Unix Shell Scripts

Norman Matloff

July 15, 2002

1 Introduction

In previous discussions we have talked about many of the facilities of the C shell, such as command aliasing, job control, etc. In addition, any collection of **csh** commands may be stored in a file, and **csh** can be invoked to execute the commands in that file. Such a file is known as a shell **script** file. The language used in that file is called shell script language. Like other programming languages it has variables and flow control statements (e.g. if-then-else, while, for, goto).

In Unix there are several shells that can be used, the C shell (**csh** and its extension, the T C shell **tcsh**), the Bourne Shell (**sh** and its extensions the Bourne Again Shell **bash** and the highly programmable Korn shell **ksh**) being the more commonly used.

Note that you can run any shell simply by typing its name. For example, if I am now running **csh** and wish to switch to **ksh**, I simply type **ksh**, and a Korn shell will start up for me. All my commands from that point on will be read and processed by the Korn shell (though when I eventually want to log off, exiting the Korn shell will still leave me in the C shell, so I will have to exit from it too).

2 Invoking Shell Scripts

There are two ways to invoke a shell script file.

2.1 Direct Interpretation

In direct interpretation, the command

```
csh filename [arg ...]
```

invokes the program **csh** to interpret the script contained in the file `filename'.

2.2 Indirect Interpretation

In indirect interpretation, we must insert as the first line of the file

```
#! /bin/csh
```

or

```
#! /bin/csh -f
```

(there are situations in which this is not necessary, but it won't hurt to have it), and the file must be made *executable* using **chmod** (see previous discussion). Then it can be invoked in the same way as any other command, i.e., by typing the script file name on the command line.

The -f option says that we want fast startup, which it will achieve by <u>not</u> reading or executing the commands in .cshrc Thus for example, we won't have the `set' values for shell variables or the aliases from that file, but if we don't need them, this will be much faster (if we need a few of them, we can simply add them to the script file itself).

3 Shell Variables

Like other programming languages the **csh** language has variables. Some variables are used to control the operation of the shell, such as $path and $history, which we discussed earlier. Other variables can be created and used to control the operation of a shell script file.

3.1 Setting Variables

Values of shell variable are all character-based: A value is formally defined to be a **list** of zero or more **elements**, and an element is formally defined to be a character string. In other words, a shell variable consists of an array of strings.

For example,

```
    set X
```

will set the variable $X to have an empty list as its value. The command

```
    set V = abc
```

will set V to have the string `abc' as its value. The command

```
    set V = (123 def ghi)
```

will set V to a list of three elements, which are the strings `123', `def' and `ghi'.

The several elements of a list can be treated like array elements. Thus for V in the last example above, $V[2] is the string `def'. We could change it, say to `abc', by the command

```
set V[2] = abc
```

3.2 Referencing and Testing Shell Variables

The value of a shell variable can be referenced by placing a **$** before the name of the variable. The command

```
echo $path
```

will output the value of the variable $path. Or you can access the variable by enclosing the variable name in curly brace characters, and then prefixing it with a **$**. The command

```
echo ${path}
```

would have the same result as the last example. The second method is used when something is to be appended to the contents of the variable. For example, consider the commands

```
set fname = prog1
rm ${fname}.c
```

These would delete the file `prog1.c'.

To see how many elements are in a variable's list, we prefix with a **#** then a **$**. The command

```
echo $#V
```

above would print 3 to the screen, while

```
echo $#path
```

would reveal the number of directories in your search path.

The @ command can be used for computations. For example, if you have shell variables $X and $Y, you can set a third variable $Z to their sum by

```
@Z = $X + $Y
```

4 Command Arguments

Most commands have arguments (parameters), and these are accessible via the shell variable $argv. The first parameter will be $argv[1], the second $argv[2], and so on. You can also refer to them as $1, $2, etc. The number of such arguments (analogous to argc in the C language) is $#argv.

For example, consider the following script file, say named Swap:

```
#! /bin/csh -f

set tmp = $argv[1]
cp $argv[2] $argv[1]
cp $tmp $argv[2]
```

This would do what its name implies, i.e. swap two files. If, say, I have files x and y, and I type

```
Swap x y
```

then the new contents of x would be what used to be y, and the new contents of y would be what used to be x.

5 Language Constructs

The shell script language, like other programming languages, has constructs for conditional execution (if-then-else; while), iterative execution (for loop), a switch statement, and a goto statement:

1. if-then-else

The syntax of the if-then-else construct is

```
    if ( expr ) simple-command
```

or

```
    if ( expr ) then
        commandlist-1
    [else
        commandlist-2]
    endif
```

The expression *expr* will be evaluated and according to its value, the *commandlist-1* or the *commandlist-2* will be executed. The portion of the construct enclosed in '[' and ']' is optional.[1]

As an example, suppose we write a shell script which is supposed to have two parameters, and that the code will set up two variables, `name1' and `name2' from those two parameters, i.e.

```
set name1 = $argv[1]
set name2 = $argv[2]
```

(which presumably it would make use of later on). But suppose we also wish to do error-checking, emitting an error message if the user gives fewer than two, or more than two, parameters. We could use the following code

```
if ($#argv <> 2) then
    echo "you must give exactly two parameters"
else
    set name1 = $argv[1]
    set name2 = $argv[2]
endif
```

2. while

The syntax of *while* loop construct is

```
while ( expr )
        commandlist
end
```

The *commandlist* will be executed until the *expr* evaluates to false.

3. foreach

The syntax of *foreach* loop construct is

```
foreach var ( worddlist )
    commandlist
end
```

The *commandlist* is executed once for each word in the *wordlist*, and each time the variable *var* will contain the value of that word. For example, the following script can search all immediate subdirectories of the current directory for a given file (and then quit if it finds one):

```
#! /bin/csh -f
```

```
        set f = $1
        foreach d (*)
            if (-e $d/$f) then
                 echo FOUND: $d/$f
                 exit(0)
            endif
        end
        echo $f not found in subdirectories
```

For example, say I call this script FindImm, and my current directory consists of files s, t and u, with s and t being subdirectories, and with t having a file x. Typing

```
FindImm x
```

would yield the message

```
FOUND: t/x
```

Here is how it works: In the line

```
        foreach d (*)
```

the `*' is a wild card, so it would expand to a list of all files in my current directory, i.e. the list (s t u). So, the for-loop will first set d = s, then d = t and finally d = u.

In the line

```
        if (-e $d/$f) then
```

the -e means existence; in other words, we are asking if the file $d/$f exists. If we type `FindImm x' as in the example above, $f would be x, and $d would start out as s, so we would be asking if the file s/x exists (the answer would be no).

4. switch

The switch command provides a multiple branch similar to the switch statement in C. The general form of switch is:

```
    switch ( str )
        case string1:
            commandlist1
            breaksw
        case string2:
            commandlist2
            breaksw
        default
```

```
        commandlist
    endsw
```

The given string *str* is successively matched against the case patterns. Control flow is switched to where the first match occurs. As in file name expansion, a case label may be a literal string, or contain variable substitution, or contain wild-card character such as *,?, etc.

5. **Goto**

The *goto* command provides a way to branch unconditionally to a line identified by a label.

```
goto lab
```

where *lab* is a label on a line (by itself) somewhere in the script in the form

```
lab:
```

6 Examples

6.1 A Shell Script For Deleting Files

This code, which we will call Del, will delete files like **rm** does, prompting for your confirmation for each file to be deleted, including directory files (which the -i option of **rm** won't do).

```
#! /bin/csh -f

    foreach name ($argv)
       if ( -f $name ) then
          echo -n "delete the file '${name}' (y/n/q)?"
       else
          echo -n "delete the entire directory '${name}' (y/n/q)? "
       endif
       set ans = $<
       switch ($ans)
             case n:
                continue
             case q:
                exit
             case y:
                rm -r $name
                continue
       endsw
    end
```

(Before reading further, try this program yourself. Set up a test directory, with several files in it, at least one of which is a subdirectory, with at least one file there. Then type `Del *'.)

The line

```
if ( -f $name ) then
```

tests to see if the file whose name is in $name is an ordinary file, as opposed to a directory file.

The -n option of **echo** tells the shell not to print the newline character, so that our answer, y/n/q, will be on the same line.

In the line

```
set ans = $<
```

the symbol `$ < ' means the input from the keyboard.

The keyword `continue' means to go to the top of the enclosing loop.

The -r option of the **rm** command means that if an argument is a directory, then remove that directory, and all the files (and subdirectories, etc.) within it.

7 Further Information

There are several books dealing with the C shell, but you should first read the man page for **csh**. You will find all kinds of features not mentioned here.

Footnotes:

[1]This is standard notation in the software world, so remember it.