by Peter Collinson, Hillside Systems

Small Text Databases

My brother-in-law died suddenly in February. He had no partner and so we've been unexpectedly stuck with the task of getting his affairs in order. He was in love with railways, and spent much of his spare time traveling around the United Kingdom and Ireland on the many pairs of iron tracks that exist in these islands. His house is stuffed with books about railways, so I decided to create a catalog of these tomes that we can send to second-hand booksellers.

The first rule of any such project is to see what exists on your systems that may provide an "off-the-shelf" solution. On UNIX, there is a database mechanism accessed by `refer` that is intended to provide citations to papers. It enables authors to access a central database to find the full details of a particular paper. The system allows a citation to be automatically included in the `nroff` source of the paper or book the author is writing. This system is bendable for other uses, such as address lists, but it wasn't a good fit for my project. I decided to start from scratch.

The first problem with such a project is data capture. I phoned a second-hand bookseller and asked what information he required. He said he needed the title, author and publisher. I decided to add the ISBN. I had to make a second pass over the books when another bookseller said he needed to know whether the book was bound with a hard or soft cover, since this is important pricing information. It turns out that the second-hand market doesn't use the ISBN at all.

I now had an idea of the data to be captured and I knew that I was going to process the data using the standard set of UNIX tools. What next?

UNIX deals with text databases pretty well but, in general, a "standard" database contains one record per line, with the fields within the records separated by a unique character. It can be very error-prone to create this kind of file by hand with a text editor, it's not always clear which field you are entering.

Creating such a database is best done with a data entry script that prompts you for the contents of a specific field and allows you to enter the data for that field. When the script terminates, the complete record can be written. However, mistakes will inevitably be made in data entry. Errors are usually spotted after you've hit Return to terminate the input of a specific field. So it's prudent to build editing capabilities into this type of script.

I decided that because I was going to be doing the data entry, I could use a text editor. I would simply create a text file that consists of records separated by blank lines. Each field in the record starts with some identifiable text that acts as a prompt and a tag for the data. I'd worry about creating the UNIX single-line record file later. I created a template file:

```
Title:
Author:
Publisher:
ISBN:
Cover: H
```

(Incidentally, I like to put a single space after the initial colon because the file then looks tidier.) I spent some time communing with GNU `emacs` (which I am beginning to use after a delay of many years) and taught it to copy the last record in the file, clearing the "ISBN" field and resetting the "Cover" field. I created a new record from the last one by typing a chorded keystroke, which also positioned the cursor at the start of the data in the "Title" field. I also convinced `emacs` that the Tab key should position the text cursor in the next field down, placed just after the colon and character space that exists on the line.

Three long days, 1,200 miles of driving and 850 books later, I had a catalog of the books.

Cleaning the Data

The next stage is to check the data is clean. I want to make sure only a single blank line separates each record and there is no trailing white space (tabs or spaces) in the file that might get in the way of processing. I'd also like to make sure each record has the correct number of fields. I am fairly confident the fields are in the correct order, but checking that I have five fields per record tells me that two records have not been joined together by simply omitting the blank line acting as a separator.

One temptation with this type of job is to simply hack on the source files using an editor, because it's a one-off task. Well, one-off tasks are usually done at least twice and sometimes a few more times than that, so I generally feel it's worthwhile to create a small script that does the task for you. The script can then be reused when that one-off job needs to be redone.

Perl is actually very good at this kind of cleanup operation; you can read the whole file into one string and then apply a couple of pattern-matching commands to clean the file. If you don't have access to Perl, or want to use the standard UNIX tools, then you'll probably end up creating a shell script that uses `sed` and `awk`. The scripts below assume you are using either the Bourne shell (`sh`), Korn shell (`ksh`) or GNU's Bourne-again shell (`bash`).

To clean the spaces from the file, I tend to use `sed`:

```
sed -e 's/[<SP><TAB>]*$//' file > newfile
```

(You should replace `<SP>` with a real character space and `<TAB>` with a real tab character.) The `sed` command reads the file one line at a time, performing the substitute command on each line. The new text field at the end of the `s` command is empty, so the command looks for either a space or a tab (`[<SP><TAB>]`) repeated several times (`*`) until the end of the line is reached (`$`) and will delete any matched data that is found.

Incidentally, some shells won't allow you to type a tab character into an interactive invocation because it is used for file name completion. I'm assuming that the commands are being typed into a file and then executed. When using small command files for complex `sed` and `awk` programs, I'll often place the commands into a shell variable:

```
sedprog='s/[<SP><TAB>]*$//'
sed -e "$sedprog" file > newfile
```

which means you can split the command invocation from the command specification. The double quotes around `$sedprog` are important.

Dealing with Blank Lines

Getting rid of trailing spaces is easy. But how do we compress multiple empty lines into a single empty line signifying the end of the record? Well, to be frank, I was stumped by this. The `sed` manual page for Solaris contains a lengthy example of multiple-line suppression, but I was convinced there should be a better way. I decided to use `awk`:

```
oneblank='BEGIN {  blanks=0  }
/^$/  {  blanks++; next;  }
      {  if (blanks != 0) printf "\n";
         blanks = 0;
         print $0;
      }
END   { printf "\n";  }'
awk "$oneblank" file
```

If you have not come across `awk` before, then this might frighten you. However, it's quite easy to understand. Honest. The `awk` command earns its living by reading a data file one line at a time and applying a program to each line. The `awk` program can consist of several lines, each starting with a pattern, followed by a set of statements in curly braces. The statements are executed if the pattern matches the line that has just been read. `BEGIN` and `END` are special patterns that are executed just before a program is run and just after, respectively.

Our program above looks for empty lines using the standard regular expression idiom of `/^$/`. When empty lines are found, we count them by incrementing the `blanks` variable, which we carefully set to zero when the program starts. Actually, presetting the variable isn't strictly needed because `awk` ensures that variables start with a zero value. However, it's good practice in other languages that don't act in such a benign way, and so I tend to include the statement.

After we've incremented the `blanks` variable, we invoke the `next` statement, which skips to the end of the script, reads the next line from the input and starts processing it. If we didn't use `next`, then the remainder of the script would be applied to all empty lines because the next chunk of script has no selection pattern.

With no pattern, silence gives consent and the next code section in the curly braces will be executed for every non-blank input line.

First, we look to see if any blank lines have been found; if so, we print a single newline to create an end-of-record indicator. We have to use the formatted print statement `printf` to force the output of a single newline. Not forgetting to reset the `blank` counter to zero, we print the whole input line. The magic `$0` variable in `awk` contains the entire line that is being processed at that point.

Finally, we cope with one of the two difficult boundary conditions: the end of the file. We'd like to ensure that a blank line terminates the last record in the file, so that an end-of-record marker is placed at the end of the file. Making sure that the last line is blank is easy: We print a newline character at the end of the file.

The other boundary condition we have to think about is what happens at the start of the file. If the original file starts with one or more blank lines, then our processed version will start with one. This will be inconvenient. However, this condition is easy to establish, we simply check that the source file begins with the text that is the start of the first record.

Notice that the `awk` script is relying on the result of the previous space-stripping script; we know that blank lines really are empty and don't contain any invisible white space. Also, we don't actually need to count blank lines. We could use a switch, setting `blanks` to one when we find a blank line.

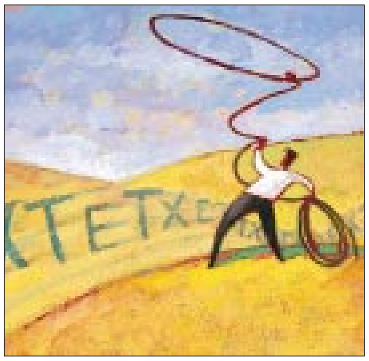
We have one further piece of checking to do. We would like to ensure that each record contains exactly five lines. Because we are stepping through the file in the script above, it seems natural to extend the script to do that. When we find a blank line, we can check that it has been preceded by five active lines:

```
oneblank='BEGIN  {  blanks=0; rct = 0 }
/^$/  {  if (rct != 0 && rct != 5) {
            printf "Record length error \
             at line %d\n", NR > "/dev/tty"
        }
        rct = 0;
        blanks++; next;
     }
     {  if (blanks != 0) printf "\n";
        blanks = 0;
        rct++;
        print $0;
     }
END  {  printf "\n";  }'
awk "$oneblank" file
```

Although this may seem complex, there is actually very little here that's new. I am using the `rct` variable to count the number of lines in each record, in precisely the same way I used `blanks` to count the number of blank lines. I check the value of `rct`. If it holds five, then all is well. If its value is zero, then we are processing a second or third blank line, and again all is well.

If `rct` contains any other value, then we have a problem and print an error message. The formatted print statement will output the string replacing `%d` with the value of the `NR` variable. The `NR` variable is maintained by `awk` and holds the number of records processed to date. This invocation of `awk` is treating each input line as a record, so `NR` contains the line number in the source file. We can use this line number to find and fix any problem in the source file. Incidentally, I've split the argument string for the `printf` statement for printing. You should join the lines together if you want to try out this script; as it stands `awk` will complain.

There's one other piece of magic. I am printing the error message to the user's terminal (`> "/dev/tty"`) rather than to the output file. This ensures the user will see this error message, and it won't be simply added to the output file causing further confusion.



Creating a Single-Line Record

We can now guarantee that we have clean data. The file can be processed to remove trailing spaces, and we can check that each record contains five lines. So all the inconsistencies that may have been introduced by originating the file with a text editor can be eliminated. We can now move to the next stage of removing the prompts from the file and compressing each record onto a single line. We'll need to identify a character that doesn't appear in the data to act as a field separator. I'm using the vertical bar character in the examples below.

To join together each of the lines in the data, we tell `awk` that it should use a specific record and field separator. You can do this from the command line, but I am doing it at the start of the `awk` program itself:

```
combine='BEGIN { FS="0; RS="" }
       { printf "%s|%s|%s|%s|%s\n",
        $1,$2,$3,$4,$5
       }'
```

Again, I've wrapped the line for printing. The first line of this script sets the field separator to the end of each line, shown by the newline character. Because of this setting, `awk` will see an empty line as a null string, and we set the end-of-record marker to the null string to show this. When this script is run, `awk` will separate the file records using the blank line that we have carefully created as the end of the record, and each line before that will form a field in the record, addressed in turn by the `$1`…`$5` syntax. I hope now you understand my concern with ensuring that the file ended in a blank line; otherwise, `awk` will not have seen an end-of-record indicator for the last record on the file.

At the end of every record, `awk` will print a single line, where the five lines or fields in the record are joined into one, each separated by a vertical bar character. The `%s` character in the formatted print statement tells `printf` to print a string.

Are we done? No. There is one final step. We must remove the prompts from the data. Running the scripts above on the template file will give us a single record that looks like this:

```
Title:|Author:|Publisher:|ISBN:|Cover: H
```

We no longer need these prompts because we are now deducing the meaning of a field by the position of that field in the record. Deleting this prompt information is a job for `sed`:

```
delprompt='s/\|[a-zA-Z]*:[ ]*/|/g
   s/^[a-zA-Z]*:[ ]*//'
```

Again this may seem a little scary, but it's easy really. The first editing statement does the bulk of the work. We use the substitute command to look for a regular expression and replace it with new text. We use the vertical bar to "anchor" the search; essentially, we look along the line for a vertical bar, a word, a colon and an optional space, and when it is found, we replace what we have matched with a vertical bar.

The elements to be matched are as follows:

`|`	– A vertical bar. This needs escaping because the vertical bar character is interpreted as "alternate expression" by `sed`'s regular expression parser.
`[a-zA-Z]*`	– A word, which is either "a to z" or "A to Z," repeated as many times as we need it.
`:`	– A colon.
`[ ]*`	– An optional space. Actually, the square brackets are not needed, but they make the space stand out as being something significant, so I often write a specific space character like this in regular expressions. The star (`*`) means that a match will be made when we find a space repeated zero or more times; so this idiom matches nothing, or one or more spaces.

Note that the word match above will also match nothing. I've paid no attention to dealing with this problem, because I know the prompt is always there in my source data.

The `g` at the end of the first expression tells `sed` to repeat this operation along the line until no further matches are found. The second command to `sed` picks up and deletes the `Title:` entry that appears at the start of each line; because there is no vertical bar at the start of the line, the first statement won't match. We use the caret anchor (`^`) here to mean the start of the line.

Well, that all looks good, so we can now combine all the various stages together in one pipeline:

```
sed -e "$sedprog" |
awk "$oneblank" |
awk "$combine" |
sed -e "$delprompt"
```

If we place this in a file called `cleanfile`, we can then say

```
sh cleanfile < booklist > booksingle
```

What Next?

Well, the data is now in a form that's accessible by a range of UNIX tools. We can print it using `troff` (or `groff`) by inserting the data into a file and inserting it into the following:

```
.TS
tab(|);
l l l l l.
<insert data here>
.TE
```

The `.TS` and `.TE` macros are used by the `tbl` program to generate a table. Actually, it's somewhat more complicated than this, so get help if you are not up to speed on `troff`.

We probably want to sort the data before printing it, and the `sort` command can deal with the output file simply. For example,

```
$ sort -t '|' booksingle
```

will use the vertical bar as a field separator, and sort using the fields left to right. The vertical bar needs quoting to get it past the shell. I wanted to sort the file into publisher, then title and author order, and had to use a more complicated `sort` command:

```
$ sh cleanfile < booklist |
     sort -t '|' +2 -3 +0 -2
```

The above command tells the `sort` program to order first by the Publisher field, then by Title and then by Author. It's easiest to think that numbers refer to the separators between the fields:

```
0    1    2      3
 Title|Author|Publisher|ISBN
```

So `+2` says "start ordering after separator 2" and `-3` means "stop ordering after separator 3." We are sorting alphabetically depending on the Publisher field. If the Publisher fields are equal, we start ordering again after (notional) separator 0 and stop after separator 2, so we then sort by Title and then Author.

Further Reading

I've used *sed & awk* by Dale Dougherty and Arnold Robbins (published by O'Reilly and Associates Inc., Second Edition, March 1997, ISBN 1-56592-225-5) as source material for this article. I have the first edition, but the book is now in its second edition.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* pc@cpg.com.