FILE ...]

    Options/Arguments:
        --all          Even display dotfiles.
        --long         Display detailed file listing.
        --recursive   Recurse into subdirectories.
    EOF
}

exit 1;
}

# Convert a permission to its symbolic form.
sub mde {
    my $m = shift;
    my $c = '';
    $c .= ($m & 4 ? 'r' : '-');
    $c .= ($m & 2 ? 'w' : '-');
    $c .= ($m & 1 ? 'x' : '-');
    return $c;
}

# Format a time just like "ls -l" does.
@month = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
sub hmt {
    my $t = shift;
    my @l = localtime($t);
    my $now = time;
    my $threshold = 6 * 30 * 24 * 60 * 60;
    if ($t < $now - $threshold || $t > $now + $threshold) {
        return sprintf('%s %2d %d', $month[$l[4]], $l[3], $l[5] + 1900);
    } else {
        return sprintf('%s %2d %02d:%02d', $month[$l[4]], $l[3], $l[2], $l[1]);
    }
}
```

```

}
}

# Recursive function to handle a file.
sub dofile {
my ($sb, $name, $path, $ind) = @_;
if (defined $sb) {
print ' ' x $ind, '<file';
print " name=\"$name\"";
print " path=\"$path\"";
printf ' device="%X", $sb->dev;
printf ' inode="%d", $sb->ino;
printf ' nlink="%d", $sb->nlink;
printf ' uid="%d", $sb->uid;
printf ' gid="%d", $sb->gid;
printf ' rdev="%X", $sb->rdev;
printf ' size="%d", $sb->size;
printf ' blksize="%d", $sb->blksize;
printf ' blocks="%d", $sb->blocks;
$type = 'unknown';
if (S_ISREG($sb->mode)) {
$type = 'regular';
} elsif (S_ISDIR($sb->mode)) {
$type = 'directory';
} elsif (S_ISLNK($sb->mode)) {
$type = 'symlink';
} elsif (S_ISBLK($sb->mode)) {
$type = 'block';
} elsif (S_ISCHR($sb->mode)) {
$type = 'character';
} elsif (S_ISFIFO($sb->mode)) {
$type = 'fifo';
} elsif (S_ISSOCK($sb->mode)) {
$type = 'socket';
} elsif (S_ISWHT($sb->mode)) {
$type = 'whiteout';
}
print " type=\"$type\"";
printf ' mode="%04o", $sb->mode & 0777;
printf ' atime="%d", $sb->atime;
printf ' mtime="%d", $sb->mtime;
printf " ctime=\"%d\", $sb->ctime;

# Loop up the owner's username.
@pw = getpwuid($sb->uid);
if (@pw) {
$u = $pw[0];
} else {
$u = $sb->uid;
}
print " username=\"$u\"";

# Look up the the group's name.
@g = getgrgid($sb->gid);
if (@g) {
$g = $g[0];
} else {
$g = $sb->gid;
}
print " group=\"$g\"";

# Construct the symbolic mode.
if ($type eq 'regular') {
$modechars = '-';
} elsif ($type eq 'directory') {
$modechars = 'd';
} elsif ($type eq 'symlink') {
$modechars = 'l';
} elsif ($type eq 'block') {
$modechars = 'b';
} elsif ($type eq 'character') {
$modechars = 'c';
} elsif ($type eq 'fifo') {
$modechars = 'p';
}

```

```

} elsif ($type eq 'socket') {
    $modechars = 's';
} else {
    $modechars = '?';
}

$modechars .= mde(($sb->mode & 0700) >> 6);
$modechars .= mde(($sb->mode & 0070) >> 3);
$modechars .= mde($sb->mode & 0007);

print " modechars=\"\$modechars\"";

$hmt = hmt($sb->ctime);
print " human-mtime=\"\$hmt\"";

$modechars = '';
if ($rec && $type eq 'directory') {
    # Recurse.
    print ">\n";
    opendir DIR, "$path/$name";
    for (grep { $_ ne '.' && $_ ne '..' } readdir DIR) {
        dofile(lstat("$path/$name/$_"), $_, ($path eq '.' ? $name
            : "$path/$name"), $ind + 2);
    }
    closedir DIR;
    print ' ' x $ind, "</file>\n";
} else {
    print "/>\n";
}
} else {
    print STDERR " ${name}: file not found\n";
}
}

# Parse command line.
for ($i = 0; $i <= $#ARGV; $i++) {
    if ($ARGV[$i] eq '--help' || $ARGV[$i] eq '-h') {
        usage();
    } elsif ($ARGV[$i] eq '--recursive' || $ARGV[$i] eq '-R') {
        $rec = 1;
    } elsif ($ARGV[$i] eq '--all' || $ARGV[$i] eq '-a') {
        $all = 1;
    } elsif ($ARGV[$i] eq '--long' || $ARGV[$i] eq '-l') {
        $long = 1;
    } elsif ($ARGV[$i] =~ /^~/) {
        usage();
    } else {
        push @files, $ARGV[$i];
    }
}

print "<files xmlns=\"urn:ns:clm.xml-unix.ls\"";
print " all=\"true\" if $all;
print " long=\"true\" if $long;
print " rec=\"true\" if $rec;
print ">\n";

# Work on the files in the current directory if no files were given
# on the command line.
unless (@files) {
    if ($rec) {
        @files = ('.');
    } else {
        opendir DIR, '.';
        @files = readdir DIR;
        closedir DIR;
    }
}

# Handle each file.
for (@files) {
    dofile(lstat($_), basename($_), basepath($_), 2);
}

```

```
print "</files>\n";
```

A.2. ps.pl

```
#!/usr/bin/perl
# ps.pl
#
# A version of ps that outputs in XML.

use Getopt::Long qw(:config bundling);

# Well behaved fields.
@fields = qw(alarm blocked c caught cp pcpu drs dsiz egid egroup eip etime esp
  euid euser flags fgid fgroup fsgid fsgroup fsuid fsuser fuid fuser gid
  group ignored lim majflt pmem minflt nice pagein pending pgid pgrp ppid
  pri intpri priority psr rgid rgroup rss ruid ruser state sgid sgroup sid
  spid stackp suid suser svgid svgroup svuid svuser sz thcount tid time
  timeout tpgid trss tty uid user vsize nwchan);
# Fields which require sepcial treatment.
@specialfields = qw(lstart wchan command);

# Escape value to fit in an attribute.
sub escape {
  my $x = shift;
  s/&/&amp;/g;
  s/'/'&quot;/g;
  s/</&lt;/g;
  return $x;
}

sub usage {
  print <<EOF;
Usage: ps [--all] [--full] [--format field,field,...]

Options/Arguments:
  --all      Display all processes, not just those in this session.
  --full    List full details of each process.
  --format   Display given fields in output.
EOF
}

exit 1;
}

# Parse command line.
for ($i = 0; $i <= $#ARGV; $i++) {
  $_ = $ARGV[$i];
  usage() if $_ eq '--help' || $_ eq '-h';
  $all = 1 if $_ eq '--all' || $_ eq '-a' || $_ eq '-e';
  $full = 1 if $_ eq '--full' || $_ eq '-f';
  if ($_ eq '--format' || $_ eq '-o') {
    $i++;
    $format = $ARGV[$i];
  }
}

$fieldlist = 'pid';
for (@fields) {
  $fieldlist .= ",$_";
}

# Run the system ps
%p = ();
open PS, "/bin/ps --no-header -ewo $fieldlist |";
for (<PS>) {
  s/^\s*//;
  @f = split /\s+/;
  $n = 0;
  $pid = shift @f;
  # Extract the fields from the output
  %p{$pid} = {};
  for (@f) {
    %p{$pid}{$fields[$n]} = $_;
    $n++;
  }
}
```

```

}

# Run the system ps again for the badly behaved fields
for $field (@specialfields) {
  open PS, "/bin/ps --no-header -ewo pid,$field |";
  for (<PS>) {
    # Extract that field from the output
    /(\d+)\s+(.*)/;
    $p{$1}{$field} = $2;
  }
}

close PS;

print "<processes xmlns=\"urn:ns:clm.xml-unix.ps\"";
print " sid=\"\", $p{${$}}{sid}, \"\"";
print " all=\"true\"\" if $all;
print " full=\"true\"\" if $full;
print " format=\"$format\"\" if $format ne '';
print ">\n";
# Dump processes in order of PID
for (sort { $a <=> $b } keys %p) {
  $proc = $p{$_};
  print " <process pid=\"$_\"#command=\"$proc->{command}\"/>\n";
  for (keys %$proc) {
    print " $_='\", escape($proc->{$_}), \"'\";
  }
  print "/>\n";
}
print "</processes>\n";

```

A.3. cat.cpp

```

#include <iostream>
#include <string>
#include <vector>

#include <libxml/parser.h>
#include <libxml/xpath.h>

const string DEFAULT_DEST_XPATH = "/*";
const string DEFAULT_SOURCE_XPATH = "/*/node()";

// Class to encapsulate command line arguments.
class Args
{
  int _argc;
  vector<string> _args;
public:
  typedef vector<string>::iterator iterator;
  typedef vector<string>::const_iterator const_iterator;

  Args(int argc, char *argv[])
    : _argc(argc)
  {
    _args.resize(argc);
    for (int i = 0; i < argc; i++)
      _args[i] = argv[i];
  }

  const string& prog_name() const
  {
    return _args[0];
  }

  iterator begin()
  {
    return _args.begin();
  }

  iterator end()
  {
    return _args.end();
  }

```

```

    }
};

// Source document filename and XPath
struct Path
{
    string _file;
    string _xpath;
};

bool is_option(const string& s)
{
    return s[0] == '-';
}

void usage()
{
    cerr << "Usage: cat [--format] [--encoding enc] [--to xpath] destdoc\n"
          "      [--from xpath] sourcedoc ...]\n"
          "\n"
          "Options/Arguments:\n"
          "\t--format      adds whitespace to the output document to indent it.\n"
          "\t--encoding enc  writes the output document in the given encoding.\n"
          "\tdestdoc       XML document to concatenate on to.\n"
          "\t--to xpath     XPath expression indicating where in destdoc\n"
          "\t              the concatenated nodes should be inserted.\n"
          "\tsourcedoc     XML document to be concatenated on to destdoc.\n"
          "\t--from xpath   XPath expression indicating which nodes in\n"
          "\t              sourcedoc should be used.\n"
          "\n"
          "The source XPath expression defaults to \"/*&#x2F;node()\".\n"
          "The destination XPath expression defaults to \"/*&#x2F;\".\n"
          ;
    exit(1);
}

void parse_commandline(Args& args, Path& dest, vector<Path>& source,
    bool& format, string& encoding)
{
    int state = -3;
    Args::const_iterator i = args.begin() + 1;
    Path next_source;
    while (i != args.end())
    {
        if (*i == "--help" || *i == "-h")
            usage();

        switch (state)
        {
            case -3:
                if (*i == "--format" || *i == "-f")
                {
                    format = true;
                    i++;
                }
                state = -2;
                break;

            case -2:
                if (*i == "--encoding" || *i == "-e")
                {
                    state = -1;
                    i++;
                }
                else
                    state = 0;
                break;

            case -1:
                if (is_option(*i))
                    usage();
                encoding = *i;
                state = 0;
                i++;
        }
    }
}

```

```

        break;

case 0:
    if (*i == "--to" || *i == "-t")
    {
        state = 1;
        i++;
    }
    else if (*i == "--from" || *i == "-f")
    {
        state = 4;
        next_source._xpath = DEFAULT_SOURCE_XPATH;
        i++;
    }
    else
        state = 2;
    break;

case 1:
    if (is_option(*i))
        usage();
    dest._xpath = *i;
    state = 2;
    i++;
    break;

case 2:
    if (*i == "--from" || *i == "-f")
    {
        state = 4;
        next_source._xpath = DEFAULT_SOURCE_XPATH;
        i++;
    }
    else if (is_option(*i))
        usage();
    dest._file = *i;
    state = 3;
    i++;
    break;

case 3:
    next_source._xpath = DEFAULT_SOURCE_XPATH;
    if (*i == "--from" || *i == "-f")
    {
        state = 4;
        i++;
    }
    else
        state = 5;
    break;

case 4:
    if (is_option(*i))
        usage();
    next_source._xpath = *i;
    state = 5;
    i++;
    break;

case 5:
    if (is_option(*i))
        usage();
    next_source._file = *i;
    source.push_back(next_source);
    state = 3;
    i++;
    break;
    }
}

if (state == -1 || state == 1 || state == 4 || state == 5)
    usage();
}

```

```

void error(const string& msg)
{
    cerr << "cat: " << msg << endl;
    exit(2);
}

int main(int argc, char *argv[])
{
    Args args(argc, argv);

    Path dest;
    vector<Path> source;
    bool format;
    string encoding;

    dest._file = "-";
    dest._xpath = DEFAULT_DEST_XPATH;

    parse_commandline(args, dest, source, format, encoding);

    xmlSubstituteEntitiesDefault(1);
    xmlLoadExtDtdDefaultValue = 1;

    // Read in the destination document
    xmlDocPtr xml_dest = xmlParseFile(dest._file.c_str());

    // Evaluate the XPath expression to find the destination document's
    // insertion point
    xmlXPathContextPtr xml_dest_context = xmlXPathNewContext(xml_dest);

    xmlXPathObjectPtr insertion_point_nodeseq = xmlXPathEval(
        (const xmlChar*) dest._xpath.c_str(), xml_dest_context);
    if (insertion_point_nodeseq == NULL)
        error("Destination XPath must be valid");
    if (insertion_point_nodeseq->type != XPATH_NODESET)
        error("Destination XPath must evaluate to a nodeset");
    if (insertion_point_nodeseq->nodesetval == 0 ||
        insertion_point_nodeseq->nodesetval->nodeNr == 0)
        error("Destination XPath must evaluate to a nodeset with at least one node");
    xmlNodePtr insertion_point = insertion_point_nodeseq->nodesetval->nodeTab[0];

    xmlXPathFreeObject(insertion_point_nodeseq);
    xmlXPathFreeContext(xml_dest_context);

    // Insert each source document's nodes into the destination document at the
    // insertion point
    for (vector<Path>::const_iterator i = source.begin(); i != source.end(); i++)
    {
        // Read in the source document
        xmlDocPtr xml_source = xmlParseFile(i->_file.c_str());

        // Evaluate the XPath expression to find the source document's nodes
        xmlXPathContextPtr xml_source_context = xmlXPathNewContext(xml_source);

        xmlXPathObjectPtr source_nodeseq = xmlXPathEval(
            (const xmlChar*) i->_xpath.c_str(), xml_source_context);
        if (source_nodeseq == NULL)
            error("Source XPath must be valid");
        if (source_nodeseq->type != XPATH_NODESET)
            error("Source XPath must evaluate to a nodeset");

        // Loop through each node in the nodeset and add them to the
        // destination document
        for (int j = 0; j < source_nodeseq->nodesetval->nodeNr; j++)
            xmlAddChild(insertion_point, xmlCopyNode(
                source_nodeseq->nodesetval->nodeTab[j], 1));

        xmlXPathFreeObject(source_nodeseq);
        xmlXPathFreeContext(xml_source_context);
        xmlFreeDoc(xml_source);
    }

    int save_ret;
    if (encoding == "")

```

```

    save_ret = xmlSaveFormatFile("-", xml_dest, format ? 1 : 0);
else
    save_ret = xmlSaveFormatFileEnc("-", xml_dest, encoding.c_str(),
    format ? 1 : 0);

if (save_ret == 0)
    error("Error writing output document");

xmlFreeDoc(xml_dest);
xmlCleanupParser();
}

```

A.4. sort.cpp

```

#include <iostream>
#include <string>
#include <vector>

#include <libxml/parser.h>
#include <libxml/xpath.h>

const string DEFAULT_CONTEXT_XPATH = "/*/*";
const string DEFAULT_NODE_XPATH = ".//text()";

// Class to encapsulate command line arguments.
class Args
{
    int _argc;
    vector<string> _args;
public:
    typedef vector<string>::iterator iterator;
    typedef vector<string>::const_iterator const_iterator;

    Args(int argc, char *argv[])
        : _argc(argc)
    {
        _args.resize(argc);
        for (int i = 0; i < argc; i++)
            _args[i] = argv[i];
    }

    const string& prog_name() const
    {
        return _args[0];
    }

    iterator begin()
    {
        return _args.begin();
    }

    iterator end()
    {
        return _args.end();
    }
};

bool is_option(const string& s)
{
    return s[0] == '-';
}

void usage()
{
    cerr << "Usage: sort [--context xpath] [--node xpath | --expr xpathexpr] [--reverse] [--numeric] [source\n"
        "\n"
        "Options/Arguments:\n"
        "\t--context xpath    XPath location indicating the context of the \"match\"\n"
        "\t                    path.\n"
        "\t--node xpath        XPath location indicating which nodes in the context\n"
        "\t                    nodes should be used for sorting.\n"
        "\t--expr xpathexpr    XPath expression evaluated in the given context to\n"
        "\t                    be used for sorting.\n"

```

```

        "\t--reverse          sorts nodes in reverse order.\n"
        "\t--numeric          indicates that sorting should be performed numerically, not\n"
        "\t                    lexicographically.\n"
        "\tsourcedoc            XML document to be sorted.\n"
        "\n"
        "The context XPath location defaults to \"/*/*\n"
        "The node XPath location defaults to \".//text()\n"
        ;
    exit(1);
}

void parse_commandline(Args& args, vector<string>& source, string& context,
    string& node, string& expr, bool& numeric, bool& reverse)
{
    int state = 0;
    Args::const_iterator i = args.begin() + 1;

    node = "";
    expr = "";

    while (i != args.end())
    {
        if (*i == "--help" || *i == "-h")
            usage();

        switch (state)
        {
            case 0:
                if (*i == "--context" || *i == "-c")
                {
                    state = 1;
                    i++;
                }
                else if (*i == "--node" || *i == "-n")
                {
                    state = 2;
                    i++;
                }
                else if (*i == "--expr" || *i == "-e")
                {
                    state = 3;
                    i++;
                }
                else if (*i == "--reverse" || *i == "-r")
                {
                    reverse = true;
                    i++;
                }
                else if (*i == "--numeric" || *i == "-N")
                {
                    numeric = true;
                    i++;
                }
                else
                {
                    if (is_option(*i))
                        usage();
                    state = 4;
                }
                break;

            case 1:
                if (is_option(*i))
                    usage();
                context = *i;
                state = 0;
                i++;
                break;

            case 2:
                if (is_option(*i) || !expr.empty())
                    usage();
                node = *i;
                state = 0;
        }
    }
}

```

```

        i++;
        break;

    case 3:
        if (is_option(*i) || !node.empty())
            usage();
        expr = *i;
        state = 0;
        i++;
        break;

    case 4:
        if (is_option(*i))
            usage();
        source.push_back(*i);
        i++;
        break;
    }
}

if (state == 1 || state == 2 || state == 3)
    usage();
}

void error(const string& msg)
{
    cerr << "sort: " << msg << endl;
    exit(2);
}

string context = DEFAULT_CONTEXT_XPATH;
string node;
string expr;
bool numeric = false;
bool reverse = false;

// Compare two nodes in a document.
bool lt(xmlNodePtr a, xmlNodePtr b, xmlXPathContextPtr c)
{
    if (!node.empty())
    {
        // --node used on command line

        // Evaluate node in the a context
        c->node = a;
        xmlXPathObjectPtr an = xmlXPathEval((xmlChar*) node.c_str(), c);
        // Evaluate node in the b context
        c->node = b;
        xmlXPathObjectPtr bn = xmlXPathEval((xmlChar*) node.c_str(), c);
        // Loop over each node in the result nodesets
        for (int i = 0; i < min(an->nodesetval->nodeNr, bn->nodesetval->nodeNr); i++)
        {
            // Cast the nodes to a string
            xmlChar* s1 = xmlXPathCastNodeToString(an->nodesetval->nodeTab[i]);
            xmlChar* s2 = xmlXPathCastNodeToString(bn->nodesetval->nodeTab[i]);
            // Compare the strings
            int r = xmlStrcmp(s1, s2);
            xmlFree(s1);
            xmlFree(s2);
            if (r < 0)
                return !reverse;
            else if (r > 0)
                return reverse;
        }
        return false;
    }
    else
    {
        // --expr used on command line

        // Evaluate node in the a context
        c->node = a;
        xmlXPathObjectPtr an = xmlXPathEvalExpression((xmlChar*) expr.c_str(), c);
        // Evaluate node in the b context

```

```

c->node = b;
xmlXPathObjectPtr bn = xmlXPathEvalExpression((xmlChar*) expr.c_str(), c);
double r;
if (numeric)
{
    // Compare the two nodes numerically
    double d1 = xmlXPathCastToNumber(an);
    double d2 = xmlXPathCastToNumber(bn);
    r = d1 - d2;
}
else
{
    // Compare the two nodes lexicographically
    xmlChar* s1 = xmlXPathCastToString(an);
    xmlChar* s2 = xmlXPathCastToString(bn);
    r = xmlStrcmp(s1, s2);
    xmlFree(s1);
    xmlFree(s2);
}
xmlXPathFreeObject(an);
xmlXPathFreeObject(bn);
// Reverse the sort order if required
if (r < 0)
    return !reverse;
else
    return reverse;
}
}

int main(int argc, char* argv[])
{
    Args args(argc, argv);

    vector<string> source;

    parse_commandline(args, source, context, node, expr, numeric, reverse);

    if (node.empty() && expr.empty())
        node = DEFAULT_NODE_XPATH;

    xmlSubstituteEntitiesDefault(1);
    xmlLoadExtDtdDefaultValue = 1;

    if (source.empty())
        source.push_back("-");

    // First thing to do is cat the source documents together.
    // If the default cat behaviour is unacceptable, the user should cat the
    // documents himself before running sort.

    // Read in the destination document
    xmlDocPtr xml_dest = xmlParseFile(source[0].c_str());

    // Evaluate the XPath expression to find the destination document's
    // insertion point
    xmlXPathContextPtr xml_dest_context = xmlXPathNewContext(xml_dest);

    xmlXPathObjectPtr insertion_point_nodeset = xmlXPathEval(
        (const xmlChar*) "/*", xml_dest_context);
    if (insertion_point_nodeset == NULL)
        error("Destination XPath must be valid");
    if (insertion_point_nodeset->type != XPATH_NODESET)
        error("Destination XPath must evaluate to a nodeset");
    if (insertion_point_nodeset->nodesetval == 0 ||
        insertion_point_nodeset->nodesetval->nodeNr == 0)
        error("Destination XPath must evaluate to a nodeset with at least one node");
    xmlNodePtr insertion_point = insertion_point_nodeset->nodesetval->nodeTab[0];

    xmlXPathFreeObject(insertion_point_nodeset);
    xmlXPathFreeContext(xml_dest_context);

    // Insert each source document's nodes into the destination document at the
    // insertion point
    for (vector<string>::const_iterator i = source.begin() + 1; i != source.end(); i++)

```

```

{
// Read in the source document
xmlDocPtr xml_source = xmlParseFile(i->c_str());

// Evaluate the XPath expression to find the source document's nodes
xmlXPathContextPtr xml_source_context = xmlXPathNewContext(xml_source);

xmlXPathObjectPtr source_noderset = xmlXPathEval(
    (const xmlChar*) "/*/node()", xml_source_context);
if (source_noderset == NULL)
    error("Source XPath must be valid");
if (source_noderset->type != XPATH_NODESET)
    error("Source XPath must evaluate to a nodeset");

// Loop through each node in the nodeset and add them to the destination
// document
for (int j = 0; j < source_noderset->nodesetval->nodeNr; j++)
    xmlAddChild(insertion_point, xmlCopyNode(
        source_noderset->nodesetval->nodeTab[j], 1));

xmlXPathFreeObject(source_noderset);
xmlXPathFreeContext(xml_source_context);
xmlFreeDoc(xml_source);
}

// Now that the documents have been cated together, the sorting can be done

// Evaluate the context XPath location
xmlXPathContextPtr xml_dest_ctx = xmlXPathNewContext(xml_dest);
xmlXPathObjectPtr context_noderset = xmlXPathEval(
    (const xmlChar*) context.c_str(), xml_dest_ctx);

// Make sure all context nodes have the same parent
xmlNodePtr parent;
int num = context_noderset->nodesetval->nodeNr;
if (num > 0)
{
    parent = context_noderset->nodesetval->nodeTab[0]->parent;
    for (int i = 1; i < num; i++)
        if (parent != context_noderset->nodesetval->nodeTab[i]->parent)
            error("Context nodes must all have the same parent");

    xmlNodePtr pos[num];
    xmlNodePtr n = parent->children;
    for (int i = 0; i < num && n; n = n->next)
        if (context_noderset->nodesetval->nodeTab[i] == n)
            pos[i++] = n;

// bubble sort (stable!)
for (int i = 0; i < num; i++)
{
    for (int j = 0; j < num; j++)
        if (i != j)
            if (lt(pos[i], pos[j], xml_dest_ctx))
            {
                // swap
                xmlNodePtr no = pos[i];
                pos[i] = pos[j];
                pos[j] = no;

                if (parent->children == pos[i])
                {
                    parent->children = pos[j];
                }
                else if (parent->children == pos[j])
                {
                    parent->children = pos[i];
                }
                if (parent->last == pos[i])
                {
                    parent->last = pos[j];
                }
                else if (parent->last == pos[j])
                {

```

```

        parent->last = pos[i];
    }

    pos[i]->prev->next = pos[j];
    pos[i]->next->prev = pos[j];
    pos[j]->prev->next = pos[i];
    pos[j]->next->prev = pos[i];

    xmlNodePtr next = pos[i]->next;
    xmlNodePtr prev = pos[i]->prev;
    pos[i]->next = pos[j]->next;
    pos[i]->prev = pos[j]->prev;
    pos[j]->next = next;
    pos[j]->prev = prev;
}

}

// Write result document
xmlSaveFormatFile("-", xml_dest, 0);
}

xmlFreeDoc(xml_dest);
xmlCleanupParser();
}

```