Advanced Unix - Scripts

This document explains how to create and execute Bourne shell scripts. You will also learn about shell builtins, input/output redirection, and quoting as well as the difference between interactive and subshell scripting.

Table of Contents

If you have any comments or suggestions about this document, send them to consult@rice.edu via electronic mail.

1 Introduction

1.1 Scripts

What is a script? A script is a set of commands, either in a file, or typed in at the command line, that perform multiple operations on a file or files. Another term for scripts that might be familiar to you is a macro.

How do they work? To run a file as a script, the file must have the execution bit set. As you'll recall from the Unix I course, this means the file looks like this when you look at it:

```
prompt% ls -1
-rwxr-xr-x  1 joeuser 60 Apr  9 01:57 jocks
```

If the execution bit is set, what happens next depends on the first line of the file. If the first two characters are anything other than #![1], the file is interpreted as a Bourne shell script. If the characters #! are followed by an explicit program location, such as /usr/bin/awk, that program is run as an interpreter on the contents of the file.

As an alternative to setting the execution bits on a file and then running that file directly, it is also possible to read commands from a non-executable file into the current shell. Because this runs the current shell as an interpreter on the file being read, it should be used very carefully, since any environment variables set or changed in the script will be altered in the current shell. As a result, it is safer to run the script in its own subshell.

2 The C Shell

Because the C shell is the default interactive shell on IS-maintained domains, we'll look at functions and features of it first. Here's an example of a simple C shell script:

```
#!/bin/csh
cd ~/csh
ls -l
```

The commands inside it look like things we might type at a prompt. As a result, it's not hard to tell what this script would do. First, it would change directory to a subdirectory called csh, then it would list all the files there in long format (this is accomplished with the "-l" argument to ls).

1. If the kernel loader sees a magic number of 0x2321, it treats the file as an interpreter file. 0x2321 is the hex value of the characters #!. So called "magic numbers" are just values the kernel uses as signals to interpret files in a certain manner.

Exercise 1: You'll find the script above in a file called "simple.csh" in your csh subdirectory. Check the permissions on it, then run it.

The shell script operates just as an interactive shell would in deciding what to execute as well: first it checks to see if there are any filename operations to perform; if it finds any it resolves them. Next it goes through the script one line at a time and executes the commands on those lines. If the command is a shell builtin, it is an incorporated function of the shell - no external program is needed to do it[1].

2.1 Filename Substitution

Filename substitution was covered in Unix II, but to refresh those who might have forgotten its nuances, table 1 details substitution rules.

TABLE 1.

*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by '-' matches any character lexically between the pair, inclusive.
{ str,str,...}	Expand to each string in the comma-separated list.
~ [user]	The user's home directory, as indicated by the value of the variable home, or that of user, as indicated by the password entry for user.

2.2 Shell Variables

Sometimes in a shell script you'll want to save information for use in another command. In such cases, you'll probably want to save it into a variable. Scripts have their own variables, which are stored seperately from those of the shell that you start them from. When the script exits, those variables are forgotten.

The C shell has two ways to set variables, depending on whether they're shell or environment variables. To set environment variables, **csh** has a builtin called **setenv**. The syntax is:

```
prompt% setenv KEY value[:othervalue:...]
```

You can take variables out of your environment with **unsetenv**. Shell variables are assigned using **set**, and disposed of using **unset**. Here are a few examples of setting variables:

```
set today = 'date'                    #shell variable today
set name = "Joe User"                 #shell variable name
```

1. Some of the C and Bourne shell builtins are included in tables 3 and 9 respectively. There are others; for a complete set, see the manual pages for csh and sh.

set string = 'Yes, this is a string.’ #shell variable string

setenv HOST 'hostname' #environment variable HOST

To find out the value of a variable, one would preface its name with a dollar sign.

In the C shell, shell variables can be arrays. The shell variable that stores the path is stored as an array (although the environment that stores the path does so as a single string, where the individual directories are delimited by colons) . Elements of a shell variable array are referenced by using the name of the variable, followed by the number of the element you want (arrays start with the first element) in square brackets. To get the entire shell variable path printed out, you'd enter:

```
prompt% echo $path
```

To get just the third element, you'd enter:

```
prompt% echo $path[3]
```

TABLE 2.

$#name ${#name}	These give the number of words in the variable. The second form is required when using variable syntax. Variable syntax occurs any time you have a variable name directly adjacent to a special character.
$0	This substitutes the name of the file from which command input is being read. An error occurs if the name is not known.
$argv [n]	Prints the nth element of the variable **argv**, which contains the arguments passed into the shell.
$n	Equivalent to $argv[n].
${n}	Used in variable syntax context.
$*	Equivalent to $argv. Prints all parameters.
$?var	Substitures the string 1 if var is set or 0 if it is not set.
$?0	Substitutes 1 if the current input filename is known, or 0 if it is not.
$$	Substitute the process number of the (parent) shell.
$<	Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a C shell script.

Exercise 2: Try the commands above at your prompt. Before doing so, try: `echo path` to check the behavior of the **echo** command.

The **echo** command is a shell builtin that just prints its arguments to STDOUT. If its arguments are strings, it prints the string. If they are variables (prefaced with a dollar sign) , it prints the value of the variable.

There are some special variables besides those listed above that are automatically set by C shell. They are summarized in table 2. Take particular note of the $< variable, as it is the primary method used to read keyboard input while a C shell script is running.

2.3 Quoting

There are three kinds of quotes the shell recognizes, and each of them does something different. When you use backquotes (`),as in the example, the shell does any filename substitution that might be necessary within the string between the backquotes, then runs the resultant string as a command in a subshell. Anv errors generated when this subshell is run are returned to the tty (these errors are called STDERR). The output is returned to the calling shell. In the example, this output is assigned to the variable "today".

Exercise 3: Enter the command: `set today = 'date'` in your shell to set the shell variable today. After you've done so, see what the value stored in `$today` is by entering at your prompt:
`prompt% echo $today`

Double quotes have the effect of passing the text enclosed in them into a variable or command as a single argument, rather than as a bunch of space-separated arguments. Thus:

```
set sentence = "This is a sentence."
```

sets the variable sentence and gives it one value, which is the entire expression "This is a sentence." On the other hand, this example:

```
set sentence = This is a sentence.
```

will set the variable **sentence** to "This"; the C shell loses the extra arguments because they are not parenthesized, indicating an array context, or quoted, indicating a single argument.

Variable substitution is performed in double quotes. This is what differentiates it from single quotes - in single quotes no variable substitution is done.

Here's an example script that utilizes variable substitution and array referencing in double quotes:

```
#!/bin/csh
set fish = (tuna mackerel swordfish shark whale)
set i = 1
while(i < 6)
echo "I'm so hungry, I could eat a whole $fish[$i]\!"
@ i++
end
```

The value of the variable in **$fish [$i]** is substituted for those letters in the expression. The exclamation point needs to be escaped in the C shell, because bang (!) is a special character used in history substitution, which is performed in double quotes. Having a command history is one of the more useful interactive features of **csh**, but it isn't much help in writing scripts.

Exercise 4: Checks the modifications on the shell script **fish.csh**. Once you know it is executable, run it.

Backticks execute the string inside them in a subshell and return the STDOUT and STDERR to the tty. These values may be ignored, redirected, or set to a variable as in the case of our original example:

```
set today = 'date'
```

As with double quotes, variable substitution and file expansion is performed inside backquotes. The next exercise demonstrates this:

Exercise 5: Type **pwd** and observe it's output. Then try typing the following in your shell. After you've set each variable, use echo to check its current value.

```
set this = pwd
set that = '$this'
```

The shell splits input lines into words at SPACE and TAB characters. The characters ;,&,!,<, >, (, and) are special characters. These *metacharacters* can be used literally -that is, their special meaning can be suppressed - by preceding them with a \[1]. A newline preceded by a \ is equivalent to a space character.

2.4 Input and Output Redirection

In the Unix II course you learned how to redirect STDIN and STDOUT. To recap briefly, STDIN can be redirected with the less than sign (<), and STDOUT can be redirected with either one or two greater than signs (> or >>). To append the standard output to an existing file, you must use two greater than signs in a row, like so: >>. If the file will be *created* by the redirected output, use one.

Besides redirecting input and output to files, it is also possible to pipe output from one process to another. Pipes normally associate the STDOUT of the first process (starting from the left) with the STDIN of the second. But not always; pipelines can be separated by semi colons (;), in which case they are executed sequentially. Pipelines that are separated by && or || pipe command form conditional sequences in which the execution of pipelines on the right depends upon the success or failure, respectively, of the pipeline on the left.

Exercise 6: Try this example of conditional execution:

```
test -f nosuchfile && echo This echo will never happen.
```

---

1. Well, not always. There are some annoying exceptions.

2.5 Builtin Commands

We've already used a number of shell builtins in this course. **Echo** writes whatever follows it to STDOUT. **Set** is used to assign values to variables. **While** loops while the condition that follows it in parentheses is true. The C shell has a number of builtin commands, many of which are primarily intended for interactive use. Table 3 lists a number of **csh** builtin commands that are useful for scripts.

The @ operator is one of the C shell's builtin features that is not shared by the Bourne shell. Inside an environment initiated with the at sign, you can set variables to the result of any of a variety of mathematical operations. These operations are integer based, and are resolved right to left if the operators are of equal precedence, so parenthesize when using the @.

TABLE 3.

cd [dir]	Change the shell's working directory to directory dir. If no argument is given, change to the home directory of the user.
echo [-n] list	The words in list are written to the shell's standard output, separated by space characters. The output is terminated with a newline unless the -n option is used.
exec command	Execute command in place of the current shell, which terminates.
foreach var (wordlist) ... end	The variable var is successively set to each member of wordlist. The sequence of commands between this command and the matching end is executed for each new value of var. When this command is read from the terminal, the loop is read up once prompting with? before any statements in the loop are executed.
goto label	The shell searches for a line of the form **label**: possibly preceded by space or tab characters. Execution continues after the indicated line.
if (expr) command	If the specified expression evaluates to true, the single command with arguments is executed.

if (expr) then
...
else if (expr2) then
...
else
...
endif
	If expr is true, commands up to the first else are executed. Otherwise, if expr2 is true, the commands between the else if and the second else are executed. Otherwise, commands between the else and the endif are executed. Any number of elseif pairs are allowed, but only one else. Exactly one endif is required.

onintr [- | label]	Control the action of the shell on interrupts. With no arguments, onintr restores the default action of the shell on interrupts. With the '-' argument, the shell ignores all interrupts. With a label argument, the shell executes a goto label when an interrupt is received.
repeat count command	Repeat command count times. Command is subject to the same restrictions as with the one-line **if** statement.

TABLE 3.

cd [dir]	Change the shell's working directory to directory dir. If no argument is given, change to the home directory of the user.
while (expr) ... end	While expr is true (evaluates to non-zero), repeat commands between the while and the matching end statement. **Break** and **Continue** may be used to terminate or continue the loop early.
@ [var = expr] @ [var [n] = expr]	With no arguments, display the values for all shell variables. With arguments, sets the variable var, or the n'th word in the value of var, to the value that expr evaluates to. If the expression contains the characters >, < & or |, then at least this part of expr must be placed within parentheses. All standard mathematical operators are available in this environment. Spaces are mandatory in separating components of expr that would otherwise be single words.

TABLE 4.

-r filename	Return true, or 1 if the user has read access. Otherwise it returns false, or 0.
-w filename	True if the user has write access.
-x filename	True if the user has execute permission (or search permission on a directory).
-e filename	True if file exists.
-o filename	True if the user owns file.
-z filename	True if file is of zero length (empty).
-f filename	True if file is a plain file.
-d filename	True if file is a directory.

In addition to these builtins, the functionality of the program **test** is also part of the C shell. **Test** returns true or false (i.e. , one or zero) depending on whether the criteria specified by that test are satisfied by the file passed as an argument . For instance, the exit status of **test -f** when run on a regular text file will be **1**. **Test** is used primarily in shell scripts. As a result, when included in a **while** or **if** statement, when a flag is encountered as the first expression, the operation is assumed to be a file test. The "test" statement itself is unnecessary. For example:

```
if ( -f .login )
```

is equivalent to:

```
if ( test -f .login )
```

Table 4 has a complete list of C shell file test flags.

There was a while loop in one of the scripts we've already seen. In the C shell, while and foreach are the favored methods for looping. In the script "fish.csh" , the loop looked like this:

```
while ( $i < 6 )

echo "I'm so hungry, I could eat a whole $fish[$i]\!"
@ i++

end
```

As you could tell from the output, the echo command was executed five times. Each time, the variable i was incremented by one, until the test condition of the while statement was no longer true - that is, when i reached six. The counter must be declared in the C shell before the first comparison, which is why the loop was preceded by the declaration (and assignment):

```
set i = 1
```

The same output could be generated from a **foreach** loop, with a few important differences to the script. Here's what it looks like:

```
#!/bin/csh
foreach name (tuna mackerel swordfish shark whale)
echo "I'm so hungry, I could eat a whole $name\!"
end
```

The variable **$name** is set to each of the values in the parentheses in turn. This command is particularly useful when you want to manipulate a lot of files with the same extension in an identical manner. Here's an example script you could type in an interactive C shell to rename all the C source files in the current directory:

```
prompt% foreach source ( *.c )
? mv $source $source.bak
? end
```

Exercise 7: Change the names of all the files in your names directory by adding a . jones at the end, using a **`foreach`** loop and the **mv** command.

2.6 Writing a C Shell Script

We've seen enough C shell tools to write a script of our own. We might as well write something useful. There is a utility called **compress** on the system which you can use to reduce the size of files by storing them in a special binary format that takes advantage of repetition and patterns in the original file. While files are compressed, you can't look at them normally with a pager (like **more**) without first uncompressing them, either on disk (using **uncompress**) or in memory (using **zcat**).

Compress is *very* useful for cutting down on disk usage, for those who are often near or at their quota usage maximum. The **quota** command returns current quota usage.

```
prompt% quota -v
```

The CRC course accounts you're using now don't have user quotas, but normally each user has a quota in the range of 5-10 megabytes. Once you've reached this amount of disk usage, you are considered to be over your quota and have a timeout period to get back down below it. After this time has expired, you can't write any files to your home directory.

Which brings us back to **compress**, a common way to get back below the quota limit. Compress is a poorly written program, in that it fails to detect when it's not writing any output to your home

directory when you're at your quota limit. So it creates a file to replace the file you try to compress with a zero length file. You original file is deleted, and you're left with nothing.

Exercise 8: We're going to write a script that will create a subdirectory in /**tmp**, move the file(s) we want to compress there, then move the file back to the home directory if under disk quota.
Since this is an operation you may want to run on multiple files, it would probably be a good idea to have a foreach loop to do each file one at a time. Error checking is a good idea, but not essential. There are four main tasks:
Making a subdirectory in /tmp. Mov-
ing the file(s) specified in the script argument to that directory**.** Compress-
ing each of these files**..**
Moving the file(s) back to the original directory (if under quota).

Some Hints: You'll want to make the subdirectory in **/tmp** before entering the loop, because it's an operation you'll only need to perform once. Good directory names are the current process id, or maybe the username of the person who's running the script.

It might also be useful to know that if you try to move a file into a directory where you can't write (like the home directory of a user who is over quota and whose timer has expired), mv will fail the write and keep the file in the original directory. It will also set the exit status (available in the variable **$status**) to 1. It is also important to know that compress renames each file by adding a . Z to the end of it.

Another thing to think about: What if the file entered as an argument to the script is not in the current directory, that is, if it is an absolute pathname like ~userid/lib/bigfile. Will your script deal with this possibility correctly? If not, you might want to look at the manual page for basename to figure out how to get just the name of the file, and just the original directory, so you can put the file back where you found it.

If you want to see your script execute each line, you can get a verbose play-by-play on what it's doing by putting "set echo" at the top of the script. Another way to do the same thing is by giving the shell the -x option on the interpreter invokation line of the script.

This problem isn't as hard as it sounds. It can be done without error checking in fewer than ten lines.

Problems with the C Shell

While an adequate interactive shell, csh is a poor tool for most script programming tasks. Most scripts will run up against one C shell quirk or another that will prevent them from working correctly without a lot of additional effort. Not only that, the C shell has limitations in areas that make it inferior to the Bourne shell for similar tasks. A few examples of annoying C shell bugs are included in Appendix A.

A quote from the end of the csh man page will serve to close out this section:

> Although robust enough for general use, adventures into the esoteric periphery of the C shell may reveal unexpected quirks.

3 The Bourne Shell

The C shell is similar in many ways to the Bourne shell (so phrased because the Bourne shell is the more elderly of the two), which on the system is titled **sh**. Sometimes when the Bourne shell is being referred to, it is simply called "shell" . It is the original Unix shell, and of the standard installed shell programs, it is still the most widely used for programming purposes because of the relative simplicity of its rules and its lack of unpredictable bugs (unlike **csh**).

We'll cover the subjects in this section in the same order as they were discussed in the previous one, and briefly touch on how these differences between the C shell and Bourne shell affect command behavior. Generally speaking, the **sh** lacks some of the interactive features of **csh**, but these aren't much missed by the programmer, since they aren't much use in writing scripts.

General shell function is the same in every shell, so the explanation of that process given in the last section still applies. To recap, the shell executes commands as they are given to it, according to a set of rules. Separate commands are normally separated by either newlines, or by a semi-colon. There are shell builtins, and external commands that are searched for in the PATH. The primary differences between shells are in syntax and functionality.

Exercise 9: In order for you to try out the interactive commands that serve as examples in this section, you'll need to be running the Bourne shell instead of the C shell. You can start up a Bourne shell by typing **sh** at your prompt. Don't be alarmed when it changes to a dollar sign ($). We'll learn how to fix that in a bit.

3.1 Filename Generation

Filename generation is much the same in sh as it is in csh, with a couple of exceptions. First, brace expansion ({}) doesn't exist in sh, and neither does the *~user* operator. In the Bourne shell, the tilde (~) is just another character and receives no special treatment. For interactive purposes, this would be a bit annoying, since you need to know the name of the filesystem a user's directory is on in order to cd there. For scripts, it's generally not much missed - it's just one less character that needs to be escaped to be used normally.

3.2 Variables

Variables in the Bourne shell are more straightforward and homogeneous than in the C shell, and are treated more consistently. There are no special variables that are propagated from shell to environment variables, and vice versa. There is only one command to set a normal variable, and one command to make any variable into an environment variable. In sh, to set a key equal to a value, the syntax is:

```
prompt$ key=value
```

Note that it is important that no spaces be included between the variable and the value to which it is being set. To make a variable visible to the environment, it must be "'exported". The shell builtin export is designed for this task. One possible syntax follows:

```
prompt$ EDITOR=/usr/local/bin/gnuemacs export EDITOR
```

These commands needn't follow each other on the same line, or even be in the order given in the example above. Here is an example where the environment is alerted to the existence of an environment variable before that variable is set:

```
prompt$ export PAGINATOR
prompt$ env | grep PAGINATOR
prompt$ PAGINATOR=/usr/5bin/pg
prompt$ env | grep PAGINATOR
PAGINATOR=/usr/5bin/pg
```

The Bourne shell also is more flexible in its treatment of variables than the C shell. If an unset variable is referenced, its value is assumed to be null, unlike the C shell, which will print an error and exit. In fact, there are operators that are used specifically in conjunction with variables whose status as set or unset is not known. Table 5 explains them.

Given these rules, it's easy to set a variable without going to a lot of trouble to find out which variables are set and which aren't first. Say we want to find out who the user running our script is. We could use something like:

Table 5

${parameter}	The value, if any, of the parameter is substituted. The braces are required only when the parameter is followed by a letter, digit, or underscore that is not to be interpreted as part of its name.
${parameter: -word}	If parameter is set and is nonnull, substitute its value; otherwise substitute word.
${parameter : =word}	If parameter is not set or is null set it to word; the value of the parameter is substituted.
${parameter: ?word}	If parameter is set and is nonnull, substitute its value; otherwise, print word and exit from the shell. If word is omitted, the message ‘parameter null or not set' is printed.
${parameter: +word}	If parameter is set and is nonnull, substitute word; otherwise substitute nothing.

Table 6

#	The number of positional parameters in decimal.
-	Flags supplied to the shell on invocation or by the set command.
?	The decimal value returned by the last synchronously executed command.
$	The process number of this shell.
!	The process number of the last background command invoked.

```
prompt$ PERSON=${USER:-${LOGNAME:-'whoami'}}
```

This sets the variable PERSON to be $USER if that is set, otherwise it uses $LOGNAME, unless that's not set either, in which case it uses the command whoami, which should return the effective current username. If the first part of the search works, as it does in this case when the $USER variable is set, the rest of the expression never gets called. Thus there's no penalty for using increasingly convoluted and time consuming commands as you move to the right, since the first successful (non-null) entry will preclude the others from being executed.

Like the C shell, there are some variables the Bourne shell sets automatically. All of these variables contain state information of some kind or other. They are summarized in table 6.

Here are some examples of setting Bourne shell variables:

```
prompt$ TERM=vtlOO export TERM

prompt$ TODAY='date | awk ‘{print $4}’’

prompt$ foo="foo is a commonly used variable name in
Unix...”

prompt$ NOW=${TODAY:-"No time."}
```

In the Bourne shell, the environment variable PS 1 contains the value of the current prompt. By assigning to it, we can change our prompt.

Exercise 10: Change the prompt to your userid by setting the value of the PS1 variable to that of the variable LOGNAME, or if that's not defined, use USER.

3.3 Quoting

Quoting in the Bourne shell is very similar to quoting in the C shell. Commands are read from the string between two backticks (' ')and the standard output from these commands may be used to set a variable.

No interpretation is done on the string before the string is read, except to remove backslashes (\) used to escape other characters. Escaping backticks allows nested command substitution, like so:

```
prompt$ font='grep font \'cat filelist\''
```

The backslashes inside the imbedded command protect the back ticks from immediate interpretation, so the one just before "cat" fails to match the initial one before "grep" .

Some characters naturally act as delimiters in the Bourne shell, and when such characters are encountered, they have the effect of separating one logical word from the next. Some of them have this effect as a result of the kind of operations they perform; others are chosen for readability. They are summarized in table 7.

Since these characters all have some special meaning, each can be made to stand for itself by preceding it with a backslash (\) or inserting it between a pair of quote marks (' ' or " "). As in the C shell, filename substitution is performed inside double quotes but not single quotes.

Table 7

;	Causes sequential execution of commands.
&	Used to run commands asynchronously.
( )	Parentheses are used to group commands into a single logical word.
| ^	Pipes and circumflexes redirect input and output
< >	As do greater and less than
newlines	The default record separator
spaces tabs	The default field separator

3.4 I/O Redirection: File Descriptors

The redirection of input and output in the Bourne shell is both simpler than that of the C shell, and more powerful: Simpler because there are fewer commands, more powerful because it utilizes file *descriptors* as variables to which STDIN, STDOUT, and STDERR can be assigned.

Initially, file descriptor 0 is associated with STDIN, 1 with STDOUT, and 2 with STDERR. But if any redirection is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit. For example:

```
... 2>&1
```

associates file descriptor 2 (STDERR) with the file currently associated with file descriptor 1 (STDOUT; normally both go to the tty). The order in which redirections are specified is significant. The shell evaluates redirections left-to-right.

```
... 1>xxx 2>&1
```

First associates file descriptor 1 with file xxx. It associates file descriptor 2 with the file associated with file descriptor 1 (namely, file xxx). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file xxx.

Table 8 lists the Bourne shell redirection commands.

In the C shell the ampersand is used to redirect STDERR, here it is used to manipulate file descriptors. Here's an example of using this capability to pipe STDERR to a command while leaving STDOUT alone:

```
prompt% exec 3>&1; grep string file 2>&1 1>&3 | sed s/
exp/new/ 1>&2
```

Table 8

<word	Use file word as standard input (file descriptor 0).
>word	Use file word as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.
»word	Use file word as standard output. If the file exists output is appended to it: otherwise, the file is created.
«[-] word	After parameter and command substitution is done on word, the shell input is read up t6 the first line that literally matches the resulting word, or to an EOF.
<&digit	Use the file associated with file descriptor digit as standard input. Similarly for the standard output using >&digit.
<&-	The standard input is closed. Similarly for the standard. output using'>&-'.

File descriptor three (which starts unused - there are ten file descriptors total can be used [0-9]) is first associated with file descriptor one, which initially points at our tty. Next, the STDERR of the grep command goes to file descriptor one, which is what pipes read from. while the output of file descriptor one is associated with that of three, which we just pointed at our tty. Thus STDERR from the grep is piped to sed, which puts the STDOUT back at the same place as STDERR, which points to the terminal.

Figure 1 attempts to give a picture of what's going on.

You may not need to be able to make maneuvers like this very often, but at least in the Bourne shell, if the need arises, you can. This is impossible in the C shell.

Exercise 11: There is a program called find that can be used to return all kinds of useful information about files, execute commands, and more. Because its default behavior is to work down the directory tree specified as its first argument, it is often used for entire directory structures. Output can be redirected to files and used for other purposes. However, whenever find can't change directory, it prints an error. This exercise is to change directory to your tree subdirectory (actually a link) and use find on that directory to create one file with the names of readable directories, and another with the unreadable ones given by the error messages find will print out. Try it without any redirection first to see what the output looks like. The actual find command in this case is (after you've cd'd to tree): prompt$ find . -type d -print

---

One undocumented feature you might stumble upon (if you do a lot of Bourne shell programming) is that redirecting a control structure (such as a loop, or an "if" statement) causes a subshell to be created, in which the structure is executed. As a result, variables set in that subshell aren't changed in the current shell. As an example:

```
#!/bin/sh
(stuff up here)
forvar in this that the other
      here=there
      variable=stuff
```

```
end > file

("here" and "variable" still aren't set down here...)
```

3.5 Builtins

The Bourne shell has fewer builtins than the C shell, but like input/output redirection, they are generally more powerful than their csh counterparts. Instead of foreach, there are for, while, if, and case, which are similar commands, but they differ syntactically. To compare these functions to the C shell ones we've already seen, let's revisit a couple familiar examples.

First, the simple script we started the course with:

```
#!/bin/sh
cd $HOME/csh
ls -l
```

The only difference between this script and our original script is that we are now using $HOME instead of a tilde (~) , since the Bourne shell doesn't use the tilde convention.

The fish script is very similar to what we've seen before as well:

```
#!/bin/sh
for fish in tuna mackerel swordfish shark whale
do
echo I'm so hungry, I could eat a whole ${fish}!
done
```

The Bourne shell doesn't support arrays, but this functionality isn't generally required in script writing. If you really need an array for something, it is possible to use the positional parameters (which are initially set to the arguments passed to your script) with the set command. There are other tools that do the same job, such as awk, as well. There's a script that uses a while loop and the positional parameters in your sh directory called "fishy. sh." It's kind of tricky, but demonstrates a couple of things you might want to know if you ever use positional parameters in sh programming.

An incomplete list of Bourne shell builtins is listed in table 9. for, if, case, and while are similar to the C shell versions. Read is analogous to the $< variable of csh, but is far more powerful, as you can read multiple variables simultaneously, like so:

```
read first second third extra
```

If the input typed at the keyboard now were "This is a six word sentence.", the variable assignment would be:

```
prompt$ echo $first
This
prompt$ echo $second
```

```
is
prompt$ echo $third
a
prompt$ echo $extra
six word sentence.
```

Table 9

for name [in word...] do list done	Each time a for command is executed, name is set to the next word taken from the in word list. If in word ... is omitted, then the for command executes the do list once for each positional parameter that is set (see Parameter Substitution below). Execution ends when there are no more words in the list.
case word in [pattern [| pattern] ...) list ;;] ... esac	A case command executes the list associated with the first pattern that matches word. The form of the patterns is the same as that used for filename generation except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.
if list then list [elif list then list] ... [else list] fi	The list following if is executed and, if it returns a zero exit status, the list following the first then is executed. Otherwise, the list following elif is executed and, if its value is zero, the list following the next then is executed. Failing that, the else list is executed. If no else list or then list is executed, then the if command returns a zero exit status.
while list do list done	A while command repeatedly executes the while list and, if the exit status of the last command in the list is zero, executes the do list; otherwise the loop terminates. If no commands in the do list are executed, then the while command returns a zero exit status; until may be used in place of while to negate the loop termination test.
read [name ...]	One line is read from the standard input and, using the internal field separator, IFS (normally a SPACE or TAB character), to delimit word boundaries, the first word is assigned to the first name, the second word to the second name, etc., with leftover words assigned to the last name. Lines can be continued using \newline.
shift [n]	The positional parameters are shifted to the left, from position n+1 to position 1, and so on. Previous values for $i through $n are discarded. If n is not given, it is assumed to be 1.

trap [arg] [n] ...	The command arg is to be read and executed when the shell receives signal(s) n. Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If arg is absent all trap(s) n are reset to their original values. If arg is the null string this signal is ignored by the shell and by the commands it invokes. If n is 0 the command arg is executed on exit from the shell. The trap command with no arguments prints a list of commands associated with each signal number.
. filename	Read and execute commands from filename and return. The search path specified by PATH is used to find the directory containing filename.
eval [argument ...]	The arguments are read as input to the shell and the resulting command(s) executed.

Trap is also a much more powerful command than the analogous onintr of the C shell, because it can be used to catch any (catchable) signal, not just interrupts. The form is:

```
trap action signal(s)
```

The source command of csh is similar to . of the Bourne shell. This command is particularly useful for large scripts, because it allows some degree of modularity - you can call one small, task-specific script from inside another with:

```
.taskfile
```

Exercise 12:Look over the files "whats.sh" and “rnkill.sh" in your sh directory. When you see how trap, read, and if work, with what you already know about for, you should be able to do the final exercise.

3.6 Writing a Bourne Shell Script

We're going to write the compress script again, using the Bourne shell this time. You can use your csh script as a guide if you like, for general program layout. Remember that the commands are slightly different syntactically, so you may want to refer back to table 9 to double check that your sh commands are properly formatted.

You should do error checking this time, even if you didn't do so when making your C shell script. If you want a general idea of how to do more complete error checking, use "compress.csh" in your csh directory as a reference.

When you've got it working, time your csh compression script against the sh version. If you get typical results, you'll see a strong reason for using the Bourne shell instead of the C shell. If you have your own Unix account somewhere, you may want to mail yourself the scripts you've written. The format is:

```
prompt$ mail userid < filename
```

Appendix A C Shell Bugs

1. Expression Evaluation

Consider this statement in the csh:

```
if ($?MANPAGER) setenv PAGER $MANPAGER
```

Despite your attempts to only set PAGER when you want to, the csh aborts:

```
MANPAGER: Undefined variable.
```

That's because it parses the whole line anyway AND EVALUATES IT! You have to write this:

```
if ($?MANPAGER) then
setenv PAGER $MANPAGER
endif
```

That's the same problem you have here:

```
if ($?X && $X == 'foo') echo ok
X:Undefined variable
```

This forced to write a couple nested if's. This is gross and stupid because it renders short-ciruit booleans useless. If this were really C-like, you would expect to be able to safely write such because of the common C construct:

```
if (p && p->member)
```

Undefined variables are not fatal errors in the Bourne shell, so this issue does not arise.

2. Error Handling

Wouldn't it be nice to know you had an error in your script before you ran it? That's what the -n flag is for: just check the syntax. This is especially good to make sure seldom taken segments of code are correct. Alas, the csh implementation of this doesn't work. Consider this statement:

```
exit (i)
```

Of course, they really meant

```
exit (1)
```

or just

```
exit 1
```

Either shell will complain about this. But if you hide this in an if clause, like so:

```
#!/bin/csh -fn
if (1) then
```

```
exit (i)
endif
```

The csh tells you there's nothing wrong with this script. The equivalent construct in the Bourne shell, on the other hand, tells you this:

```
#!/bin/sh -n
if (1) then
exit (i)
endif
```

/tmp/x: syntax error at line 3: '(' unexpected

3. File Descriptors

The most common problem encountered in csh programming is that you can't do file descriptor manipulation. All you are able to do is redirect stdin, or stdout, or dup stderr into stdout. Bourne-compatible shells offer you an abundance of more exotic possibilities.

(a)Writing Files

In the Bourne shell, you can open or dup random file descriptors. For example:

```
exec 2>errs.out
```

means that from then on, all of stderr goes into errs file.

Or what if you just want to throw away stderr and leave stdout alone? Pretty simple operation, eh?

```
cmd 2>/dev/null
```

Works in the Bourne shell. In the csh, you can only make a pitiful attempt like this:

```
(cmd > /dev/tty) >& /dev/null
```

But who said that stdout was my ttv? So it's wrong. This simple operation CANNOT BE DONE in the csh.

(b)Reading Files

In the csh, all you've got is $<, which reads a line from your tty. What if you've redirected? Tough noogies, you still get your tty. Read in the Bourne shell allows you to read from stdin, which catches redirection. It also means that you can do things like this:

```
exec 3<filel exec 4<file2
```

Now you can read from fd 3 and get lines from filel , or from file2 through fd 4. In modern bournelike shells, this suffices:

```
read some_var 0<&3 read another_var 0<&4
```

(c)Closing FDs

In the Bourne shell, you can close file descriptors you don't want open, like 2>&-, which isn't the same as redirecting it to /dev/null.

More Elaborate Combinations

Maybe you want to pipe stderr to a command and leave stdout alone. Not too hard an idea, right? You can't do this in the csh as I mentioned in 1. In a Bourne shell, you can do things like this:

```
exec 3>&1

grep yyy xxx 2>&1 1>&3 3>&- | sed s/file/foobar/ 1>&2
3>&-

grep: xxx: No such foobar or directory
```

Normal output would be unaffected. The closes there were in case something really cared about all it's FDs. We send stderr to the sed, and then put it back out 2.

Consider the pipeline:

A|B|C

You want to know the status of C, well, that's easy: it's in $?, or $status in csh. But if you want it from A, you're out of luck - if you're in the csh. In the Bourne shell, you can get it. Here's something I had to do where I ran dd's stderr into a grep -v pipe to get rid of the records in/out noise, but had to return the dd's exit status, not the grep's:

```
device=/dev/rmt8

dd_noise='^ [0-9]+\+[0-9]+ records (in|out)$'

exec 3>&1

status='((dd if=$device ibs=64k 2>&1 1>&3 3>&- 4>&-;
echo $? >&4) | egrep -v "$dd_noise" 1>&2 3>&- 4>&-)
4>&1'

exit $status;
```

4. Command Orthogonality

(a)Built-ins

The csh is a horrid botch with its built-ins. You can't put them together in many reasonable way. Even simple little things like this:

```
% time | echo
```

which while nonsensical, shouldn't give me this message:

```
Reset tty pgrp from 9341 to 26678
```

Others are more fun:

```
% sleep 1 | while

while: Too few arguments.

[5] 9402

% jobs
```

```
[5] 9402 Done sleep |
```

Some can even hang your shell. Try typing while you're sourcing something. Or redirecting a source command.

(b)Flow control

You can't mix flow-control and commands, like this:

```
who | while read line; do
echo "gotta $line"
done
```

You can't combine multiline things in a csh using semicolons. There's no easy way to do this

```
alias cmd 'if (foo) then bar; else snark; endif'
```

(c)Stupid non-orthogonal parsing bugs

Certain reasonable things just don't work, like this:

```
kill -1 'cat foo'
'cat foo': Ambiguous.
```

But this is ok:

```
/bin/kill -1 'cat foo'
```

There are many more of these.

5. Signals

In the csh, all you can do with signals is trap SIGINT. In the Bourne shell, you can trap any signal, or the end-of-program exit. For example, to blow away a tempfile on any of a variety of signals:

```
trap 'rm -f /usr/adm/tmp/i$$ ;
echo "ERROR: abnormal exit" ;
exit' 1 2 3 15
trap 'rm tmp.$$' 0# on program exit
```

6. Quoting

You can't quote things reasonably in the csh:

```
set foo = "Bill asked, \"How's tricks?\""
```

doesn't work. This makes it really hard to construct strings with mixed quotes in them. In the Bourne shell, this works just fine. In fact, so does this:

```
cd /mnt; /usr/ucb/finger -m -s 'ls \'u\''
```

Dollar signs cannot be escaped in doublequotes in the csh.

```
set foo = "this is a \$dollar quoted and this is $HOME
not quoted" dollar: Undefined variable.
```

You have to use backslashes for newlines, and it's just darn hard to get them into strings sometimes.

```
set foo = "this \
and that";
echo $foo
this  and that
echo "$foo"
Unmatched ". # say what???
echo $foo:q
```

You don't have these problems in the Bourne shell, where it's just fine to write things like this:

```
echo ‘This is
some text that contains
several newlines.’
```

7.Variables

There's this big difference between global (environment) and local (shell) variables. In csh, you use a totally different syntax to set one from the other.

In Bourne shell, this

```
VAR=foo cmds args
```

is the same as

```
(export VAR; VAR=foo; cmd args)
```

or csh's

```
(setenv VAR; cmd args)
```

You can't use :t, :h, etc on envariables. Watch:

```
echo Try testing with $SHELL:t
```

It's really nice to be able to say

```
${PAGER-more} or FOO=${BAR: -${BAZ}}.
```

to be able to run the user's PAGER if set, and more otherwise. You can't do this in the csh. It takes more verbiage.

You can't get the process number of the last background command from the csh. In the Bourne shell, it's $!.

8. Random Bugs

Here's one:

```
fg %?string
^Z
kill  %?string
No match.
```

Huh? Here's another

```
!%s%x%s
```

Coredump, or garbage,

If you have an alias with backquotes, and use that in backquotes in another one, you get a coredump.

Try this:

```
% repeat 3 echo "/vmu*"
/vmu*
/vmunix
/vmunix
```

What???

There are a lot, probably over 100, of these.

Appendix B Help

If you have problems working with Shell Programming, or any of the hardware, contact the Consulting Center at.348.4983, stop by Mudd 103, or send email to consult@rice.edu.