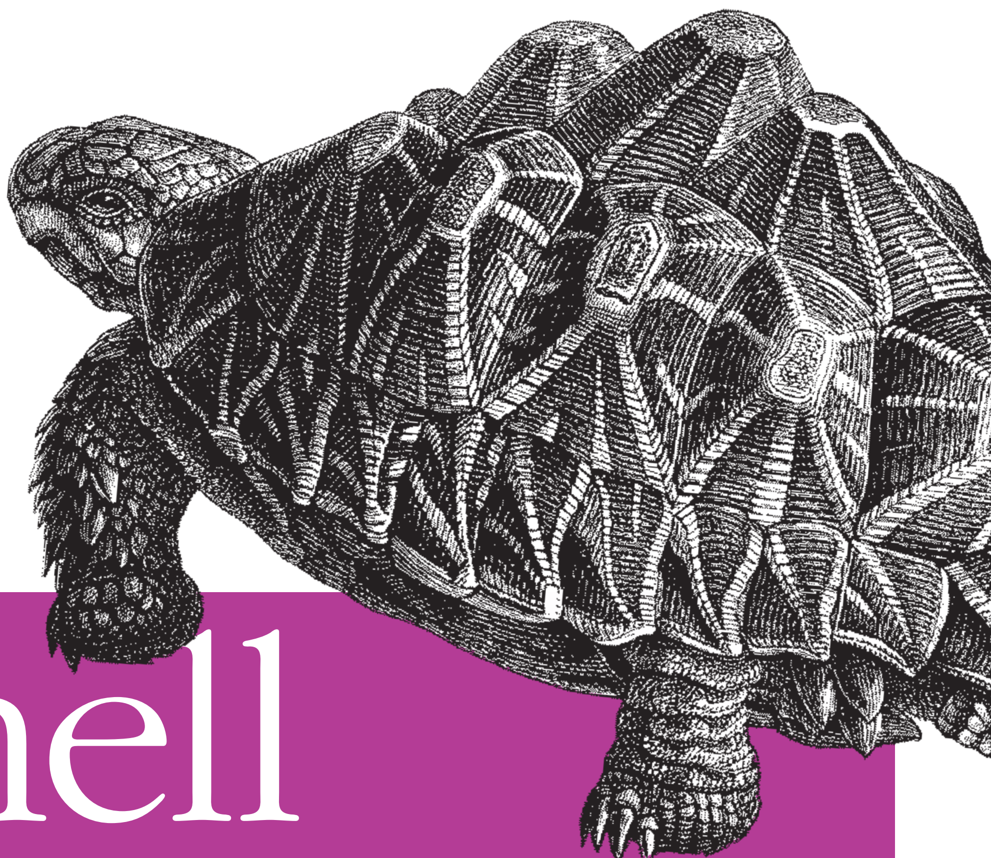
Automate Your Unix Tasks



Classic

Shell Scripting

O'REILLY®

Arnold Robbins & Nelson H. F. Beebe

CHAPTER 5

Pipelines Can Do Amazing Things

In this chapter, we solve several relatively simple text processing jobs. What's interesting about all the examples here is that they are scripts built from simple pipelines: chains of one command hooked into another. Yet each one accomplishes a significant task.

When you tackle a text processing problem in Unix, it is important to keep the Unix tool philosophy in mind: ask yourself how the problem can be broken down into simpler jobs, for each of which there is already an existing tool, or for which you can readily supply one with a few lines of a shell program or with a scripting language.

5.1 Extracting Data from Structured Text Files

Most administrative files in Unix are simple flat text files that you can edit, print, and read without any special file-specific tools. Many of them reside in the standard directory, `/etc`. Common examples are the password and group files (`passwd` and `group`), the filesystem mount table (`fstab` or `vfstab`), the hosts file (`hosts`), the default shell startup file (`profile`), and the system startup and shutdown shell scripts, stored in the subdirectory trees `rc0.d`, `rc1.d`, and so on, through `rc6.d`. (There may be other directories as well.)

File formats are traditionally documented in Section 5 of the Unix manual, so the command `man 5 passwd` provides information about the structure of `/etc/passwd`.*

Despite its name, the password file must always be publicly readable. Perhaps it should have been called the user file because it contains basic information about every user account on the system, packed together in one line per account, with fields separated by colons. We described the file's format in "Text File Conventions" [3.3.1]. Here are some typical entries:

```
jones:*:32713:899:Adrian W. Jones/OSD211/555-0123:/home/jones:/bin/ksh
dorothy:*:123:30:Dorothy Gale/KNS321/555-0044:/home/dorothy:/bin/bash
```

* On some systems, file formats are in Section 7; thus, you might need to use `man 7 passwd` instead.

```
toto:*:1027:18:Toto Gale/KNS322/555-0045:/home/toto:/bin/tcsh
ben:*:301:10:Ben Franklin/OSD212/555-0022:/home/ben:/bin/bash
jhancock:*:1457:57:John Hancock/SIG435/555-0099:/home/jhancock:/bin/bash
betsy:*:110:20:Betsy Ross/BMD17/555-0033:/home/betsy:/bin/ksh
tj:*:60:33:Thomas Jefferson/BMD19/555-0095:/home/tj:/bin/bash
george:*:692:42:George Washington/BST999/555-0001:/home/george:/bin/tcsh
```

To review, the seven fields of a password-file entry are:

1. The username
2. The encrypted password, or an indicator that the password is stored in a separate file
3. The numeric user ID
4. The numeric group ID
5. The user's personal name, and possibly other relevant data (office number, telephone number, and so on)
6. The home directory
7. The login shell

All but one of these fields have significance to various Unix programs. The one that does not is the fifth, which conventionally holds user information that is relevant only to local humans. Historically, it was called the gecos field, because it was added in the 1970s at Bell Labs when Unix systems needed to communicate with other computers running the General Electric Comprehensive Operating System, and some extra information about the Unix user was required for that system. Today, most sites use it just to record the personal name, so we simply call it the name field.

For the purposes of this example, we assume that the local site records extra information in the name field: a building and office number identifier (OSD211 in the first sample entry), and a telephone number (555-0123), separated from the personal name by slashes.

One obvious useful thing that we can do with such a file is to write some software to create an office directory. That way, only a single file, `/etc/passwd`, needs to be kept up-to-date, and derived files can be created when the master file is changed, or more sensibly, by a `cron` job that runs at suitable intervals. (We will discuss `cron` in "crontab: Rerun at Specified Times" [13.6.4].)

For our first attempt, we make the office directory a simple text file, with entries like this:

```
Franklin, Ben                     •OSD212•555-0022
Gale, Dorothy                     •KNS321•555-0044
...
```

where • represents an ASCII tab character. We put the personal name in conventional directory order (family name first), padding the name field with spaces to a

convenient fixed length. We prefix the office number and telephone with tab characters to preserve some useful structure that other tools can exploit.

Scripting languages, such as `awk`, were designed to make such tasks easy because they provide automated input processing and splitting of input records into fields, so we could write the conversion job entirely in such a language. However, we want to show how to achieve the same thing with other Unix tools.

For each password file line, we need to extract field five, split it into three subfields, rearrange the names in the first subfield, and then write an office directory line to a sorting process.

`awk` and `cut` are convenient tools for field extraction:

```
... | awk -F: '{ print $5 }' | ...
... | cut -d: -f5 | ...
```

There is a slight complication in that we have two field-processing tasks that we want to keep separate for simplicity, but we need to combine their output to make a directory entry. The `join` command is just what we need: it expects two input files, each ordered by a common unique key value, and joins lines sharing a common key into a single output line, with user control over which fields are output.

Since our directory entries contain three fields, to use `join` we need to create three intermediate files containing the colon-separated pairs *key:person*, *key:office*, and *key:telephone*, one pair per line. These can all be temporary files, since they are derived automatically from the password file.

What key do we use? It just needs to be unique, so it could be the record number in the original password file, but in this case it can also be the username, since we know that usernames are unique in the password file and they make more sense to humans than numbers do. Later, if we decide to augment our directory with additional information, such as job title, we can create another nontemporary file with the pair *key: jobtitle* and add it to the processing stages.

Instead of hardcoding input and output filenames into our program, it is more flexible to write the program as a *filter* so that it reads standard input and writes standard output. For commands that are used infrequently, it is advisable to give them descriptive, rather than short and cryptic, names, so we start our shell program like this:

```
#! /bin/sh
# Filter an input stream formatted like /etc/passwd,
# and output an office directory derived from that data.
#
# Usage:
#       passwd-to-directory < /etc/passwd > office-directory-file
#       ypcat passwd | passwd-to-directory > office-directory-file
#       niscat passwd.org_dir | passwd-to-directory > office-directory-file
```

Since the password file is publicly readable, any data derived from it is public as well, so there is no real need to restrict access to our program's intermediate files. However, because all of us at times have to deal with sensitive data, it is good to develop the programming habit of allowing file access only to those users or processes that need it. We therefore reset the umask (see "Default permissions" in Appendix B) as the first action in our program:

```
umask 077                                    Restrict temporary file access to just us
```

For accountability and debugging, it is helpful to have some commonality in temporary filenames, and to avoid cluttering the current directory with them: we name them with the prefix `/tmp/pd.`. To guard against name collisions if multiple instances of our program are running at the same time, we also need the names to be unique: the process number, available in the shell variable `$$`, provides a distinguishing suffix. (This use of `$$` is described in more detail in Chapter 10.) We therefore define these shell variables to represent our temporary files:

```
PERSON=/tmp/pd.key.person.$$                 Unique temporary filenames
OFFICE=/tmp/pd.key.office.$$
TELEPHONE=/tmp/pd.key.telephone.$$
USER=/tmp/pd.key.user.$$
```

When the job terminates, either normally or abnormally, we want the temporary files to be deleted, so we use the `trap` command:

```
trap "exit 1"                                     HUP INT PIPE QUIT TERM
trap "rm -f $PERSON $OFFICE $TELEPHONE $USER" EXIT
```

During development, we can just comment out the second `trap`, preserving temporary files for subsequent examination. (The `trap` command is described in "Trapping Process Signals" [13.3.2]. For now, it's enough to understand that when the script exits, the `trap` command arranges to automatically run `rm` with the given arguments.)

We need fields one and five repeatedly, and once we have them, we don't require the input stream from standard input again, so we begin by extracting them into a temporary file:

```
awk -F: '{ print $1 ":" $5 }' > $USER          This reads standard input
```

We make the *key:person* pair file first, with a two-step `sed` program followed by a simple line sort; the `sort` command is discussed in detail in "Sorting Text" [4.1].

```
sed -e 's=/.*==' \
    -e 's=^\([^:]*\):\(.*\) \([^ ]*\)=\1:\3, \2=' <$USER | sort >$PERSON
```

The script uses `=` as the separator character for `sed`'s `s` command, since both slashes and colons appear in the data. The first edit strips everything from the first slash to the end of the line, reducing a line like this:

```
jones:Adrian W. Jones/OSD211/555-0123        Input line
```

to this:

```
jones:Adrian W. Jones                           Result of first edit
```

The second edit is more complex, matching three subpatterns in the record. The first part, `^\([^:]*\)`, matches the username field (e.g., `jones`). The second part, `\(.*\)❒`, matches text up to a space (e.g., `Adrian❒W.❒`; the ❒ stands for a space character). The last part, `\([^❒]*\)`, matches the remaining nonspace text in the record (e.g., `Jones`). The replacement text reorders the matches, producing something like `Jones,❒Adrian W.` The result of this single `sed` command is the desired reordering:

```
jones:Jones, Adrian W.                          Printed result of second edit
```

Next, we make the *key:office* pair file:

```
sed -e 's=^\([^:]*\):[^/]*/\([^/]*\)/.*$=\1:\2=' < $USER | sort > $OFFICE
```

The result is a list of users and offices:

```
jones:OSD211
```

The *key:telephone* pair file creation is similar: we just need to adjust the match pattern:

```
sed -e 's=^\([^:]*\):[^/]*/[^/]*/\([^/]*\)=\1:\2=' < $USER | sort > $TELEPHONE
```

At this stage, we have three separate files, each of which is sorted. Each file consists of the key (the username), a colon, and the particular data (personal name, office, telephone number). The `$PERSON` file's contents look like this:

```
ben:Franklin, Ben
betsy:Ross, Betsy
...
```

The `$OFFICE` file has username and office data:

```
ben:OSD212
betsy:BMD17
...
```

The `$TELEPHONE` file records usernames and telephone numbers:

```
ben:555-0022
betsy:555-0033
...
```

By default, `join` outputs the common key, then the remaining fields of the line from the first file, followed by the remaining fields of the line from the second line. The common key defaults to the first field, but that can be changed by a command-line option: we don't need that feature here. Normally, spaces separate fields for `join`, but we can change the separator with its `-t` option: we use it as `-t:`.

The join operations are done with a five-stage pipeline, as follows:

1. Combine the personal information and the office location:

   ```
   join -t: $PERSON $OFFICE | ...
   ```

The results of this operation, which become the input to the next stage, look like this:

```
ben:Franklin, Ben:OSD212
betsy:Ross, Betsy:BMD17
...
```

2. Add the telephone number:

```
... | join -t: - $TELEPHONE | ...
```

The results of this operation, which become the input to the next stage, look like this:

```
ben:Franklin, Ben:OSD212:555-0022
betsy:Ross, Betsy:BMD17:555-0033
...
```

3. Remove the key (which is the first field), since it's no longer needed. This is most easily done with `cut` and a range that says "use fields two through the end," like so:

```
... | cut -d: -f 2- | ...
```

The results of this operation, which become the input to the next stage, look like this:

```
Franklin, Ben:OSD212:555-0022
Ross, Betsy:BMD17:555-0033
...
```

4. Re-sort the data. The data was previously sorted by login name, but now things need to be sorted by personal last name. This is done with `sort`:

```
... | sort -t: -k1,1 -k2,2 -k3,3 | ...
```

This command uses a colon to separate fields, sorting on fields 1, 2, and 3, in order. The results of this operation, which become the input to the next stage, look like this:

```
Franklin, Ben:OSD212:555-0022
Gale, Dorothy:KNS321:555-0044
...
```

5. Finally, reformat the output, using `awk`'s `printf` statement to separate each field with tab characters. The command to do this is:

```
... | awk -F: '{ printf("%-39s\t%s\t%s\n", $1, $2, $3) }'
```

For flexibility and ease of maintenance, formatting should always be left until the end. Up to that point, everything is just text strings of arbitrary length.

Here's the complete pipeline:

```
join -t: $PERSON $OFFICE |
    join -t: - $TELEPHONE |
        cut -d: -f 2- |
            sort -t: -k1,1 -k2,2 -k3,3 |
                awk -F: '{ printf("%-39s\t%s\t%s\n", $1, $2, $3) }'
```

The awk `printf` statement used here is similar enough to the shell `printf` command that its meaning should be clear: print the first colon-separated field left-adjusted in a 39-character field, followed by a tab, the second field, another tab, and the third field. Here are the full results:

```
Franklin, Ben                         •OSD212•555-0022
Gale, Dorothy                         •KNS321•555-0044
Gale, Toto                            •KNS322•555-0045
Hancock, John                         •SIG435•555-0099
Jefferson, Thomas                     •BMD19•555-0095
Jones, Adrian W.                      •OSD211•555-0123
Ross, Betsy                           •BMD17•555-0033
Washington, George                    •BST999•555-0001
```

That is all there is to it! Our entire script is slightly more than 20 lines long, excluding comments, with five main processing steps. We collect it together in one place in Example 5-1.

Example 5-1. Creating an office directory

```
#! /bin/sh
# Filter an input stream formatted like /etc/passwd,
# and output an office directory derived from that data.
#
# Usage:
#       passwd-to-directory < /etc/passwd > office-directory-file
#       ypcat passwd | passwd-to-directory > office-directory-file
#       niscat passwd.org_dir | passwd-to-directory > office-directory-file

umask 077

PERSON=/tmp/pd.key.person.$$
OFFICE=/tmp/pd.key.office.$$
TELEPHONE=/tmp/pd.key.telephone.$$
USER=/tmp/pd.key.user.$$

trap "exit 1"                                HUP INT PIPE QUIT TERM
trap "rm -f $PERSON $OFFICE $TELEPHONE $USER" EXIT

awk -F: '{ print $1 ":" $5 }' > $USER

sed -e 's=/.*= =' \
    -e 's=^\([^:]*\):\(.*\) \([^ ]*\)=\1:\3, \2=' < $USER | sort > $PERSON

sed -e 's=^\([^:]*\):[^/]*/\([^/]*\)/.*$=\1:\2=' < $USER | sort > $OFFICE

sed -e 's=^\([^:]*\):[^/]*/[^/]*/\([^/]*\)=\1:\2=' < $USER | sort > $TELEPHONE

join -t: $PERSON $OFFICE |
    join -t: - $TELEPHONE |
        cut -d: -f 2- |
            sort -t: -k1,1 -k2,2 -k3,3 |
                awk -F: '{ printf("%-39s\t%s\t%s\n", $1, $2, $3) }'
```

The real power of shell scripting shows itself when we want to modify the script to do a slightly different job, such as insertion of the job title from a separately maintained *key:jobtitle* file. All that we need to do is modify the final pipeline to look something like this:

```
join -t: $PERSON /etc/passwd.job-title |          Extra join with job title
    join -t: - $OFFICE |
        join -t: - $TELEPHONE |
            cut -d: -f 2- |
                sort -t: -k1,1 -k3,3 -k4,4 |       Modify sort command
                    awk -F: '{ printf("%-39s\t%-23s\t%s\t%s\n",
                              $1, $2, $3, $4) }'   And formatting command
```

The total cost for the extra directory field is one more `join`, a change in the `sort` fields, and a small tweak in the final `awk` formatting command.

Because we were careful to preserve special field delimiters in our output, we can trivially prepare useful alternative directories like this:

```
passwd-to-directory < /etc/passwd | sort -t'•' -k2,2 > dir.by-office
passwd-to-directory < /etc/passwd | sort -t'•' -k3,3 > dir.by-telephone
```

As usual, • represents an ASCII tab character.

A critical assumption of our program is that there is a *unique key* for each data record. With that unique key, separate views of the data can be maintained in files as *key:value* pairs. Here, the key was a Unix username, but in larger contexts, it could be a book number (ISBN), credit card number, employee number, national retirement system number, part number, student number, and so on. Now you know why we get so many numbers assigned to us! You can also see that those handles need not be numbers: they just need to be unique text strings.

5.2 Structured Data for the Web

The immense popularity of the World Wide Web makes it desirable to be able to present data like the office directory developed in the last section in a form that is a bit fancier than our simple text file.

Web files are mostly written in a markup language called *HyperText Markup Language (HTML)*. This is a family of languages that are specific instances of the *Standard Generalized Markup Language (SGML)*, which has been defined in several ISO standards since 1986. The manuscript for this book was written in DocBook/XML, which is also a specific instance of SGML. You can find a full description of HTML in *HTML & XHTML: The Definitive Guide* (O'Reilly).*

* In addition to this book (listed in the Bibliography), hundreds of books on SGML and derivatives are listed at *http://www.math.utah.edu/pub/tex/bib/sgml.html* and *http://www.math.utah.edu/pub/tex/bib/sgml2000.html*.

A Digression on Databases

Most commercial databases today are constructed as *relational databases*: data is accessible as *key:value* pairs, and join operations are used to construct multicolumn tables to provide views of selected subsets of the data. Relational databases were first proposed in 1970 by E. F. Codd,[a] who actively promoted them, despite initial database industry opposition that they could not be implemented efficiently. Fortunately, clever programmers soon figured out how to solve the efficiency problem. Codd's work is so important that, in 1981, he was given the prestigious ACM Turing Award, the closest thing in computer science to the Nobel Prize.

Today, there are several ISO standards for the *Structured Query Language (SQL)*, making vendor-independent database access possible, and one of the most important SQL operations is `join`. Hundreds of books have been published about SQL; to learn more, pick a general one like *SQL in a Nutshell*.[b] Our simple office-directory task thus has an important lesson in it about the central concept of modern relational databases, and Unix software tools can be extremely valuable in preparing input for databases, and in processing their output.

[a] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, 13(6) 377–387, June (1970), and *Relational Database: A Practical Foundation for Productivity*, Communications of the ACM, 25(2) 109–117, February (1982) (Turing Award lecture).

[b] By Kevin Kline and Daniel Kline, O'Reilly & Associates, 2000, ISBN 1-56592-744-3. See also *http://www.math.utah.edu/pub/tex/bib/sqlbooks.html* for an extensive list of SQL books.

For the purposes of this section, we need only a tiny subset of HTML, which we present here in a small tutorial. If you are already familiar with HTML, just skim the next page or two.

Here is a minimal standards-conformant HTML file produced by a useful tool written by one of us:*

```
$ echo Hello, world. | html-pretty
<!-- -*-html-*- -->
<!-- Prettyprinted by html-pretty flex version 1.01 [25-Aug-2001] -->
<!-- on Wed Jan  8 12:12:42 2003 -->
<!-- for Adrian W. Jones (jones@example.com) -->

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
    <HEAD>
        <TITLE>
            <!-- Please supply a descriptive title here -->
        </TITLE>
        <!-- Please supply a correct e-mail address here -->
        <LINK REV="made" HREF="mailto:jones@example.com">
```

* Available at *http://www.math.utah.edu/pub/sgml/*.

```
    </HEAD>
    <BODY>
        Hello, world.
    </BODY>
</HTML>
```

The points to note in this HTML output are:

- HTML comments are enclosed in <!-- and -->.
- Special processor commands are enclosed in <! and >: here, the DOCTYPE command tells an SGML parser what the document type is and where to find its grammar file.
- Markup is supplied by angle-bracketed words, called *tags*. In HTML, lettercase is *not* significant in tag names: html-pretty normally uppercases tag names for better visibility.
- Markup environments consist of a begin tag, <*NAME*>, and an end tag, </*NAME*>, and for many tags, environments can be nested within each other according to rules defined in the HTML grammars.
- An HTML document is structured as an HTML object containing one HEAD and one BODY object.
- Inside the HEAD, a TITLE object defines the document title that web browsers display in the window titlebar and in bookmark lists. Also inside the HEAD, the LINK object generally carries information about the web-page maintainer.
- The visible part of the document that browsers show is the contents of the BODY.
- Whitespace is not significant outside of quoted strings, so we can use horizontal and vertical spacing liberally to emphasize the structure, as the HTML prettyprinter does.
- Everything else is just printable ASCII text, with three exceptions. Literal angle brackets must be represented by special encodings, called *entities*, that consist of an ampersand, an identifier, and a semicolon: < and >. Since ampersand starts entities, it has its own literal *entity* name: &. HTML supports a modest repertoire of entities for accented characters that cover most of the languages of Western Europe so that we can write, for example, café du bon goût to get café du bon goût.
- Although not shown in our minimal example, font style changes are accomplished in HTML with B (bold), EM (emphasis), I (italic), STRONG (extra bold), and TT (typewriter (fixed-width characters)) environments: write <B>bold phrase</B> to get **bold phrase**.

To convert our office directory to proper HTML, we need only one more bit of information: how to format a table, since that is what our directory really is and we don't want to force the use of typewriter fonts to get everything to line up in the browser display.

In HTML 3.0 and later, a table consists of a `TABLE` environment, inside of which are rows, each of them a table row (`TR`) environment. Inside each row are cells, called table data, each a `TD` environment. Notice that columns of data receive no special markup: a data column is simply the set of cells taken from the same row position in all of the rows of the table. Happily, we don't need to declare the number of rows and columns in advance. The job of the browser or formatter is to collect all of the cells, determine the widest cell in each column, and then format the table with columns just wide enough to hold those widest cells.

For our office directory example, we need just three columns, so our sample entry could be marked up like this:

```
<TABLE>
    ...
    <TR>
        <TD>
            Jones, Adrian W.
        </TD>
        <TD>
            555-0123
        </TD>
        <TD>
            OSD211
        </TD>
    </TR>
    ...
</TABLE>
```

An equivalent, but compact and hard-to-read, encoding might look like this:

```
<TABLE>
    ...
    <TR><TD>Jones, Adrian W.</TD><TD>555-0123</TD><TD>OSD211</TD></TR>
    ...
</TABLE>
```

Because we chose to preserve special field separators in the text version of the office directory, we have sufficient information to identify the cells in each row. Also, because whitespace is mostly not significant in HTML files (except to humans), we need not be particularly careful about getting tags nicely lined up: if that is needed later, `html-pretty` can do it perfectly. Our conversion filter then has three steps:

1. Output the leading boilerplate down to the beginning of the document body.
2. Wrap each directory row in table markup.
3. Output the trailing boilerplate.

We have to make one small change from our minimal example: the `DOCTYPE` command has to be updated to a later grammar level so that it looks like this:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN//3.0">
```

You don't have to memorize this: html-pretty has options to produce output in any of the standard HTML grammar levels, so you can just copy a suitable DOCTYPE command from its output.

Clearly, most of the work is just writing boilerplate, but that is simple since we can just copy text from the minimal HTML example. The only programmatic step required is the middle one, which we could do with only a couple of lines in awk. However, we can achieve it with even less work using a sed stream-editor substitution with two edit commands: one to substitute the embedded tab delimiters with </TD><TD>, and a following one to wrap the entire line in <TR><TD>...</TD></TR>. We temporarily assume that no accented characters are required in the directory, but we can easily allow for angle brackets and ampersands in the input stream by adding three initial sed steps. We collect the complete program in Example 5-2.

Example 5-2. Converting an office directory to HTML

```
#! /bin/sh
# Convert a tab-separated value file to grammar-conformant HTML.
#
# Usage:
#    tsv-to-html < infile > outfile

cat << EOFILE                         Leading boilerplate
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN//3.0">
<HTML>
    <HEAD>
        <TITLE>
            Office directory
        </TITLE>
        <LINK REV="made" HREF="mailto:$USER@`hostname`">
    </HEAD>
    <BODY>
        <TABLE>
EOFILE

sed -e 's=&=\&amp;=g' \               Convert special characters to entities
    -e 's=<=\&lt;=g' \
    -e 's=>=\&gt;=g' \
    -e 's=\t=</TD><TD>=g' \           And supply table markup
    -e 's=^.*$=           <TR><TD>&</TD></TR>='

cat << EOFILE                         Trailing boilerplate
        </TABLE>
    </BODY>
</HTML>
EOFILE
```

The << notation is called a *here document*. It is explained in more detail in "Additional Redirection Operators" [7.3.1]. Briefly, the shell reads all lines up to the delimiter following the << (EOFILE in this case), does variable and command substitution on the contained lines, and feeds the results as standard input to the command.

There is an important point about the script in Example 5-2: it is independent of the number of columns in the table! This means that it can be used to convert *any* tab-separated value file to HTML. Spreadsheet programs can usually save data in such a format, so our simple tool can produce correct HTML from spreadsheet data.

We were careful in `tsv-to-html` to maintain the spacing structure of the original office directory, because that makes it easy to apply further filters downstream. Indeed, `html-pretty` was written precisely for that reason: standardization of HTML markup layout radically simplifies other HTML tools.

How would we handle conversion of accented characters to HTML entities? We *could* augment the `sed` command with extra edit steps like `-e 's=é=é=g'`, but there are about 100 or so entities to cater for, and we are likely to need similar substitutions as we convert other kinds of text files to HTML.

It therefore makes sense to delegate that task to a separate program that we can reuse, either as a pipeline stage following the `sed` command in Example 5-2, or as a filter applied later. (This is the "detour to build specialized tools" principle in action.) Such a program is just a tedious tabulation of substitution commands, and we need one for each of the local text encodings, such as the various ISO 8859-*n* code pages mentioned in "How Are Files Named?" in Appendix B. We don't show such a filter completely here, but a fragment of one in Example 5-3 gives the general flavor. For readers who need it, we include the complete program for handling the common case of Western European characters in the ISO 8859-1 encoding with this book's sample programs. HTML's entity repertoire isn't sufficient for other accented characters, but since the World Wide Web is moving in the direction of Unicode and XML in place of ASCII and HTML, this problem is being solved in a different way, by getting rid of character set limitations.

Example 5-3. Fragment of iso8859-1-to-html program

```
#! /bin/sh
# Convert an input stream containing characters in ISO 8859-1
# encoding from the range 128..255 to HTML equivalents in ASCII.
# Characters 0..127 are preserved as normal ASCII.
#
# Usage:
#       iso8859-1-to-html infile(s) >outfile

sed \
        -e 's= =\ =g' \
        -e 's=¡=\&iexcl;=g' \
        -e 's=¢=\&cent;=g' \
        -e 's=£=\&pound;=g' \
...
        -e 's=ü=\&uuml;=g' \
        -e 's=  =\&yacute;=g' \
        -e 's= æ=\&thorn;=g' \
        -e 's=ÿ=\&yuml;=g' \
        "$@"
```

Here is a sample of the use of this filter:

```
$ cat danish                          Show sample Danish text in ISO 8859-1 encoding
Øen med åen lå i læ af én halv,äò,
og én stor ,äò, langs den græske kyst.

$ iso8859-1-to-html danish            Convert text to HTML entities
&Oslash;en med &aring;en l&aring; i l&aelig; af &eacute;n halv&oslash;,
og &eacute;n stor &oslash;, langs den gr&aelig;ske kyst.
```

5.3 Cheating at Word Puzzles

Crossword puzzles give you clues about words, but most of us get stuck when we cannot think of, say, a ten-letter word that begins with a b and has either an x or a z in the seventh position.

Regular-expression pattern matching with awk or grep is clearly called for, but what files do we search? One good choice is the Unix spelling dictionary, available as /usr/dict/words, on many systems. (Other popular locations for this file are /usr/share/dict/words and /usr/share/lib/dict/words.) This is a simple text file, with one word per line, sorted in lexicographic order. We can easily create other similar-appearing files from any collection of text files, like this:

```
cat file(s) | tr A-Z a-z | tr -c a-z\' '\n' | sort -u
```

The second pipeline stage converts uppercase to lowercase, the third replaces nonletters by newlines, and the last sorts the result, keeping only unique lines. The third stage treats apostrophes as letters, since they are used in contractions. Every Unix system has collections of text that can be mined in this way—for example, the formatted manual pages in /usr/man/cat*/* and /usr/local/man/cat*/*. On one of our systems, they supplied more than 1 million lines of prose and produced a list of about 44,000 unique words. There are also word lists for dozens of languages in various Internet archives.*

Let us assume that we have built up a collection of word lists in this way, and we stored them in a standard place that we can reference from a script. We can then write the program shown in Example 5-4.

Example 5-4. Word puzzle solution helper

```
#! /bin/sh
# Match an egrep(1)-like pattern against a collection of
# word lists.
#
# Usage:
```

* Available at *ftp://ftp.ox.ac.uk/pub/wordlists/*, *ftp://qiclab.scn.rain.com/pub/wordlists/*, *ftp://ibiblio.org/pub/docs/books/gutenberg/etext96/pgw**, and *http://www.phreak.org/html/wordlists.shtml*. A search for "word list" in any Internet search engine turns up many more.

Example 5-4. Word puzzle solution helper (continued)

```
#       puzzle-help egrep-pattern [word-list-files]

FILES="
        /usr/dict/words
        /usr/share/dict/words
        /usr/share/lib/dict/words
        /usr/local/share/dict/words.biology
        /usr/local/share/dict/words.chemistry
        /usr/local/share/dict/words.general
        /usr/local/share/dict/words.knuth
        /usr/local/share/dict/words.latin
        /usr/local/share/dict/words.manpages
        /usr/local/share/dict/words.mathematics
        /usr/local/share/dict/words.physics
        /usr/local/share/dict/words.roget
        /usr/local/share/dict/words.sciences
        /usr/local/share/dict/words.unix
        /usr/local/share/dict/words.webster
      "
pattern="$1"


egrep -h -i "$pattern" $FILES 2> /dev/null | sort -u -f
```

The FILES variable holds the built-in list of word-list files, customized to the local site. The grep option -h suppresses filenames from the report, the -i option ignores lettercase, and we discard the standard error output with 2> /dev/null, in case any of the word-list files don't exist or they lack the necessary read permission. (This kind of redirection is described in "File Descriptor Manipulation" [7.3.2].) The final sort stage reduces the report to just a list of unique words, ignoring lettercase.

Now we can find the word that we were looking for:

```
$ puzzle-help '^b.....[xz]...$' | fmt
bamboozled Bamboozler bamboozles bdDenizens bdWheezing Belshazzar
botanizing Brontozoum Bucholzite bulldozing
```

Can you think of an English word with six consonants in a row? Here's some help:

```
$ puzzle-help '[^aeiouy]{6}' /usr/dict/words
Knightsbridge
mightn't
oughtn't
```

If you don't count *y* as a vowel, many more turn up: *encryption*, *klystron*, *porphyry*, *syzygy*, and so on.

We could readily exclude the contractions from the word lists by a final filter step—egrep -i '^[a-z]+$'—but there is little harm in leaving them in the word lists.

5.4 Word Lists

From 1983 to 1987, Bell Labs researcher Jon Bentley wrote an interesting column in *Communications of the ACM* titled *Programming Pearls*. Some of the columns were later collected, with substantial changes, into two books listed in the Bibliography. In one of the columns, Bentley posed this challenge: write a program to process a text file, and output a list of the *n* most-frequent words, with counts of their frequency of occurrence, sorted by descending count. Noted computer scientists Donald Knuth and David Hanson responded separately with interesting and clever literate programs,* each of which took several hours to write. Bentley's original specification was imprecise, so Hanson rephased it this way: Given a text file and an integer *n*, you are to print the words (and their frequencies of occurrence) whose frequencies of occurrence are among the *n* largest in order of decreasing frequency.

In the first of Bentley's articles, fellow Bell Labs researcher Doug McIlroy reviewed Knuth's program, and offered a six-step Unix solution that took only a couple of minutes to develop and worked correctly the first time. Moreover, unlike the two other programs, McIlroy's is devoid of explicit magic constants that limit the word lengths, the number of unique words, and the input file size. Also, its notion of what constitutes a word is defined entirely by simple patterns given in its first two executable statements, making changes to the word-recognition algorithm easy.

McIlroy's program illustrates the power of the Unix tools approach: break a complex problem into simpler parts that you already know how to handle. To solve the word-frequency problem, McIlroy converted the text file to a list of words, one per line (`tr` does the job), mapped words to a single lettercase (`tr` again), sorted the list (`sort`), reduced it to a list of unique words with counts (`uniq`), sorted that list by descending counts (`sort`), and finally, printed the first several entries in the list (`sed`, though `head` would work too).

The resulting program is worth being given a name (`wf`, for word frequency) and wrapped in a shell script with a comment header. We also extend McIlroy's original `sed` command to make the output list-length argument optional, and we modernize the `sort` options. We show the complete program in Example 5-5.

Example 5-5. Word-frequency filter

```
#! /bin/sh
# Read a text stream on standard input, and output a list of
# the n (default: 25) most frequently occurring words and
# their frequency counts, in order of descending counts, on
```

* *Programming Pearls: A Literate Program: A* WEB program for common words, Comm. ACM **29**(6), 471–483, June (1986), and *Programming Pearls: Literate Programming: Printing Common Words*, **30**(7), 594–599, July (1987). Knuth's paper is also reprinted in his book *Literate Programming*, Stanford University Center for the Study of Language and Information, 1992, ISBN 0-937073-80-6 (paper) and 0-937073-81-4 (cloth).

Example 5-5. Word-frequency filter (continued)

```
# standard output.
#
# Usage:
#       wf [n]

tr -cs A-Za-z\' '\n' |          Replace nonletters with newlines
  tr A-Z a-z |                  Map uppercase to lowercase
    sort |                      Sort the words in ascending order
      uniq -c |                 Eliminate duplicates, showing their counts
        sort -k1,1nr -k2 |      Sort by descending count, and then by ascending word
          sed ${1:-25}q         Print only the first n (default: 25) lines; see Chapter 3
```

POSIX `tr` supports all of the escape sequences of ISO Standard C. The older X/Open Portability Guide specification only had octal escape sequences, and the original `tr` had none at all, forcing the newline to be written literally, which was one of the criticisms levied at McIlroy's original program. Fortunately, the `tr` command on every system that we tested now has the POSIX escape sequences.

A shell pipeline isn't the only way to solve this problem with Unix tools: Bentley gave a six-line `awk` implementation of this program in an earlier column* that is roughly equivalent to the first four stages of McIlroy's pipeline.

Knuth and Hanson discussed the computational complexity of their programs, and Hanson used runtime profiling to investigate several variants of his program to find the fastest one.

The complexity of McIlroy's is easy to identify. All but the `sort` stages run in a time that is linear in the size of their input, and that size is usually sharply reduced after the `uniq` stage. Thus, the rate-limiting step is the first `sort`. A good sorting algorithm based on comparisons, like that in Unix `sort`, can sort n items in a time proportional to $n \log_2 n$. The logarithm-to-the-base-2 factor is small: for n about 1 million, it is about 20. Thus, in practice, we expect `wf` to be a few times slower than it would take to just copy its input stream with `cat`.

Here is an example of applying this script to the text of Shakespeare's most popular play, *Hamlet*,† reformatting the output with `pr` to a four-column display:

```
$ wf 12 < hamlet | pr -c4 -t -w80
   1148 the          671 of          550 a           451 in
    970 and          635 i           514 my          419 it
    771 to           554 you         494 hamlet      407 that
```

* *Programming Pearls: Associative Arrays*, Comm. ACM **28**(6), 570–576, June, (1985). This is an excellent introduction to the power of associative arrays (tables indexed by strings, rather than integers), a common feature of most scripting languages.

† Available in the wonderful Project Gutenberg archives at *http://www.gutenberg.net/*.

The results are about as expected for English prose. More interesting, perhaps, is to ask how many unique words there are in the play:

```
$ wf 999999 < hamlet | wc -l
4548
```

and to look at some of the least-frequent words:

```
$ wf 999999 < hamlet | tail -n 12 | pr -c4 -t -w80
      1 yaw             1 yesterday       1 yielding        1 younger
      1 yawn            1 yesternight     1 yon             1 yourselves
      1 yeoman          1 yesty           1 yond            1 zone
```

There is nothing magic about the argument `999999`: it just needs to be a number larger than any expected count of unique words, and the keyboard repeat feature makes it easy to type.

We can also ask how many of the 4548 unique words were used just once:

```
$ wf 999999 < hamlet | grep -c '^ *1•'
2634
```

The • following the digit `1` in the `grep` pattern represents a tab. This result is surprising, and probably atypical of most modern English prose: although the play's vocabulary is large, nearly 58 percent of the words occur only once. And yet, the core vocabulary of frequently occurring words is rather small:

```
$ wf 999999 < hamlet | awk '$1 >= 5' | wc -l
740
```

This is about the number of words that a student might be expected to learn in a semester course on a foreign language, or that a child learns before entering school.

Shakespeare didn't have computers to help analyze his writing,* but we can speculate that part of his genius was in making most of what he wrote understandable to the broadest possible audience of his time.

When we applied `wf` to the individual texts of Shakespeare's plays, we found that *Hamlet* has the largest vocabulary (4548), whereas *Comedy of Errors* has the smallest (2443). The total number of unique words in the Shakespeare corpus of plays and sonnets is nearly 23,700, which shows that you need exposure to several plays to enjoy the richness of his work. About 36 percent of those words are used only once, and only one word begins with x: Xanthippe, in *Taming of the Shrew*. Clearly, there is plenty of fodder in Shakespeare for word-puzzle enthusiasts and vocabulary analysts!

* Indeed, the only word related to the root of computer that Shakespeare used is computation, just once in each of two plays, *Comedy of Errors* and *King Richard III*. "Arithmetic" occurs six times in his plays, "calculate" twice, and "mathematics" thrice.

5.5 Tag Lists

Use of the `tr` command to obtain lists of words, or more generally, to transform one set of characters to another set, as in Example 5-5 in the preceding section, is a handy Unix tool idiom to remember. It leads naturally to a solution of a problem that we had in writing this book: how do we ensure consistent markup through about 50K lines of manuscript files? For example, a command might be marked up with `<command>tr</command>` when we talk about it in the running text, but elsewhere, we might give an example of something that you type, indicated by the markup `<literal>tr</literal>`. A third possibility is a manual-page reference in the form `<emphasis>tr</emphasis>(1)`.

The `taglist` program in Example 5-6 provides a solution. It finds all begin/end tag pairs written on the same line and outputs a sorted list that associates tag use with input files. Additionally, it flags with an arrow cases where the same word is marked up in more than one way. Here is a fragment of its output from just the file for a version of this chapter:

```
$ taglist ch05.xml
...
      2 cut                              command         ch05.xml
      1 cut                              emphasis        ch05.xml <----
...
      2 uniq                             command         ch05.xml
      1 uniq                             emphasis        ch05.xml <----
      1 vfstab                           filename        ch05.xml
...
```

The tag listing task is reasonably complex, and would be quite hard to do in most conventional programming languages, even ones with large class libraries, such as C++ and Java, and even if you started with the Knuth or Hanson literate programs for the somewhat similar word-frequency problem. Yet, just nine steps in a Unix pipeline with by-now familiar tools suffice.

The word-frequency program did not deal with named files: it just assumed a single data stream. That is not a serious limitation because we can easily feed it multiple input files with `cat`. Here, however, we need a filename, since it does us no good to report a problem without telling where the problem is. The filename is `taglist`'s single argument, available in the script as `$1`.

1. We feed the input file into the pipeline with `cat`. We could, of course, eliminate this step by redirecting the input of the next stage from `$1`, but we find in complex pipelines that it is clearer to separate *data production* from *data processing*. It also makes it slightly easier to insert yet another stage into the pipeline if the program later evolves.

   ```
   cat "$1" | ...
   ```

2. We apply sed to simplify the otherwise-complex markup needed for web URLs:

   ```
   ... | sed -e 's#systemitem *role="url"#URL#g' \
             -e 's#/systemitem#/URL#' | ...
   ```

   This converts tags such as <systemitem role="URL"> and </systemitem> into simpler <URL> and </URL> tags, respectively.
3. The next stage uses tr to replace spaces and paired delimiters by newlines:

   ```
   ... | tr ' (){}[]' '\n\n\n\n\n\n\n' | ...
   ```

4. At this point, the input consists of one "word" per line (or empty lines). Words are either actual text or SGML/XML tags. Using egrep, the next stage selects tag-enclosed words:

   ```
   ... | egrep '>[^<>]+</' | ...
   ```

   This regular expression matches tag-enclosed words: a right angle bracket, followed by at least one nonangle bracket, followed by a left angle bracket, followed by a slash (for the closing tag).
5. At this point, the input consists of lines with tags. The first awk stage uses angle brackets as field separators, so the input <literal>tr</literal> is split into four fields: an empty field, followed by literal, tr, and /literal. The filename is passed to awk on the command line, where the -v option sets the awk variable FILE to the filename. That variable is then used in the print statement, which outputs the word, the tag, and the filename:

   ```
   ... | awk -F'[<>]' -v FILE="$1" \
             '{ printf("%-31s\t%-15s\t%s\n", $3, $2, FILE) }' | ...
   ```

6. The sort stage sorts the lines into word order:

   ```
   ... | sort | ...
   ```

7. The uniq command supplies the initial count field. The output is a list of records, where the fields are *count*, *word*, *tag*, *file*:

   ```
   ... | uniq -c | ...
   ```

8. A second sort orders the output by word and tag (the second and third fields):

   ```
   ... | sort -k2,2 -k3,3 | ...
   ```

9. The final stage uses a small awk program to filter successive lines, adding a trailing arrow when it sees the same word as on the previous line. This arrow then clearly indicates instances where words have been marked up differently, and thus deserve closer inspection by the authors, the editors, or the book-production staff:

   ```
   ... | awk '{
                 print ($2 == Last) ? ($0 " <----") : $0
                 Last = $2
              }'
   ```

The full program is provided in Example 5-6.

Example 5-6. Making an SGML tag list

```
#! /bin/sh -
# Read an HTML/SGML/XML file given on the command
# line containing markup like <tag>word</tag> and output on
# standard output a tab-separated list of
#
#       count word tag filename
#
# sorted by ascending word and tag.
#
# Usage:
#       taglist xml-file

cat "$1" |
  sed -e 's#systemitem *role="url"#URL#g' -e 's#/systemitem#/URL#' |
    tr ' ( ){ }[ ]' '\n\n\n\n\n\n\n' |
      egrep '>[^<>]+</' |
        awk -F'[<>]' -v FILE="$1" \
            '{ printf("%-31s\t%-15s\t%s\n", $3, $2, FILE) }' |
          sort |
            uniq -c |
              sort -k2,2 -k3,3 |
                awk '{
                        print ($2 == Last) ? ($0 " <----") : $0
                        Last = $2
                     }'
```

In "Functions" [6.5], we will show how to apply the tag-list operation to multiple files.

5.6 Summary

This chapter has shown how to solve several text processing problems, none of which would be simple to do in most programming languages. The critical lessons of this chapter are:

- Data markup is extremely valuable, although it need not be complex. A unique single character, such as a tab, colon, or comma, often suffices.
- Pipelines of simple Unix tools and short, often inline, programs in a suitable text processing language, such as `awk`, can exploit data markup to pass multiple pieces of data through a series of processing stages, emerging with a useful report.
- By keeping the data markup simple, the output of our tools can readily become input to new tools, as shown by our little analysis of the output of the word-frequency filter, `wf`, applied to Shakespeare's texts.
- By preserving some minimal markup in the output, we can later come back and massage that data further, as we did to turn a simple ASCII office directory into a web page. Indeed, it is wise never to consider any form of electronic data as

final: there is a growing demand in some quarters for page-description languages, such as PCL, PDF, and PostScript, to preserve the original markup that led to the page formatting. Word processor documents currently are almost devoid of useful logical markup, but that may change in the future. At the time of this writing, one prominent word processor vendor was reported to be considering an XML representation for document storage. The GNU Project's `gnumeric` spreadsheet, the Linux Documentation Project,* and the OpenOffice.org† office suite already do that.

- Lines with delimiter-separated fields are a convenient format for exchanging data with more complex software, such as spreadsheets and databases. Although such systems usually offer some sort of report-generation feature, it is often easier to extract the data as a stream of lines of fields, and then to apply filters written in suitable programming languages to manipulate the data further. For example, catalog and directory publishing are often best done this way.

* See *http://www.tldp.org/*.

† See *http://www.openoffice.org/*.