Data manipulation in perl

Douglas M. Bates
Department of Statistics
University of Wisconsin – Madison *

ABSTRACT

We are fortunate to have many different tools – statistical packages, spreadsheets, or database systems – for analysis and presentation of data. But often the first step in an analysis is manipulating or massaging the original data into a form that can be read by the package. Experienced Unix[a] users may use **sed**, **awk**, or shell scripts for this. Recently a more powerful language, **perl**, has been made freely available. It provides a superset of the capabilities of **sed**, **awk**, and **sh** as a scripting language. We describe its use for simple data manipulation and for querying a membership directory. Finally we outline a more complicated query program in **perl** for the *Current Index to Statistics*.

Keywords: sed; awk; shell scripts; database query; Current Index

[a] Unix is a registered trademark of Unix System Laboratories

1. INTRODUCTION

Often the first step in a data analysis consists of manipulating the raw data into a form that can be read by a statistical package or database system. For a single analysis on a small data set, this manipulation can be carried out with a text editor but editing such files by hand becomes tedious and error-prone when the data set is large or when there are many data sets to process. It then becomes worthwhile to create a special purpose program or script to perform the manipulation. On Unix systems this will often be done with shell scripts that use the stream editor, **sed**, or the **awk** programming language (Aho *et al.*, 1988).

Recently Larry Wall of Jet Propulsion Laboratories created a language called **perl** — the Practical Extraction and Report Language. This powerful language for creating scripts provides the facilities of **sed**, **awk**, and the standard shells. It simplifies writing data manipulation scripts because you only have to learn one language instead of three or four. It also provides capabilities not in any of these other languages.

In section 2. we give a brief outline of the **perl** language and follow that with a couple of simple examples in sections 3. and 4. In section 5. we describe some of the advantages of **perl** and conclude with discussion of two larger applications that are more in the line of tools to enhance statistics research than statistical applications. These are query programs for the Joint Statistical Directory and the *Current Index to Statistics* databases.

2. A BRIEF OUTLINE OF PERL

The source code for **perl** is freely available via anonymous ftp on the Internet from sites such as prep.ai.mit.edu (in the directory pub/gnu) or ftp.uu.net (in the directory pub/languages/perl). It can be straightforwardly installed on most workstations. The source file contains the source for an extensive manual entry (about 90 pages long) that describes the syntax and use of the language. A more complete reference is the book "Programming perl" (Wall and Schwartz, 1990).

The simplest uses of **perl** involve reading one or more text files a line at a time, changing the line in some fashion, and sending the result to an output file. Usually, the **perl** program is stored in a file but simple, "one-liner" applications can be written on the command line after a `-e` flag as in **sed** and **awk**.

Three types of variables are used in **perl**: scalars, whose names must begin with the `$` character; arrays, whose names must begin with `@`; and associative arrays whose names must begin with `%`. As in **awk**, a scalar can be either a numeric value or a character string, depending on context. There are many special variables in **perl**. The most important is `$_` which is the default argument for many functions. Its value is usually the contents of the current line.

Algorithms in **perl** are expressed with functions and control structures. The syntax is very rich so we will introduce specific functions and control structures as we need them.

A **perl** program to process one of more text files usually has the form

```
#!/usr/bin/perl
# initialize any variables
while (<>) {
   chop;
     # process the line
     # print the result
}
# final processing if needed
```

As in shell scripts, the `#` character introduces comments. The first line here is a special comment that allows the name of the script to be used as a command. Most shells for Unix systems adopt the convention that a file with the execute bit set (with `chmod +x`) and beginning with the characters `#!` is treated as input for the program specified immediately afterwards. One set of options can also be specified on this line. We assume that the **perl** compiler is stored as `/usr/bin/perl` or linked to that name.

The `while` loop is testing for the availability of another line

* This research supported by the National Science Foundation under research grant DMS-9005904.

to process. Generally, reading a line from a file handle in **perl** is indicated with broken brackets as in

```
$this_line = <STDIN>;
```

When no file handle is specified, the convention is to check for arguments specified on the command line, treat them as names of files, and try to open them for reading. If there are no arguments (actually if the array `@ARGV` is empty the first time a line is requested), the file handle `STDIN` is used instead. This reproduces the behaviour of many Unix tools that accept input from files whose names are given as arguments and otherwise accept input from the standard input stream.

To make it easier to write simple scripts, the flag `-p` to **perl** indicates that it should behave as in the loop above with the current value of `$_` being printed at the end of the loop. The flag `-n` has a similar meaning except that `$_` is not printed; the programmer must explicitly call a `print` function to produce any output.

When **perl** reads a line from a file, it retains the newline character at the end of the line. Since we often want to remove that character, it is common to "chop" the line immediately after reading it. The function `chop` removes the last character in a string. If no argument is given to `chop`, it removes the last character in `$_`.

3. Changing Field Delimiters

As a simple example, suppose you have a data file with entries like

```
5,3,001,000,692
5,2,138,000,698
5,3,146,000,698
...
```

where each line represents a single case and the values of different variables for that case are separated by commas. This is a typical organization of data as a *flat-file* where the *field delimiter* is a comma. You may find, though, that the package you are using requires the fields to be separated by blanks instead of commas. In other words, you want the file to look like

```
5 3 001 000 692
5 2 138 000 698
5 3 146 000 698
...
```

Here we develop a simple tool for changing the field delimiter in such flat files.

3.1. Commas to blanks

A person with experience using **sed** would make the change from commas to blanks with

```
sed -e 's/,/ /g' < comma.dat > blank.dat
```

The edit command, given after the `-e` flag, causes a global substitution of commas by blanks. A **perl** equivalent is almost identical

```
perl -p -e 's/,/ /g;' < comma.dat > blank.dat
```

If there are several files, say `in1.dat`, `in2.dat`, ... to be manipulated in this way, the **sed** script can be used as

```
sed -e 's/,/ /g' in*.dat > blank.dat
```

but this would direct the output from all the input files to a single file. Often we want to take each of the input files and create a separate output file for it, something we cannot do directly with **sed**. But **perl** does allow such "in-place" editing with a `-i` flag

```
perl -i -p -e 's/,/ /g;' in*.dat
```

Each file will be overwritten with its edited version. If this seems a little too risky to you, you can change the `-i` flag to

```
perl -i.bak -p -e 's/,/ /g;' in*.dat
```

Now the files will be edited in place but the original will be preserved under a name created by adding the extension `.bak` to the current name. This allows you to back out of unanticipated and undesired changes.

As an example of such an undesired change, you may discover that the file can contain comments as well as data and you want to preserve the comment lines intact. If the comments are introduced by a # character as the first character on the line, it is easy to cause those lines to be preserved by adding an `unless` clause to the substitute command as in

```
perl -i.bak -pe 's/,/ /g unless /^#/;' in*.dat
```

The pattern following the `unless` keyword is a regular expression, similar to those used in other Unix tools, that only matches lines with a # character at the beginning of the line.

3.2. White space to commas

The preceding example is rather trivial; we would expect any kind of editing tool to be able to replace commas by blanks. But consider the opposite operation of taking a file whose fields are delimited by white space and creating a comma-delimited file. It may not be sufficient to use a simple text substitution where any occurrence of a blank is replaced by a comma since there may be multiple spaces between fields or a tab character may be used instead of a space or there may be spurious spaces or tabs at the end of a line.

Here we can use some facilities of **perl** that are more like the **awk** language which splits each input line into *fields*. The simplest version of the `split` function in **perl** splits the input line into fields delimited by white space. The inverse to `split` is `join` so it is tempting to think that

```
perl -i.bak -p -e 'join(',', split);' in*.dat
```

would accomplish the desired transformation. But it turns out that this command results in the output file being a copy of the input file.

To understand why this command does not cause any changes, we have to remember that the `-p` flag means to print the current contents of `$_` at the end of the loop. This was useful to us in the earlier examples because a substitution like `s/,/ /g` is applied to `$_` in place. But `join` is a function that creates a new character string. Unless it explicitly printed or assigned to a variable, it will be discarded. We can still make the delimiter transformation a one-liner by doing the printing explicitly and changing the `-p` flag to `-n`. The command is then

```
perl -ne 'print join(',', split),"\n";' in*.dat
```

The newline character must be added explicitly since the **perl** `print` function does not implicitly add one. We indicate the

newline character by "\n". There is a subtle difference between strings enclosed by single quotes and those enclosed by double quotes. Substitution of patterns like \n, \t, ... or variable names by their corresponding special characters or string values is only done on strings enclosed by double quotes.

3.3. *A general delimiter transformer*

By this point things are getting a little complicated for a one line command. Also, we are getting close to a general method for changing delimiters — split on the current delimiter and join with the new delimiter. To exploit the generality, we should create a script, say delim.pl, and explicitly perform operations such as cycling over the input lines. Such a script is

```
#!/usr/bin/perl -i.bak
$in = ' ';
$out = ',';
while (<>) {
    chop;
    print join($out, split(/$in/)),"\n";
}
```

The **perl** code defines two character strings, $in and $out, and uses them in the split and join functions. A sample usage would now be

```
delim.pl in*.dat
```

Because we now use variables to define the input and output delimiters, we can change the behaviour of the script. One of the most convenient ways to do this is on the command line itself with flags like -i for the input delimiter, -o for the output delimiter, and -c for the pattern indicating a comment line that should be preserved intact. Since the command line is available within the **perl** script as the array @ARGV, we could write code to check for the various flags and their arguments but it is much simpler to use the standard **perl** library subroutine Getopts. This subroutine takes a string specifying which single character options are allowed and whether they take a value. It signals the presence of an option by assigning a value to the variable $opt_i for the flag -i. It also removes the flags and their arguments, if any, from the array @ARGV. Using this subroutine, the script becomes

```
#!/usr/bin/perl -i.bak
require 'getopts.pl';
&Getopts('i:o:c:');
$in = $opt_i ? $opt_i : ' ';
$out =  $opt_o ? $opt_o : ',';
while (<>) {
    if ($opt_c && /$opt_c/) {print; next;}
    chop;
    print join($out, split(/$in/)),"\n";
}
```

As in the C programming language (Kernighan and Richie, 1988), the line

```
$in = $opt_i ? $opt_i : ' ';
```

is equivalent to

```
if ($opt_i) {$in = $opt_i;} else {$in = ' ';}
```

In checking for the comment pattern we first check to see if $opt_c is defined. If we didn't do this and no pattern was given, we would be matching against the null pattern and that would always be a successful match. The program would preserve every line intact.

A sample usage to change commas to blanks but skipping lines starting with # would be

```
delim.pl -i ',' -o ' ' -c '^#' in*.dat
```

4. Fixed Format Fields

The few lines of data shown in the previous section were obtained from an administrative data file. When I receive this file it is in the form

```
Smith, John A.        LS  5    3  001   000   692
Jones, Mary E.        LS  5    3  001   000   692
Thompson, J. Walter   LS  5    3  001   000   692
Miller, Susan         LS  5    3  001   000   692
. . .
```

where the person's name occupies the first 20 characters of the line, the college occupies the next 4, the level occupies the next 2, and so on. This is an example of a fixed format data file.

This file is convenient for a person to read but not as convenient for a statistical package to read. If we wish to read such data into a package like S (Becker *et al.*, 1988; Chambers and Hastie, 1991) we have to indicate that the name should be treated as a single unit by enclosing it in quotation marks. We want to transform the file to look like

```
"Smith, John A." "LS" 5 3 001 000 692
"Jones, Mary E." "LS" 5 3 001 000 692
"Thompson, J. Walter" "LS" 5 3 001 000 692
"Miller, Susan" "LS" 5 3 001 000 692
. . .
```

Because the surname and the given names are separated by blanks and because everyone does not have the same number of given names, we cannot approach this by splitting the input line into fields delimited by white space. Instead we use the unpack function to separate the fields according to position. Here is the program that transforms the first data file into the second.

```
#!/usr/bin/perl -i.bak
$format = "A20 A4 A2 A5 A5 A6 A6";
while (<>) {
   undef @rec;
   foreach (unpack($format, $_)) {
      s/^\s+//;
      if (/^\d+$/) {push(@rec,$_);}
      else {push(@rec, '"' . $_ . '"');}
   }
   print join(' ', @rec),"\n";
}
```

The $format variable specifies the format of the line. Here an "A" followed by a field width indicates that the field is to be interpreted as an ASCII character string with trailing white space suppressed. An array, @rec, will be used to accumulate the fields before printing. Since values will be added to it by pushing them onto the end, it must be initialized to a null array at the beginning

of the loop. The input record is then split with the `unpack` function producing an array value. The `foreach` control structure cycles through the elements of the array assigning each value in turn to the variable `$_`. Assignment within the `foreach` loop does not change the value of `$_` outside the loop.

The substitution line `s/^\s+//` strips leading white space from the current field. If the field consists solely of digits, it is pushed onto the `@rec` array as it is, otherwise it is surrounded by quotation marks before being pushed onto the array. Finally the array is joined into a single string and printed.

Here the format of the line is hard-coded into the program. This program could be made more general by using the subroutine `Getopts` to pick up the format from the command line. Two other enhancements are to make the pattern for a numeric value more general and to force a conversion to a numeric type before pushing the value onto the array. Here is such a modified program

```
#!/usr/bin/perl -i.bak
require 'getopts.pl';
&Getopts('f:');
die "A format string must be specified.\n"
    unless $opt_f;
while (<>) {
   undef @rec;
   foreach (unpack($opt_f, $_)) {
      s/^\s+//;
      if (/^([-+]?\d+\.?\d*|\.\d+)$/)
         {push(@rec,$_ + 0);}
      else {push(@rec, '"' . $_ . '"');}
   }
   print join(' ', @rec),"\n";
}
```

that produces the output

```
"Smith, John A." "LS" 5 3 1 0 692
"Jones, Mary E." "LS" 5 3 1 0 692
"Thompson, J. Walter" "LS" 5 3 1 0 692
"Miller, Susan" "LS" 5 3 1 0 692
. . .
```

As you might imagine, it is not easy to formulate the pattern that determines whether a string looks like a number. Even the one given above is incomplete because it does not take into account cases where an exponent is given. A alternative approach is to recall that the only reason for the double quotes is to protect embedded white space in text fields. We can change the whole algorithm to check for the presence of white space rather than something that looks like a number. Here is the alternate version

```
#!/usr/bin/perl -i.bak
require 'getopts.pl';
&Getopts('f:');
die "A format string must be specified.\n"
   unless $opt_f;
while (<>) {
   undef @rec;
   foreach (unpack($opt_f, $_)) {
      s/^\s+//;
      $_ = '"' . $_ . '"' if /\s/;
      push(@rec, $_);
   }
   print join(' ', @rec),"\n";
}
```

5. Reasons to use perl

The examples I have given have only begun to illustrate some of the uses of **perl**. Some of the reasons that you may want to use **perl** as a data manipulation language are:

Perl is freely available. Larry Wall has chosen to make **perl** freely available under the same conditions that the Free Software Foundation applies to their software. The code is well maintained and supported. There is the documentation for the language mentioned in section 2. and there is an active Usenet newsgroup comp.lang.perl discussing the evolution of the language and programming methods in the language.

The free availability of the source code means that you can install new releases as soon as they become available and you can install it on any machine that you wish. Considering the complexity of the language, it is remarkably easy to install on a workstation. There are also precompiled versions of **perl** for VMS, for MS-DOS and for the Amiga.

Libraries and a symbolic debugger. Many library subroutines are included with **perl**. The use of `Getopts` has been illustrated in section 3. and the `look` subroutine will be illustrated in section 6.. The language has been designed to facilitate the use of libraries. The `require` and `provide` functions make it easy to load libraries. In addition, the conventions for symbol names allow you to avoid name conflicts between symbols in a library with those in the main script.

The most important library is the `perldb` library that provides a symbolic debugger for **perl**. Interestingly, the symbolic debugger is itself written in **perl**. One can use this to set breakpoints in a script, to examine or to change the values of variables, to step through the execution of a script, and so on. Once you have started to use such a tool, it is difficult to overestimate its value. For those who have gotten used to using a debugger like `gdb` or `dbx` from within GNU emacs, there is a `perldb` emacs mode as well.

Tools for conversion of existing scripts. The standard **perl** distribution includes the programs `s2p` for converting **sed** scripts to **perl**, `a2p` for converting **awk** scripts, and `find2perl` for converting **find** scripts. These conversion routines will produce a **perl** script that produces the same output as the original script although it may not be the best **perl** code for doing so. They do provide a starting point for an idiomatic **perl** implementation, though.

The **perl** compiler and the code it produces are reasonably efficient. The scripts are as easy to write as shell scripts and tend to run faster than shell scripts because they run as a single process.

Access to many low-level facilities. Many of the functions in **perl** are patterned after functions in the C library. Within **perl** you have access to system administration files (`getpwent`, etc.), dbm files (accessed as associative arrays), information in directories (`opendir, seekdir, readdir`), and file status information (`stat`). You can also use sockets and shared memory calls on machines that support them. Sockets are used to create client/server pairs that can run on different machines.

Reading and writing binary data types. If necessary, you can even read and write binary data types, either in the native byte order or in network byte order.

6. Database Searches

Within the last two years the American Statistical Association and the Institute for Mathematical Statistics have made two machine-readable databases available for purchase. The Current Index to Statistics database contains bibliographic information on books and articles of interest to statisticians. It presently contains information on more than 100,000 articles, books, or reviews from the last 12 years. It can be ordered from the IMS (send mail to ims@ucdavis.edu). The Joint Statistical Directory database contains membership information for the American Statistical Association, the Institute of Mathematical Statistics, and the Statistical Society of Canada. It can be ordered from the ASA.

6.1. Directory Lookups

The utility of both of these databases is enhanced simple query programs. Since it is much easier to write a query program for the directory, we consider that case first. The database is distributed as separate files for surnames beginning with each letter of the alphabet. Each line the the directory entry for one person and begins with "| " followed by the person's surname. Since a binary lookup on the name will be very quick in **perl**, all these files can be combined into a single file that we will call `AZ`. The file should then be sorted by

```
sort -f AZ > temp; mv temp AZ
```

because the entries in the original files are not strictly in the ASCII collating sequence. A simple query program would then look up an individual's entry by surname. Such a program is

```
#!/usr/bin/perl
require 'look.pl';
$DIR = "/usr/local/lib/ASA_members";
$Prompt = '=> ';
chdir($DIR) || die "Can't chdir to $DIR: $!\n";
open(DB, "AZ")
    || die "Can't open file ${DIR}/AZ: $!\n";
if (@ARGV) {for (@ARGV) {&do_query;}}
else {
    print $Prompt;
    while (<>) {
        chop;
        &do_query;
        print $Prompt;
    }
    print "\n";
}
sub do_query {
    $key = $_;
    &look(*DB, "| $_", 0, 1);
    while (($_ = <DB>) =~ /^\| $key/i) {print;}
}
```

Considering what the program does, it looks remarkably compact. This is because most of the work is done by the library subroutine `look` which performs a binary lookup on the given file for the a key.

Going through the program in sequence, it declares the name of the directory that holds the database and the prompt to use. The working directory is then changed to the database directory. In this case it is not really necessary to do that but in other programs where more than one file from the database needs to be accessed, it is helpful to change the directory. Another **perl** idiom is illustrated here, functions such as `chdir` and `open` return 0 if they are unsuccessful. Only in those cases will the other operand of `||` be evaluated and that causes the program to halt with an error message. (The behaviour is similar to the `stop()` function in S (Becker *et al.*, 1988).) It is common to read this as "change directory or die!".

This script can be used in two ways; the desired name or names can be listed on the command line or the script will go into a prompting loop. The `if` statement tests if any arguments have been listed on the command line. In either case, the subroutine `do_query` is called to look up the listing. The `look` subroutine positions the file at the first line which begins with a sequence that is lexically greater than or equal to the key. The characters `"| "` are prepended to the key because all records in the database start with those. Finally, all records that match the key are printed. For example,

```
=> watts, d
| WATTS, Donald G.          . . .
| WATTS, Donna Lucas        . . .
=>
```

There is much more information printed on each line but I have omitted that here in the interest of conserving space.

Some reasonable enhancements to this program are to add a special case of `q` to mean "quit" and the format the records as they are printed. This script and a version that formats the records a bit more pleasingly are available for anonymous ftp from the machine wingra.stat.wisc.edu (128.105.5.32) in the directory `pub/ASA`.

6.2. Current Index Lookups

Creating a query program for the Current Index database is considerably more complicated than a simple binary lookup in the membership directory. Since the scripts that Paul Tukey and I have created for the CIS database are rather long and complicated, I will not discuss them in detail but rather give the broad outlines.

To make it easier to distribute the database to customers on floppy disks, it is divided into two data files for each year. Typically these are about 0.5 Mb. in size. We combine the files for each year and leave them under a name like `80` for the data from 1980. Each line contains fields listing the citation source, the title, the authors' names, keywords, alternate spellings of authors' names, and alternate spellings of words in the title.

We first create an inverted index for all the non-trivial words (truncated to six characters maximum) that occur in the title, the authors' names, and the keywords. This is an intensive process that takes about an hour on a moderately fast workstation but it only has to be done once a year when a new version of the database is released. The index is a two-stage index. A master index lists all the keys and indicates which data files contain records matching

that key. For each data file with that key, it lists the number of records matching the key and a pointer to the byte position in a secondary index that gives the byte positions of all of these records. At Ron Thisted's suggestion we adopted the convention that, when there is only one record in the file matching the key, the pointer in the master index is to the record itself, not to a secondary index. To save space in the index, all the data files are given single character abbreviations. A separate configuration file matches the abbreviations to the names of the data files and the index file.

An example may make things clearer. A sample record from the master index is

```
lindne C 2 163613 D 1 519124 J 1 1201041
```

This indicates that there are two citations in the file with abbreviation C, and one citation in both D and J. The "C" file is `79` and has a secondary index called `79.ix`. The line

```
287293 421922
```

begins at position 163613 in this file. Those numbers are the byte positions of the beginnings of the two records with this key in the file `79`. For files D and J a reference to a secondary index is not necessary because there is only one record to be indexed. The number 519124 gives to the byte position of the beginning of the record with this key in the file `80`.

One advantage of this two level index method is that it keeps the master index to a manageable size. If every byte position for every file were listed in a single level index, it would be very difficult for a human to try to browse it to check on correctness. A more important aspect of this organization is that it makes it easier to intersect the set of references that match different keys. It is common for a user to want to find those citations that match all of a set of keys. For example, a favorite test case at Texas A&M is the following

```
> CIS
Keyword access to CIS database.
Type '?' for help.
=> time series timeslab
====================
Frequencies: time(4239) series(3104) timesl(4)
====================
[Number of combined matches in 87: 1]
====================
%AUTHOR   = H. Joseph Newton
%TITLE    = TIMESLAB: A time series analysis ...
%JOURNAL  = ASA Proc. of Busn. and Econ.  ...
. . .
```

Note that there are more than 4000 citations for the key 'time', more than 3000 for 'series' and only 4 for 'timesl'. We want to find those citations which match all these keys. We can tell immediately that there are no citations containing 'timesl' before 87 so we don't even have to consider the individual citations with 'time' or 'series' before then. Similarly in the 87 file we only have to match the citations with 'time' or 'series' against the one citation with 'timesl'. Doing the intersections a data file at a time results in considerable time savings.

Once the index is established a server program can be started. As mentioned in section 5., **perl** scripts can bind to sockets and accept connections over sockets. This means that the access to the database can be divided between a client run by the user on one machine and a server daemon running perhaps on another machine. In this case, the server accepts a connection to a socket, forks a child process to handle the instance of the connection, and goes back to listening on the socket. The child process accepts requests of keys to lookup and returns the citations. All the interaction with the user of prompting, decoding the user's commands, and formatting the output is carried out by the client script possibly running on another machine.

The scripts for creating the inverted index, running the server and the client are available for anonymous ftp from wingra.stat.wisc.edu as the file `pub/src/CIS.shar.Z`.

It is remarkable that a language which can be used as readily as shown in the simple examples of section 3., can also perform such powerful feats as communicating through sockets or forking child processes. It means that learning **perl** to write simple data manipulation scripts is a good investment of time because you can continue to use the same language if your needs become more complicated.

REFERENCES

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger (1988). *The AWK Programming Language*. Addison-Wesley.

Becker, Richard A., John M. Chambers, and Allan R. Wilks (1988). *The New S Language*. Wadsworth, Pacific Grove, California.

Chambers, John M. and Trevor J. Hastie, eds. *Statistical Models in S*. Wadsworth, Pacific Grove, California.

Kernighan, Brian W. and Dennis M. Richie (1988). *The C Programming Language (Second edition)*. Addison-Wesley.

Wall, Larry and Randall L. Schwartz (1990). *Programming perl*. O'Reilly & Associates, Sebastopol, California.