Shell Scripts and Awk

Tim Love tpl@eng.cam.ac.uk

February 26, 2009

For those who have written programs before and have used Unix from the command line

Contents

1 Shell Programming **1**
1.1 Wildcard characters 2
1.2 Arguments from the command line 2
1.3 Constructions 2
1.4 Quoting and special characters 3

2 Input/Output and Redirection **4**

3 Shell Variables, Aliases and Functions **5**
3.1 Variables 5
3.2 Arrays 6
3.3 Aliases 6
3.4 Functions 6

4 Some useful shell and Unix commands **7**
4.1 Shell commands 7
4.2 Unix accessories 7

5 Exercises **8**

6 Answers to examples **9**

7 Customising **11**

8 Shell Creation **11**

9 Advanced Features and Speed-ups **12**
9.1 Signals and Temporary Files 13

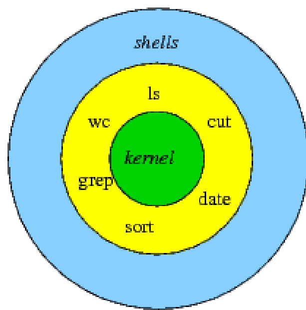
10 Awk **14**

11 References **15**

Copyright ©2009 by T.P. Love. This document may be copied freely for the purposes of education and non-commercial research. Cambridge University Engineering Department, Cambridge CB2 1PZ, England.

1 Shell Programming

At the heart of Unix is a *kernel* whose routines aren't easy to use directly. There are also many pre-written programs (`ls` to list files, etc). They're easier to use, but they don't understand the use of '*' as a wildcard character, and besides, you need some sort of editor when you type commands in if you're going to use the command line. The 'shell' is the interface between the user and the system. The shell isn't only a line editor and a command line interpreter that understands wildcards, it's also a language with variables, arrays, functions and control structures. Command lines can be put into a file and executed. These so-called "shell scripts" can quickly be written and tested and should be tried in association with other standard unix utilities before embarking on a higher level language, at least for prototyping purposes.



Various text-based shells are in use. **sh**, the Bourne Shell, is the oldest. The C-shell (**csh**) has many useful features lacking from **sh** but isn't that good for programming in. The Korn Shell (**ksh**) and the (very similar) POSIX shell are developments of **sh** that incorporates many **csh** features. **bash** is similar and is freely available (it's the default on linux and MacOS X). This document is aimed at `bash` users on CUED's Teaching System, though non-**csh** users elsewhere shouldn't have any problems.

Writing a shell script is easy - start up your editor with a file called `try` then type a few harmless commands into this file, one per line. For example

```
hostname
date
ls
```

then save it as text. You want to make this file executable, so in the terminal window type '`chmod u+x try`', adding eXecute permission for the User. Run this script by typing its name (on some systems you may need to type '`./try`' - the '.' is shorthand for 'the current directory'). You should see the machine's name, the date and a list of files.

Even scripts as simple as this can save on repetitive typing, but much more can be easily achieved.

1.1 Wildcard characters

The `*` and `?` characters have a special meaning to the shell when used where filenames are expected. If you have files called `bashful`, `sneezy` and `grumpy` in your directory and you type

```
ls *y
```

you'll list `sneezy` and `grumpy` because `*` can represent any number of characters (except an initial '.'). `?` represents a single character, so

```
ls *u?
```

will only print filenames whose penultimate letter is `u`.

1.2 Arguments from the command line

It is easy to write a script that takes arguments (options) from the command line. Type this script into a file called `args`.

```
echo this $0 command has $# arguments.
echo They are $*
```

The `echo` command echoes the rest of the line to the screen by default. Within a shell script, `$0` denotes the script's name, `$1` denotes the first argument mentioned on the command line and so on. Make the script executable then try it out with various arguments. For example, if you type

```
args first another
```

then

- `$#` would be replaced by 2, the number of arguments,
- `$0` would be substituted by `args`,
- `$*` would be substituted by `first another`, ('`*`' having a wildcard meaning)

1.3 Constructions

The shell has loop and choice constructs. These are described in the shell's manual page (type `man bash` on Linux or MacOS). Here are some examples. Type them into a file to see what they do, or copy-paste the programs onto the command line. Note that after a `#` the rest of the line isn't executed. These comment lines are worth reading.

```
# Example 0 : While loop. Keeping looping while i is less than 10
# The first line creates a variable. Note that to read a
# variable you need to put a '$' before its name
   i=0
   while [ $i -lt 10 ]
   do
     echo i is $i
     let i=$i+1
   done

# Example 1 : While loop.
# This script keeps printing the date. 'true' is a command
# that does nothing except return a true value.
# Use ^C (Ctrl-C) to stop it.
   while true
   do
     echo "date is"
     date
   done

# example 2: For Loop.
# Do a letter, word and line count of all the files in
# the current directory.
# The `*' below is expanded to a list of files. The
# variable `file' successively takes the value of
# these filenames. Preceding a variable name by `$'
# gives its value.
   for file in *
   do
     echo "wc $file gives"
     wc $file
   done
```

```
# Example 3: If.
# like the above, but doesn't try to run wc on directories
   for file in *
   do
     if [ ! -d $file ] #ie: if $file isn't a directory
     then
       echo "wc $file gives"
       wc $file
     else
       echo "$file is a directory"
     fi
   done
```

```
# Example 4 : Case - a multiple if
# Move to the user's home directory
cd
for file in .?*
do
   #Now check for some common filenames.
   case $file in
     .kshrc) echo "You have a Korn Shell set-up file";;
     .bashrc) echo "You have a Bash Shell set-up file";;
     .Xdefaults) echo "You have an X resource file";;
     .profile) echo "You have a shell login file";;
   esac
done
```

1.4 Quoting and special characters

We've already met a number of symbols that have a special meaning to the shell. Not yet mentioned are braces (`{ .. }`) which are use to surround variable names when it's not clear where they end. For example, if you want to want to print the value of `i` with "ed" added on the end, you could use `echo ${i}ed`.

Putting the symbols in single quotes disables their special meaning. Using double quotes disables some but not others. Here are some examples using quotes. Try them and see if they work as you expect.

```
echo '{  *  $  $xyz  #'
echo "  *  $  $xyz  ' #  /"
```

Single special characters can be disabled by preceding them with a '\' character, so to print a quotemark you can use `echo \"`

2 Input/Output and Redirection

Unix processes have standard input, output and error channels. By default, standard input comes from the keyboard and standard output and error go to the screen, but redirecting is simple. Type

```
date
```

and the date will be shown on screen, but type

```
date > out
```

and the output goes into the file called `out`.

```
hostname >> out
```

will append hostname's output to the file out. stderr (where error message go) can also be redirected. Suppose you try to list a non-existent file (blah, say)

```
ls blah
```

You will get an error message 'blah not found'. There's a file on all unix systems called /dev/null, which consumes all it's given. If you try

```
ls blah > /dev/null
```

you still wouldn't get rid of the error message because, not surprisingly, the message is being sent to stderr not stdout, but

```
ls blah 2>/dev/null
```

redirecting channel 2, stderr, will do the trick.

You can also redirect output into another process's input. This is done using the '|' character (often found above the <Return> key). Type 'date | wc' and the output from date will be 'piped' straight into a program that counts the number of lines, words and characters in its input. cat is a program that given a filename prints the file on-screen, so if you have a text-file called *text* you can do cat text | wc or wc < text instead of the usual wc text to count the contents of the file.

Standard output can also be 'captured' and put in a variable. Typing

```
d=$(date)
```

will set the variable d to be whatever string the command date produces. The use of $() to do this supercedes the use of backquotes (i.e. d=`date`) in older shells. $(<datafile) is a faster way of doing $(cat datafile) to put the contents of a file into a variable.

If you want to print lots of lines of text you could use many echo statements. Alternatively you could use a here document

```
cat<<END
Here is a line of text
and here is another line.
END
```

The cat command uses as its input the lines between the END words (any word could be used, but the words have to be the same).

If you want to read from a file a line at a time (or want to get input from the keyboard) use read. You can use one of the methods below – the second may be much the faster.

```
echo Reading /etc/motd using cat
cat /etc/motd | while  read line
do
  echo "$line"
done

echo Reading /etc/motd using exec redirection
exec 0</etc/motd
while read line
do
  echo "$line"
done
```

3 Shell Variables, Aliases and Functions

3.1 Variables

The shell maintains a list of variables, some used by the shell to determine its operation (for instance the `PATH` variable which determines where the shell will look for programs), some created by the user.

A shell variable can be given a type (just as variables in Java or Fortran can), but this isn't necessary. Typing

```
pix=/usr/lib/X11/bitmaps
```

(note: no spaces around the '=' sign) creates a variable '`pix`' whose value can be accessed by preceding the name with a '`$`'. '`cd $pix`' will take you to a directory of icons. If you type

```
pix=8
```

then `pix` will be treated as a number if the context is appropriate. The `let` command built into the shell performs integer arithmetic operations. It understands all the C operators except for ++ and −−. The `typeset` command can be used to force a variable to be an integer, and can also be used to select the base used to display the integer, so

```
pix=8
let "pix=pix<<2"
echo $pix
typeset -i2 pix
echo $pix
```

will print `32` then `2#100000` onscreen. The `let` command (note that `$` signs aren't necessary before the operands) uses C's << operator to shift bits 2 to the left and the `typeset` command makes `pix` into a base 2 integer.

`typeset` can do much else too. Look up the shell man page for a full list. Here are just a few examples

```
# You can control the how many columns a number occupies and whether
# the number is right or left justified. The following
# typeset command right-justifies j in a field of width 7.
j=1
typeset -R7 j
echo "($j)"

# Variables can be made read-only. This j=2 assignment causes an error.
typeset -r j
j=2

# String variables can be made upper or lower case.
string=FOO
echo $string
typeset -l string
echo $string
```

3.2 Arrays

The Korn shell and bash support one-dimensional arrays. These needn't be defined beforehand. Elements can be assigned individually (for example `name[3]=diana`) or en masse.

```
colors[1]=red
colors[2]=green
colors[3]=blue

echo The array colors has ${#colors[*]} elements.
echo They are ${colors[*]}
```

3.3 Aliases

Aliases provide a simple mechanism for giving another name to a command.

```
alias mv="mv -i"
```

runs `mv -i` whenever the `mv` command is run. Typing `alias` gives a list of active aliases. Note that only the first word of each command can be replaced by the text of an alias.

3.4 Functions

Functions are much as in other high level languages. Here's a simple example of a script that creates a function, then calls it.

```
#takes 3 args; a string, the number of the target word in the string
#and which chars in that field you want. No error checking done.
function number {
   echo $1 | cut -d' ' -f$2 | cut -c $3
}

echo -n "Percentage is "
# the next line prints characters 7-8 of word 1 in the supplied string
# by calling the above function
number "%Corr=57   Acc=47.68 Fred" 1 7-8

echo -n "Acc is "
number "%Corr=57   Acc=47.68 Fred"  2 5-9
```

4 Some useful shell and Unix commands

To become fluent writing shell scripts you need to know about common shell commands (as shown earlier) but you also need to be aware of some common unix commands.

4.1 Shell commands

See the `ksh` or `sh-posix` man page for more info on these. In `bash` you can use the `help` command.

alias :- to list aliases

let :- to do maths. Eg:- `let x=2*4-5 ; echo $x`

set :- to list the current variables and their values.

typeset -f :- to list functions

typeset -i :- to list integer variables

which :- to find where a program is (in some shells you use `whence` instead)

4.2 Unix accessories

See the corresponding man page for more info on these. Note that many of the commands have dozens of options that might save you needing to write code.

wc:- This counts words lines and characters. To count the number of files in a directory, try `ls | wc -l`

grep:- '`grep Kwan /etc/passwd`' prints all lines in `/etc/passwd` that include '`Kwan`'. The return value indicates whether any such lines have been found.

sed:- a stream editor. Doing `ls | sed s/a/o/g` will produce a listing where all the 'a' characters become 'o'. Numerous tricks and tutorials are available from the Handy one-liners for SED[1] file.

basename:- Strips the directory and (optionally) suffix from a filename.

tr:- Translates one set of characters to another set. For example, try

```
echo "date" | tr d l
```

The following does case conversion

```
echo LaTeX | tr '[:upper:]' '[:lower:]'
```

You can also use it as follows, "capturing" the output in a variable

```
old="LaTeX"
new=$(echo $old | tr '[:upper:]' '[:lower:]')
```

test:- This program can be used to check the existence of a file and its permissions, or to compare numbers and strings. Note that '`if [ -f data3 ]`' is equivalent to '`if test -f data3`'; both check for a file's existence, but the `[...]` construction is faster – it's a builtin. Note that you need spaces around the square brackets. Some examples,

```
if [ $LOGNAME = tpl ] # Did the user log in as tpl?
if [ $num -eq 6 ]     # Is the value of num 6?
if [ $(hostname) = tw000 ] # Does hostname produce 'tw000'?
                           # i.e. is that the machine name?
```

sort:- '`ls -l | sort -nr +4`' lists files sorted according to what's in column 4 of `ls -l`'s output (their size, usually).

cut:- Extracts characters or fields from lines. `cut -f1 -d':' < /etc/passwd` prints all the uid's in the passwd file. The following code shows a way to extract parts of a filename.

```
filename=/tmp/var/program.cc

b=$(basename $filename)
prefix=$(echo $b | cut -d. -f 1)
suffix=$(echo $b | cut -d. -f 2)
```

find:- finds files in and below directories according to their name, age, size, etc.

[1] http://www-h.eng.cam.ac.uk/help/tpl/unix/sed.html

5 Exercises

Suppose in a directory that you wanted to change all the filenames ending in `.f77` so that they instead ended in `.f90`. How would you go about it?

If you're new to Unix you may need to find out how to change filenames. Typing `apropos filename` will list the programs and routines whose summaries mention `filename`. Alas, the most useful command isn't mentioned there. Typing `apropos rename` lists `mv` which is what you need.

Maybe next you might try "`mv *.f77 *.f90`". Alas, this won't work - the shell will replace `*.f77` by a list of filenames and the resulting command will fail. You need to use a loop of some sort. You also need a way to remove a suffix (look up `basename`) and how to add a new suffix. It's worth experimenting with a single name first. Do `filename=test.f77` and see if you can produce a `test.f90` string from `$filename`. One solution is

```
for filename in *.f77
do
  b=$(basename $filename .f77)
  mv $filename $b.f90
done
```

Here are some more exercises with useful man pages suggested. None of the solutions to these should be much more than a dozen lines long. Answers to some of these are in the next section.

1. Change the `args` script supplied earlier so that if no argument is provided, "They are" isn't printed, and if exactly 1 argument is provided, "... 1 argument" rather than "... 1 arguments" is printed (use `if`)

2. Read in two numbers from the keyboard and print their sum (see the `read`, `echo` and `let` commands in the shell manual page).

3. Write a shell script that given a person's uid, tells you how many times that person is logged on. (`who`, `grep`, `wc`)

4. Write a shell script called `lsdirs` which lists *just* the directories in the current directory (`test`).

5. Write a shell script called `see` taking a filename name as argument which uses `ls` if the file's a directory and `more` if the file's otherwise (`test`)

6. Write a shell script that asks the user to type a word in, then tells the user how long that word is. (`read`, `wc`)

7. In many versions of unix there is a **-i** argument for **cp** so that you will be prompted for confirmation if you are about to overwrite a file. Write a script called `cpi` which will prompt if necessary without using the **-i** argument. (`test`)

8. Write a shell script that takes a uid as an argument and prints out that person's name, home directory, shell and group number. Print out the name of the group corresponding to the group number, and other groups that person may belong to. (`groups`, `awk`, `cut`. Also look at `/etc/passwd` and `/etc/groups`).

9. Sort `/etc/passwd` using the uid (first field) as the key. (`sort`)

10. Suppose that you want to write the same letter to many people except that you want each letter addressed to the senders personally. This *mailmerge* facility can be created using a shell script. Put the names of the recipients (one per line) in a file called

names, create a texfile called template which has NAME wherever you want the person's name to appear and write a script (using sed) to produce a temporary file called letter from the template file.

6 Answers to examples

Note that these script begin with a line saying #!/bin/sh. This will force the script to be interpreted by the **Posix** shell even if the user is using another, incompatible, type of shell. Your system might require a different initial line. If you leave the line out, the user's current shell will be used.

- Arguments

```
#!/bin/sh
if [ $# = 1 ]
then
   string="It is "
   ending=""
else
   string="They are "
   ending="s"
fi
echo This $0 command has $# argument${ending}.
if [ $# != 0 ]
then
echo $string $*
fi
```

- Adding Numbers

```
#!/bin/sh
echo "input a number"
read number1

echo "now input another number"
read number2

let answer=$number1+$number2
echo "$number1 + $number2 = $answer"
```

- Login Counting Script

```
#!/bin/sh
times=$(who | grep $1 | wc -l)
echo "$1 is logged on $times times."
```

- List Directories

```
#!/bin/sh
for file in $*
do
  if [ -d $file ]
  then
```

```
    ls $file
  fi
done
```

- See Script

```
#!/bin/sh
for file in $*
do
  if [ -d $file ]
  then
     echo "using ls"
     ls $file
  else
     more $file
  fi
done
```

- Word-length script

```
#!/bin/sh
echo "Type a word"
read word
echo $word is $(echo -n $word | wc -c) letters long
echo or $word is ${#word} letters long
```

- Safe Copying

```
#!/bin/sh
if [ -f $2 ]
then
        echo "$2 exists. Do you want to overwrite it? (y/n)"
        read yn
        if [ $yn = "N" -o $yn = "n" ]
        then
            exit 0
        fi
fi
cp $1 $2
```

- Mailmerge

```
#!/bin/sh
for name in $(<names)
do
  sed s/NAME/$name/ <template >letter
  # here you could print the letter file out
done
```

7 Customising

Shell scripts are used to customise your environment. `/etc/profile` is read once on login. The shell then looks for a file in your home directory called `.profile`. If you haven't got one then, depending on your set-up, a system default file might be used instead. Typing `printenv` and `alias` will show you how things have been set up for you. The '`type`' command can help you find out what will happen when you type something. E.g.

```
tw613/tpl: type cd
cd is a shell builtin
tw613/tpl: type handouts
handouts is /usr/local/bin/handouts
tw613/tpl: type lp
lp is a function
tw613/tpl: type history
history is an exported alias for fc -l
```

8 Shell Creation

Whenever you invoke a command that is not built into the shell, a new shell process is created which inherits many of the properties of its parent. However, variables and aliases are not inherited unless they are exported. Type "`export`" and you will see what's been exported by the initialisation files. Typing

```
she=janet ; he=john ; export she
```

(';' is a statement separator) creates 2 new variables, only one of which is exported. They can be printed out using

```
echo $she $he
```

Now type "`bash`". This will put you into a new shell, a child of the original.

```
echo $she $he
```

will only print out one name, the one you exported. If you type "`ps -f`" you will get an output something like

```
UID        PID  PPID  C STIME TTY          TIME CMD
tpl       6006 31173  0 14:03 pts/3    00:00:00 bash
tpl       6027  6006  0 14:03 pts/3    00:00:00 ps -f
tpl      31173 31172  0 09:01 pts/3    00:00:00 -bash
```

Notice that you are running 2 `bash` processes. The `PPID` column gives the `PID` of the parent process so you can see that the 2nd shell was started from within the 1st. You can kill this new shell by typing `exit`. More interestingly, you can suspend the shell by typing `<CTRL> Z` and fall back into the original shell. Convince yourself that this is so by trying the '`echo`' and '`ps`' commands again, or '`jobs`', then return to the child shell by typing '`fg`' (foreground). You won't often need to jump around like this but it's useful to be aware of the mechanism, especially since it helps elucidate some problems regarding scope and lifetimes of variables.

When you run a script by typing its name, a new process is started up. If the script contains a `cd` command or changes the value of an environmental variable, only the new process will be affected – the original shell's environment and current directory won't have been 'overwritten'. If however you type '`source scriptname`' then the commands in the script are run in the original process. This is called *sourcing* the script.

You can force shell script commands to run in a new process by using brackets. For example, if you type "`cd / ; ls`" you will be left in the root directory, but if you type "`(cd / ; ls)`" you'll end up where you began because the `cd` command was run in a new, temporary, process.

9 Advanced Features and Speed-ups

There are many mentioned in the shell man page. Here are just a few:-

getopt :- Helps parse command line options.

variable substitution :- 'X=${H:-/help}' sets the variable X to the value of H iff H exists, otherwise it sets X to /help.

RANDOM :- a random number

```
for i in 1 2 3
do
  echo $RANDOM is a random number
done
```

Regular Expressions :- Commands like grep and sed make use of a more sophisticated form of pattern matching than the '*' and '?' wildcard characters alone can offer. See the manual page for details - on my system typing "man 5 regexp" displays the page.

Eval :- Consider the following

```
word1=one
word2=two
number=1
```

The variable word$number has the value one but how can the value be accessed? If you try echo $word$number or even echo ${word$number} you won't get the right answer. You want $number processed by the shell, *then* the resulting command processed. One way to do this is to use eval echo \$word$number - the \ symbol delays the interpretation of the first $ character until eval does a second pass.

Alternative notations :- ((a=a<<2+3)) is the equivalent of let a="a<<2+3" – note that this variant saves on quotemarks.

```
[[ -f /etc/passwd ]]
   echo the passwd file exists
```

is another way of doing

```
if [ -f /etc/passwd ]
then
   echo the passwd file exists
fi
```

or you could use the logical && operator, which like its C cousin only evaluates the second operand if the first is true.

```
[ -f /etc/passwd ] && echo the passwd file exists
```

Bash features:- C-like for loops are possible.

```
for((i=1; $i<3; i=i+1))
do
  echo $i
done
```

Regular expressions (wildcard characters etc) can be used in some comparisons. In the following, `?[aeiou]?` is a regular expression that matches a character followed by one of a, e, i , o ,u, followed by a character

```
for word in four six sly
do
  if [[ $word == ?[aeiou]? ]]
  then
    echo $word is 3 letters long with a central vowel
  else
    echo $word is not 3 letters long with a central vowel
  fi
done
```

9.1 Signals and Temporary Files

A script may need to create temporary files to hold intermediate results. The safest way to do this is to use `mktemp` which returns a currently unused name. The following command creates a new file in `/tmp`.

```
newfile=$(mktemp)
```

If a script is prematurely aborted (the user may press `^C` for example) it's good practise to remove any temporary files. The `trap` command can be used to run a tidy-up routine when the script (for whatever reason) exits. To see this in action start the following script then press `^C`

```
newfile=$(mktemp)
trap "echo Removing $newfile ; rm -f $newfile" 0
sleep 100
```

10 Awk

Unix has many text manipulation utilities. The most flexible is `awk`. Its conciseness is paid for by

- speed of execution.
- potentially hieroglyphic expressions.

but if you need to manipulate text files which have a fairly fixed line format, `awk` is ideal. It operates on the fields of a line (the default field separator, FS, being `<space>`). When `awk` reads in a line, the first field can be referred to as '`$1`', the second '`$2`' etc. The whole line is '`$0`'. A short `awk` program can be written on the command line. eg

```
cat file | awk '{print NF,$0}'
```

which prepends each line with the Number of Fields (ie, words) on the line. The quotes are necessary because otherwise the shell would interpret special characters like '`$`' before `awk` had a chance to read them. Longer programs are best put into files.

Two examples in /export/Examples/Korn_shell (wordcount and awker) should give CUED users a start (the awk manual page has more examples). Once you have copied over wordcount and text, do

```
wordcount text
```

you will get a list of words in text and their frequency. Here is wordcount

```
awk '    {for (i = 1; i<=NF ; i++)
                 num[$i]++
     }
     END {for (word in num)
                 print word, num[word]
     }
     ' $*
```

The syntax is similar to that of C. awk lines take form

```
<pattern>          { <action> }
```

Each input line is matched against each awk line in turn. If, as here in wordcount, there is no target pattern on the awk line then all input lines will be matched. If there is a match but no action, the default action is to print the whole line.

Thus, the **for** loop is done for every line in the input. Each word in the line (NF is the number of words on the line) is used as an index into an array whose element is incremented with each instance of that word. The ability to have strings as array 'subscripts' is very useful. END is a special pattern, matched by the end of the input file. Here its matching action is to run a different sort of **for** loop that prints out the words and their frequencies. The variable word takes successively the value of the string 'subscripts' of the array num.

Example 2 introduces some more concepts. Copy /export/Examples/Korn_shell/data (shown below)

```
NAME     AMOUNT   STATUS
Tom      1.35     paid
Dick     3.87     Unpaid
Harry    56.00    Unpaid
Tom      36.03    unpaid
Harry    22.60    unpaid
Tom      8.15     paid
Tom      11.44    unpaid
```

and /export/Examples/Korn_shell/awker if you haven't done so already. Here is the text of awker

```
awk '
$3 ~ /^[uU]npaid$/ {total[$1] += $2; owing=1}

END {
  if (owing)
    for (i in total)
       print i, "owes", total[i] > "invoice"
  else
    print "No one owes anything" > "invoice"
}

' $*
```

Typing

```
awker data
```

will add up how much each person still owes and put the answer in a file called `invoice`. In `awker` the 3rd field is matched against a regular expression (to find out more about these, type `man 5 regexp`). Note that both '`Unpaid`' and '`unpaid`' will match, but nothing else. If there is a match then the action is performed. Note that `awk` copes intelligently with strings that represent numbers; explicit conversion is rarely necessary. The '`total`' array has indices which are the people's names. If anyone owes, then a variable '`owing`' is set to 1. At the end of the input, the amount each person owes is printed out.

Other `awk` facilities are:-

- fields can be split:-

  ```
  n = split(field,new_array,separator)
  ```

- there are some string manipulation routines, e.g.:-

  ```
  substr(string,first_pos,max_chars), index(string,substring)
  ```

- awk has built-in math functions (exp, log, sqrt, int) and relational operators (==, !=, >, >=, <, <=, ~ (meaning "contains"), ! ~).

As you see, `awk` is almost a language in itself, and people used to C syntax can soon create useful scripts with it.

11 References

- SHELLdorado[2]
- Shell Information[3] from CUED, including a link to this document
- Documentation Source-code[4]
- Linux Shell Scripting Tutorial - A Beginner's handbook[5]

[2] http://www.oase-shareware.org/shell/
[3] http://www-h.eng.cam.ac.uk/help/unix.html#Shell
[4] http://www-h.eng.cam.ac.uk/help/documentation/docsource/index.html
[5] http://www.freeos.com/guides/lsst/