Sh - the Bourne Shell

Last modified: Tue Jul 25 12:01:12 2023

I would appreciate it if you occasionally buy me a coffee

My Ko-fi page is here

My Venmo account is @Bruce-Barnett-31

Click here to donate via paypal, and Thank you for the support

PayPal - The safer, easier way to pay online!

Check out my other tutorials on the Unix Page, and my blog

This is my tutorial on the old Bourne shell. I created an updated version for the POSIX Shell, which is the more common version. The biggest difference is that the POSIX shell allows the use of $(.....) for command substitution.

Thanks to suggestions/correctons from:
Ryan Penn
Helmut Neujahr
DJ Phasik
Dustin King
prateek goyal

Table of Contents

Click on a topic in this table to jump there. Click on the topic title to come back to the Table Of Contents.

Bourne Shell, and filename expansion
Shell Concepts
Verifying which shell you are running

Shell basics
Meta-characters and Filename expansion
Finding the executable
Quoting with the Bourne Shell
Nested quotations
Strong versus weak quoting
Quoting over several lines
Mixing quotation marks
Quotes within quotes - take two
Placing variables within strings
Variables
A subtle point
The set command
Environment Variables
Special Environment Variables
PATH - Sets searchpath
HOME - Your home directory
CDPATH - cd searchpath
IFS - Internal Field Seperator
PS1 - Normal Prompt
PS2 - Secondary Prompt
MAIL - Incoming mail
MAILCHECK - How often to check for mail
SHACCT - Accounting file
MAILPATH - searchpath for mail folders
Bourne Shell Variables - Alternate Formats
Using quoting and shell variables
Using curly braces with variables
${variable?value} - Complain if undefined
${variable-default} - Use default if undefined
${variable+value} - Change if defined
${variable=value} - Redefine if undefined
Undefining Variables
${x:-y}, ${x:=y}, ${x:?y}, ${x:+y} forms
Order of evaluation
Special Variables in the Bourne Shell
Positional Parameters $1, $2, ..., $9
$0 - Scriptname
$* - All positional parameters
$@ - All positional parameters with spaces
$# - Number of parameters

$$ - Current process ID
$! - ID of Background job
$? - error status
$- Set variables
Options and debugging
Special options
X - Bourne Shell echo flag
V - Bourne Shell verbose flag
Combining flags
U - unset variables
N - Bourne Shell non-execute flag
E - Bourne Shell exit flag
T - Bourne Shell test one command flag
A - Bourne Shell mark for export flag
K - Bourne Shell keyword flag
H - Bourne Shell hash functions flag
The $- variable
- - Bourne Shell hyphen option
Other options
C - Bourne Shell command option
S - Bourne Shell shell-session option
I - Bourne Shell shell-interactive option
R - Bourne Shell restricted shell option
P - Bourne Shell privileged shell option
unset
Bourne Shell: Status, Pipes and branches
Unnecessary process execution
$@ versus ${1+$@}
Status and Wasted Processes
Simple Flow Control
Changing Precedence
Putting it all together
Bourne Shell Flow Control Commands: If, While and Until
Commands that must be first on the line
While - loop while true
Until - loop until true
Bourne Shell Flow Control Commands
For - Repeating while changing a variable
Case - Checking multiple cases
Break and continue
Expr - Bourne Shell Expression Evaluator

Arithmetic Operators
Relational Operators
Boolean Operators
The string operator
Precedence of the Operators
Berkeley Extensions
Bourne Shell -- Functions and argument checking
Passing values by name
Exiting from a function
Checking the number of arguments
UNIX conventions for command line arguments
Checking for optional arguments
Job Control

Original version written in 1994 and published in the Sun Observer

> How to build your own complex commands from the simple commands in the UNIX toolbox. This tutorial discusses of Bourne shell programming, describing features of the original Bourne Shell. The newer POSIX shells have more features. I first wrote this tutorial before the POSIX shells were standardized. So the information describe here should work in POSIX shells as it is a subset of the POSIX specifications.
> You're not getting the most out of UNIX if you can't write shell programs!

Bourne Shell, and filename expansion

This sections covers the Bourne shell. The manual pages are only 10 pages long, so it shouldn't be difficult to learn, right? Well, apparently I'm wrong, because most of the people I know learned one shell to customize their environment, and stayed with the C shell ever since. I understand the situation. It's hard enough to learn **one** shell language, and after struggling with one shell for a while, they are hesitant to learn another shell. After a few scripts, the new user decides the C

shell is "good enough for now" and it ends right there. They never take the next step, and learn the Bourne shell. Well, perhaps this chapter will help.

The Bourne shell is considered the primary shell in scripts. All UNIX systems have it, first of all. Second, the shell is small and fast. It doesn't have the interactive features of the C shell, but so what? Interactive features aren't much use in scripts. There are also some features the Bourne shell has that the C shell doesn't have. All in all, many consider the Bourne shell to be the best shell for writing portable UNIX scripts.

Shell Concepts

What is a shell, anyway? It's simple, really. The UNIX operating system is a complex collection of files and programs. UNIX does not require any single method or interface. Many different techniques can be used. The oldest interface, which sits between the user and the software, is the shell. Twenty five years ago many users didn't even have a video terminal. Some only had a noisy, large, slow hard-copy terminal. The shell was the interface to the operating system. Shell, layer, interface, these words all describe the same concept. By convention, a shell is a user program that is ASCII based, that allows the user to specify operations in a certain sequence.

There are four important concepts in a UNIX shell:

- The user interacts with the system using a shell.
- A sequence of operations can be scripted, or automatic, by placing the operations in a script file.
- A shell is a full featured programming language, with variables, conditional statements, and the ability to execute other programs. It can be, and is, used to prototype new programs.
- A shell allows you to easily create a new program that is not a "second-class citizen," but instead is a program with all of the privileges of any other UNIX program.

  The last two points are important. DOS does not have a shell that has as many features as the Bourne shell. Also, it is impossible to write a DOS script that emulates or replaces existing commands.

  Let me give an example. Certain DOS programs understand the meta-character "*." That is, the "RENAME" command can be told

  ```
  RENAME *.OLD *.NEW
  ```

Files "A.OLD" and "B.OLD" will be renamed "A.NEW" and "B.NEW." The DOS

commands "COPY" and "MOVE" also understand the character "*." The "TYPE" and "MORE" commands, however, do not. If you wanted to create a new command, it too, would not understand that an asterisk is used to match filenames. You see, each DOS program is given the burden of understanding filename expansion. Consequently, many programs do not.

UNIX is a different story. The shell is given the burden of expanding filenames. The program that sees the filenames is unaware of the original pattern. This means all programs act consistently, as filename expansion can be used with any command. It also means a shell script can easily replace a C program, as far as the user is concerned. If you don't like the name of a UNIX utility, it is easy to create a new utility to replace the current program. If you wanted to create a program called "DIR" you could simply create a file containing

```
#!/bin/sh
ls $*
```

The shell does the hard part.

The other difference between the DOS batch file and the UNIX shell is the richness of the shell language. It is possible to do software development using the shell as the top level of the program. Not only is it possible, but it is encouraged. The UNIX philosophy of program development is to start with a shell script. Get the functionality you want. If the end results has all of the functionality, and is fast enough, then you are done. If it isn't fast enough, consider replacing part (or all) of the script with a program written in a different language (e.g. C, Perl). Just because a UNIX program is a shell script does not mean it isn't a "real" program.

Verifying which shell you are running

Because there are many shells available, it is important to learn to distinguish between the different shells. Typing commands for one shell when you are using another is bound to cause confusion. I know from personal experience. This was aggravated by the fact that many books I used to learn UNIX never mentioned that other shells existed. Therefore, the first step is to make sure you are using the proper shell.

You can execute the following command to determine your default shell (The command you type is in **boldface**):

```
% echo $SHELL
/bin/csh
```

While this identifies your default shell, it does not accurately identify the shell you

are currently using. I will give you a better way to find out later. Because this column discusses the Bourne shell, any commands discussed here will only work right if you are using the Bourne shell. You have two choices: place these commands in a Bourne shell script. That is, create a file, make the first line read

```
#!/bin/sh
```

Make the second line, and those following, contain the commands you want to test. Then make it executable by typing

```
chmod +x filename
```

Once you do this, you can test the script by typing

```
./filename
```

The second method is to create a new window (if you desire). Then type

```
sh
```

You may see a change in the characters the shell gives you as a prompt, like the example below:

```
% /bin/sh
$
```

The Bourne shell will execute each line you type, until an end of file is found. To put it another way, when you type Control-D, the Bourne shell will terminate, and return you to the shell you were previously using. This is the same action the shell takes when a script file is executed, and the end of the script file is reached.

Shell basics

The basic actions of the shell are simple. It reads a line. This is either from a file, a script, or from a user. First, meta-characters are "handled." Second, the name of the executable is found. Third, the arguments are passed to the program. Fourth, the file redirection is setup. Lastly, the program is executed.

Meta-characters and Filename expansion

As the shell reads each line, it "handles" any special characters. This includes variable evaluation (variables start with a "$)," and filename expansion. Expansion of filenames occurs when the characters "*," "?," or "[" occur in a word. A question mark matches a single character. An asterisk matches any number of characters, including none. Square brackets are used to specify a range or particular

combination of characters. Inside square brackets, a hyphen is used to specify a range or characters. Also, if the first character inside the square brackets is an exclamation point, the complement of the range is used. Let me give some examples:

Table 1

Examples of Bourne shell filename expansions

Pattern	Matches
*	Every file in the current directory
?	Files consisting of one character
??	Files consisting of two characters
??*	Files consisting of two or more characters
[abcdefg]	Files consisting of a single letter from a to g.
[gfedcba]	Same as above
[a-g]	Same as above
[a-cd-g]	Same as above
[a-zA-Z0-9]	Files that consist of a single letter or number
[!a-zA-Z0-9]	Files that consist of a single character not a letter or number
[a-zA-Z]*	Files that start with a letter
?[a-zA-Z]*	Files whose second character matches a letter.
*[0-9]	Files that end with a number
?[0-9]	Two character filename that end with a number
*.[0-9]	Files that end with a dot and a number

As you can see, the dot is not a special character. Filenames may or may not have a dot. UNIX Programers use the dot to standardize on the type of source code of each file, but that is just a convention. There is another convention, which concerns the shell:

> **Files whose name starts with a dot are not normally listed.**

Again, it is a convention, but *ls*, *find* and the various shells follow this convention. This allows some files to be "secret," or perhaps invisible, by default. You must explicitly ask for these files, by including the dot as part of the filename. The pattern ".*" matches all hidden files. Remember that two hidden files are always in every directory, "." which indicate the present directory, and ".." which indicates the directory above the current directory. If you want to match all hidden files except these two directories, there is no easy way to specify a pattern that will always match all files except the two directories. I use

```
.??
```

or

```
.[a-zA-Z]*
```

As I said, this does not match all combinations, but works most of the time. Hackers (or crackers, if you prefer) break into computers and often use strange filenames, like ". " or ".. " to hide their traces. You may not have noticed, but there was a space in these filenames. Refering to files with spaces in the names require quoting, which I will cover later. Personally, all of my hidden files are matched by ".[a-z]*" and all of my hidden directories are matched by ".[A-Z]*." This works because I made up my own convention, and always follow it.

The slash is also special, as it is used to indicate a directory path. Filename expansion does not expand to match a slash, because a slash can never be part of the filename. Also, the same rules for filename expansion of hidden files applies if the pattern follows a slash. If you want to match hidden files in a subdirectory, you must specify the explicit pattern. Table 2 lists some examples.

Table 2	
Filename Expansion with directories	
Pattern	**Matches**
*	All non-invisible files
abc/*	All non-invisible files in directory abc
abc/.*	All invisible files in directory abc
/	All non-invisible files in all subdirectories below
/.	All invisible files in all subdirectories below

Filename expansions are based on the current directory, unless the filename starts with a slash.

The Bourne shell differs from the C shell if the meta-characters do not match any file. If this happens, the pattern is passed to the program unchanged. (The C shell will either do this, or generate an error, depending on a variable).

If you are not sure how something will expand, use the *echo* command to check. It generates output more compact than *ls*, and it will not list contents of directories like *ls* will. You will also notice the output is sorted alphabetically. The shell not only expans filenames, but sorts them, for all applications.

Finding the executable

Once the shell expands the command line, it breaks up the line into words, and takes the first word as the command to be executed. (The special Bourne variable "IFS" contains the characters used to "break up" the line. Normally, this variable contains a space and a tab.) Afterwards, the first word is used as the

name of the program. If the command is specified without an explicit directory path, the shell then searches the different directories specified by the "PATH" variable, until it finds the program specified.

If you have been following the points I made, it should not surprise you that a valid UNIX command might be

```
*
```

The contents of the directory determines the results, but if you created a file called "0," which contains

```
#!/bin/sh
echo Hey! You forgot to specify the command!
```

Click here to get file: 0.sh
and if it is the first file (alphabetically) in your directory, then executing "*" would give you an error message. (Provided the current directory was in your search path).

So you see, filename expansion can be anywhere on a command line. You can execute programs with long names without typing the entire name. However, filename expansion only works if the file is in the directory you specify. If you wanted to execute the program "my_very_own_program" without typing the complete filename, you could type

```
my_* arguments
```

as long as "my_*" expanded to a unique program name. If this was in another directory, then you you have to specify the directory path:

```
/usr/local/bin/my_* arguments
```

Understanding the relationship between the shell and the programs allows you to add features. Some people create a file called "-i" in a directory. If someone then types

```
rm *
```

while in this directory, the first argument will probably be "-i." This filename is passed to the *rm* program, which assumes the hyphen indicates an argument. Therefore *rm* runs with the interactive option, protecting programs from accidental deletion.

One last point. Many DOS users complain that they can't execute

```
RENAME *.OLD *.NEW
```

I admit that this is a little awkward to do in UNIX. I'd like to say two things in defense. First, the above usage of filename expansion is an "unnatural act," as far as the UNIX philosophy is concerned. There are many advantages to the shell handling the filename expansion, and perhaps one disadvantage in one case. I believe the advantages of the UNIX philosophy far outweight the disadvanges. Second, I don't believe it is a disadvantage anyway. Renaming files like that is wrong. The only reason to do so is because DOS does it that way, and you have to to this because you are limited to 11 characters. If you must rename them, append a string to the end instead of changing the original filename. This is UNIX. You can have filenames 256 characters long, so this approach isn't a problem. So if you must rename them, use

```
for i in *.OLD
do
        mv $i $i.NEW
done
```

This alows you to undo what you did, and retains the original filename. Even better, move the files into another directory, letting them keep their original name. I would suggest you type

```
mkdir Old
mv *.OLD Old
```

This makes undoing your action very easy, and works for files of any name, and not just "*.OLD."

Quoting with the Bourne Shell

The first problem shell programmers experience is quotation marks. The standard keyboard has three quotation marks. Each one has a different purpose, and only two are used in quoting strings. Why quote at all, and what do I mean by quoting? Well, the shell understands many special characters, called meta-characters. These each have a purpose, and there are so many, beginners often suffer from meta-itis. Example: The dollar sign is a meta-character, and tells the shell the next word is a variable. If you wanted to use the dollar sign as a regular character, how can you tell the shell the dollar sign does not indicate a variable? Answer: the dollar sign must be quoted. Why? Quoted characters do not have a special meaning. Let me repeat this with emphasis.

Quoted characters do not have a special meaning

A surprising number of characters have special meanings. The lowly space, often forgotten in many books, is an extremely important meta-character. Consider the following:

```
rm -i file1 file2
```

The shell breaks this line up into four words. The first word is the command, or program to execute. The next three words are passed to the program as three arguments. The word "-i" is an argument, just like "file1." The shell treats arguments and options the same, and does not know the difference between them. In other words, the program treats arguments starting with a hyphen as special. The shell doesn't much care, except that it follows the convention. In this case, *rm* looks at the first argument, realizes it is an option because it starts with a hyphen, and treats the next two arguments as filenames. The program then starts to delete the files specifies, but firsts asks the user for permission because of the "-i" option. The use of the hyphen to indicate an option is a convention. There is no reason you can't write a program to use another character. You could use a forward slash, like DOS does, to indicate a hyphen, but then your program would not be able to distinguish between an option and a path to filename whose first characters is a slash.

Can a file have a space in the name? Absolutely. This is UNIX. There are few limitations in filenames. As far as the operating system is concerned, You can't have a filename contain a slash or a null. The shell is a different story, and one I don't plan to discuss.

Normally, a space delineates arguments. To include a space in a filename, you must quote it. Another verb used in the UNIX documentations is "escape;" this typically refers to a single character. You "quote" a string, but "escape" a meta-character. In both cases, all special characters are treated as regular characters.

Assume, for a moment, you had a file named "file1 file2," This is one file, with a space between the "1" and the "f." If this file is to be deleted, one way to quote the space is

```
rm 'file1 file2'
```

There are other ways to do the same. Most people consider the quotation mark as something you place at the beginning and end of the string. A more accurate description of the quoting process is a switch, or toggle. The following variations are all equivalent:

```
rm 'file1 file2'
rm file1' 'file2
```

```
rm f'ile1 file'2
```

In other words, when reading a shell script, you must remember the current "quoting state." The quote toggles the state. Therefore if you see a quotation mark in the middle of a line, it may be turning the toggle on or off. You must start from the beginning, and group the quotation marks in pairs.

There are two other forms or quoting. The second uses a backslash "\" which only acts to "escape" the next character. The double quotation mark is similar to the single quotes used above, but weaker. I'll explain strong and weak quotation later on. Here is the earlier example, this time using the other forms of quoting:

> rm "file1 file2"
> rm file1 file2
> rm file1" "file2

Nested quotations

A very confusing problem is placing quotation marks within quotation marks. It can be done, but it is not always consistent or logical. Quoting a double quote is perhaps the simplist, and does what you expect. These commands each output a double quote:

```
echo '"'
echo "\""
echo \"
```

The backslash is different. Look at the three ways to output a backslash:

```
echo '\'
echo "\\"
echo \\
```

As you can see, single quotes and double quotes behave differently. A double quote is weaker, and does not quote a backslash. Single quotes are different again. You can escape them with a backslash, or quote them with double quotes:

```
echo \'
echo "'"
```

The following does **not** work:

```
echo '\''
```

It is identical to

```
echo '
```

Both examples start a quoting operation, but do not end the action. In other words, the quoting function will stay toggled, and will continue until another single quote is found. If none is found, the shell will read the rest of the script, until an end of file is found.

Strong versus weak quoting

Earlier I described single quotes as strong quoting, and double quotes as weak quoting. What is the difference? Strong quoting prevents characters from having special meanings, so if you put a character inside single quotes, what you see is what you get. Therefore, if you are not sure if a character is a special character or not, use strong quotation marks.

Weak quotation marks treat most characters as plain characters, but allow certain characters (or rather meta-characters) to have a special meaning. As the earlier example illustrates, the backslash within double quotation marks **is** a special meta-character. It indicates the next character is not, so it can be used before a backslash and before a double quotation mark, escaping the special meaning. There are two other meta-characters that are allowed inside double quotation marks: the dollar sign, and the back quote.

Dollar signs indicate a variable. One important variable is "HOME" which specifies your home, or starting directory. The following examples illustrates the difference:

```
$ echo '$HOME'
$HOME
$ echo '$HOME'
$HOME
$ echo "$HOME"
/home/barnett
$ echo "$HOME"
$HOME
```

The back quote does command substitution. The string between backquotes is executed, and the results replaces the backquoted string:

```
$ echo 'The current directory is `pwd`'
The current directory is `pwd`
$ echo 'The current directory is \`pwd\`'
The current directory is \`pwd\`
$ echo "The current directory is `pwd`"
The current directory is /home/barnett
$ echo "The current directory is \`pwd\`"
The current directory is `pwd`
```

Quoting over several lines

There is a large difference between the C shell and the Bourne shell when a quote is larger than a line. The C shell is best suited for interactive sessions. Because of this, it assumes a quote ends with the end of a line, if a second quoute character is not found. The Bourne shell makes no assumptions, and only stops quoting when you specify a second quotation mark. If you are using this shell interactively, and type a quotation mark, the normal prompt changes, indicating you are inside a quote. This confused me the first time it happened. The following Bourne shell example illustrates this:

```
$ echo 'Don't do this'
> ls
> pwd
> '
Dont do this
ls
pwd
$
```

This is a minor inconvenience if you use the shell interactively, but a large benefit when writing shell scripts that contain multiple lines of quoted text. I used the C shell for my first scripts, but I soon realized how awkward the C shell was when I included a multi-line *awk* script insice the C shell script. The Bourne shell's handling of *awk* scripts was much easier:

```
#!/bin/sh
# Print a warning if any disk is more
# than 95% full.
/usr/ucb/df | tr -d '%' | awk '
# only look at lines where the first field contains a "/"
$1 ~ /\// {     if ($5 > 95) {
                printf("Warning, disk %s is %4.2f%% full\n",$6,$5);
        }
}'
```

Click here to get file: diskwarn.sh

Mixing quotation marks

Having two types of quotation marks simplifies many problems, as long as you remember how meta-characters behave. You will find that the easiest way to escape a quotation mark is to use the other form of quotation marks.

```
echo "Don't forget!"
echo 'Warning! Missing keyword: "end"'
```

Quotes within quotes - take two

Earlier I showed how to include a quote within quotes of the same kind. As you recall, you cannot place a single quote within a string terminated by single quotes. The easiest solution is to use the other type of quotation marks. But there are times when this is not possible. There is a way to do this, but it is not obvious to many people, especially those with a lot of experience in computer languages. Most languages, you see, use special characters at the beginning and end of the string, and has an escape to insert special characters in the middle of the string. The quotation marks in the Bourne shell are **not** used to define a string. There are used to **disable** or **enable** interpretation of meta-characters. You should understand the following are equivalent:

```
echo abcd
echo 'abcd'
echo ab'c'd
echo a"b"cd
echo 'a"'b"'c"'d"
```

The last example protects each of the four letters from special interpretation, and switches between strong and weak quotation marks for each letter. Letters do not need to be quoted, but I wanted a simple example. If I wanted to include a single quote in the middle of a string delineated by a single quote marks, I'd switch to the different form of quotes when that particular character is encountered. That is, I'd use the form

```
'string1"'string2"'string3'
```

where string2 is a single quote character. Here is the real example:

```
$ echo 'Strong quotes use """ and weak quotes use "'
Strong quotes use ' and weak quotes use "
```

It is confusing, but if you start at the beginning, and following through, you will see how it works.

Placing variables within strings

Change the quoting mid-stream is also very useful when you are inserting a variable in the middle of a string. You could use weak quotes:

```
echo "My home directory is $HOME, and my account is $USER"
```

You will find that this form is also useful:

```
echo 'My home directory is '$HOME', and my account is '$USER
```

When you write your first multi-line *awk* or *sed* script, and discover you want to pass the value of a variable to the middle of the script, the second form solves this problem easily.

Variables

The Bourne shell has a very simple syntax for variables:

```
variable=value
```

The characters used for variable names is limited to letters, numbers and the underscore character. It cannot start with a number. Unlike the C shell, spaces are important when defining Bourne shell variables. Whitespace (spaces, tabs or newlines) terminate the value. If you want whitespace in a variable, it must be quoted:

```
question='What is the filename? '
```

Multiple assignments can be placed on one line:

```
A=1 B=2 C=3 D=4
```

Do not put a space after the equals sign. This terminates the value. The command

```
a=date
```

sets the variable "a" to be equal to "date," but the command

```
a= date
```

sets "a" to be the empty string, and **executes** the date command. The "date" command? Yes. Which introduces...

A subtle point

Notice how two commands are executed on one line: the variable is changed, and the "date" program is executed. It is not obvious that this is valid. The manual page doesn't mention this. Even stranger is some commands can be executed, while others cannot. The "date" command is an external program. That is, the command is not built into the shell, but is an external executable. Other

commands are internal command, built into the shell. "Echo" and "export" are shell built-in commands, and can follow the variable assignment. You might see an environment variable defined like this:

```
VAR=/usr/lib; export VAR
```

But the following works just as well:

```
VAR=/usr/lib export VAR
```

Some of the built--in commands cannot be on the same line, like "for" or "if." The "echo" command does, but it may not do what you think. Don't believe me? I'll give you an example, and you have to guess what the results will be. I suspect that that 99.9999% of you would guess wrong. Put on your thinking caps. You'll need it. **UPDATE - this information gives different results than when I wrote it in 1996. See below**

Ready?

What does the following Bourne shell commands do?

```
a=one; echo $a
a=two echo $a
a=three echo $a >$a
```

I have to be honest. I failed the test myself. Well, I got partial credit. But I wrote the quiz. The first line is simple: "one" is output to the screen. The second line behaves differently. The value of variable "a" is set to "two," but the echo command outputs "one." Remember, the shell reads the lines, expands metacharacters, and then passes it to the programs. The shell treats built-in commands like external commands, and expands the meta-characters before executing the built-in commands. Therefore the second line is effectively

```
a=two echo one
```

and **then** the command is executed, which changes the value of the variable **after** it is used.

Ready for a curve ball? What does the last line do? It creates a file. The file contains the word "two." For $64,000 and a trip to Silicon Valley, what is the name of the file that is created? For those to thought the answer is "two," I'm terrible sorry, you didn't win the grand price. We do have a nice home version of this game, and a year's supply of toothpaste. The correct answer is "three." In other words

```
echo $a >$a
```

is interpreted as

```
echo second >third
```

I am not fooling you. The variable "$a" has two different values on the same line! The C shell doesn't do this, by the way.

2011 Update - I just tried this on several systems to see what happens. On an old Sun system, it behaved as I noted. Note that this was the Bourne shell, and not Bash.

I tried it using the Bash shell on a 2.2 Linux system. The third line created a file called "three" but it contained the string "one"!.

I also tried it on a Ubuntu 10.04 system, and line three generated the error

```
bash: $a: ambiguous redirect
```

2014 Update: I just tried it on a Ubuntu 14.04 system with bash 4.3.11 and the third line created a file called "one" with the contents of "one".

Just consider this an example where the behavior is unpredictable. Now let's continue in the tutorial.]

The Bourne shell evaluates metacharacters twice: one for the commands and arguments, and a second time for file redirection. Perhaps Mr. Bourne designed the shell to behave this way because he felt

```
a=output >$a
```

ought to use the **new** value of the variable, and not the value **before** the line was executed, which might be undefined, and would certainly ave undesirable results. Although the real reason is that the above command treats the "a" variable like an environment variable, and sets the variable, marks it for export, and then executes the command. Because the variable is "passed" to the command by the environment, the shell simply sets the standard output to the appropriate file, and then processes the line for variables, and lastly executing the command on the line. More on this later.

The set command

If you with to examine the values of all of your current variables, use the command "set:"

```
$ set
DISPLAY=:0.0
HOME=/home/barnett
IFS=


LD_LIBRARY_PATH=/usr/openwin/lib:/usr/openwin/lib/server
LOGNAME=barnett
MAILCHECK=600
OPENWINHOME=/usr/openwin
PATH=/home/barnett/bin:/usr/ucb:/usr/bin
PS1=$
PS2=>
PWD=/home/barnett
SHELL=/bin/csh
TERM=vt100
USER=barnett
$
```

Notice the alphabetical order of the variables, and the equals character between the variable and the value. The "set" command is one way to determine which shell you are currently using. (The C shell puts spaces between the variable and the value.) Also note the assortment of variables already defined. These are environment variables.

Environment Variables

UNIX provides a mechanism to pass information to all processes created by a parent process by using environment variables. When you log onto a system, you are given a small number of variables, predefined. You can add to this list in your shell start-up files. Every program you execute will inherit these variables. But the information flow is one-way. New UNIX users find this confusing, and cannot understand why a shell script can't change their current directory. Picture it this way: suppose you executed hundreds of programs, and they all wanted to change their environment to a different value. It should be obvious that they can't all control the same variable. Imagine hundreds of programs trying to change the directory you are currently using! Perhaps these variables ought to be called hereditary, and not environmental. Children processes inherit these values form the parents, but can never change how the parents were created. That would require time-travel, a feature not currentable available in commercial UNIX systems.

As I mentioned before, the shell command "export" is used to update environment

variables. The command

```
export a b c
```

marks the variables "a" "b" and "c," and all child processes will inherit the current value of the variable. With no arguments, it lists those variables so marked. The command "export" is necessary. Changing the value of an environment variable does not mean this change will be inherited. Example:

```
HOME=/
myprogram
```

When "myprogram" executes, the value of the "HOME" variable is not changed. However, in this example:

```
HOME=/
export HOME
myprogram
```

the program **does** get the modified value. Another way to test this is to start a new copy of the shell, and execute the "export" command. No variables are reported.

The command marks the variable. It does not copy the current value into a special location in memory. A variable can be marked for export before it is changed. That is,

```
export HOME
HOME=/
myprogram
```

works fine.

Special Environment Variables

There are several special variables the shell uses, and there are special variables the system defined for each user. SunOS and Solaris systems use different environment variables. If in doubt, check the manual pages. I'll describe some the important Solaris variables.

PATH - Sets searchpath

The "PATH" environment variable lists directories that contain commands. When you type an arbitrary command, the directories listed are searched in the order specified. The colon is used to separate directory names. An empty string corresponds to the current directory. Therefore the searchpath

```
:/usr/bin:/usr/ucb
```

contains three directories, with the current directory being searched first. This is dangerous, as someone can create a program called "ls" and if you change your current directory to the one that contains this program, you will execute this trojan horse. If you must include the current directory, place it last in the searchpath.

```
/usr/bin:/usr/ucb:
```

HOME - Your home directory

The "HOME" variable defines where the "cd" goes when it is executed without any arguments. The HOME evvironment variable is set by the login process.

CDPATH - cd searchpath

When you execute the "cd" command, and specify a directory, the shell searches for that directory inside the current working directory. You can add additional directories to this list. If the shell can't find the directory in the current directory, it will look in the list of directories inside this variable. Adding the home directory, and the directory above the current directory is useful:

```
CDPATH=$HOME:..
export CDPATH
```

IFS - Internal Field Seperator

The "IFS" variable lists the characters used to terminate a word. I discussed this briefly earlier. Normally, whitespace separates words, and this variable contains a space, a tab and a new line. Hackers find this variable interesting, because it can be used to break into computer systems. A poorly written program may carelessly execute "/bin/ps." A hacker may redefine the PATH variable, and define IFS to be "/." When the program executes "/bin/ps," the shell will treat this as "bin ps." In other words, the program "bin" is executed with "ps" as an argument. If the hacker has placed a program called "bin" in the searchpath, then the hacker gains privileged access.

PS1 - Normal Prompt

The "PS1" variable specifies the prompt printed before each command. It is normally "$." The current directory cannot be placed inside this prompt. Well, some people make a joke, and tell a new user to place a period inside this

variable. A "." **does** signifies the current directory, however, most users prefer the actual name.

PS2 - Secondary Prompt

The "PS2" environment variable defines the secondary prompt, This is the prompt you see when you execute a multi-line command, such as "for" or "if." You also see it when you forget to terminate a quote. The default value is "> ."

MAIL - Incoming mail

The "MAIL" variable specifies where your mailbox is located. It is set by the login process.

MAILCHECK - How often to check for mail

The "MAILCHECK" variable specifies how often to check for mail, in seconds. The default value is 600 seconds (10 minutes). If you set it to zero, every time the shell types a prompt, it will check for mail.

SHACCT - Accounting file

This variable defines the accounting file, used by the *acctcom* and *acctcms* commands.

MAILPATH - searchpath for mail folders

The "MAILPATH" variable lists colon-separated filenames. You can add a "%" after the filename, and specify a special prompt for each mailbox.

In addition, several environment variables are specified by the login process. "TERM" defines the terminal type, and "USER" or "LOGNAME" defines your user ID. "SHELL" defines your default shell, and "TZ" specifies your time zone. Check the manual pages, and test your own environment to find out for sure. The external program "env" prints all current environment variables.

Bourne Shell Variables - Alternate Formats

Earlier, I discussed simple variables in the Bourne shell. Now is the time to go into more detail. Suppose you wanted to append a string to a variable. That is, suppose you had a variable "X" with the value of "Accounts," but you wanted to add a string like ".old," or "_new" making "Accounts.old" or "Accounts_new," perhaps in an attempt to rename a file. The first one is easy. The second requires

a special action. In the first case, just add the string

```
mv $X $X.old
```

The second example, however, does not work:

```
mv $X $X_new # WRONG!
```

The reason? Well, the underscore character is a valid character in a variable name. Therefore the second example evaluates two variables, "X" and "X_new." If the second one is undefined, the variable will have a value of nothing, and the shell will convert it to

```
mv Accounts
```

The *mv* command will take the offered arguments, and complain, as it always wants two or more variables. A similar problem will occur if you wish to add a letter or number to the value of a variable.

Using quoting and shell variables

There are several solutions. The first is to use shell quoting. Remember, quoting starts and stops the shell from treating the enclosed string from interpretation. All that is needed is to have a quote condition start or stop between the two strings passed to the shell. Place the variable in one string, and the constant in the other. If the variable "x" has the value "home," and you want to add "run" to the end, all of the following combinations are equal to "homerun:"

```
$x"run"
$x'run'
$x\run
$x"run
$x""run
"$x"run
```

Using curly braces with variables

There is another solution, using curly braces:

```
${x}run
```

This is a common convention in UNIX programs. The C shell also uses the same feature. The UNIX *make* utility uses this in makefiles, and requires braces for all variable references longer than a single letter. (*Make* uses either curly braces or parenthesis).

This form for variables is very useful. You could standardize on it as a convention. But the real use comes from four variations of this basic form, briefly described below:

Form	Meaning
${variable?word}	Complain if undefined
${variable-word}	Use new value if undefined
${variable+word}	Opposite of the above
${variable=word}	Use new value if undefined, and redefine

Why are these forms useful? If you write shell scripts, it is good practice to gracefully handle unusual conditions. What happens if the variable "d" is not defined - and you use the command below?

```
d=`expr $d + 1`
```

You get "expr: syntax error"
The way to fix this is to have it give an error if "d" is not defined.

```
d=`expr "${d?'not defined'}" + 1`
```

The "?" generates an error: "sh: d: not defined"
If instead, you wanted it to silently use zero, use

```
d=`expr "${d-0}" + 1`
```

This uses "0" if "d" is undefined.
If you wish to set the value if it's undefined, use "="

```
echo $z
echo ${z=23}
echo $z
```

The first echo outputs a blank line. The next 2 "echo" commands output "23."
Note that you can't use

```
new=`expr "${old=0}" + 1`
```

to change the value of "old" because the expr command is run as a subshell script, and changing the value of "old" in that shell doesn;t change the value in the parent shell.
I've seen many scripts fail with strange messages if certain variables aren't defined. Preventing this is very easy, once you master these four methods of referring a Bourne shell variable. Let me describe these in more detail.

${variable?value} - Complain if undefined

The first variation is used when something unusual happens. I think of it as the "Huh???" option, and the question mark acts as the mnemonic for this action. As an example, assume the following script is executed:

```
#!/bin/sh
cat ${HOME}/Welcome
```

But suppose the environment variable "HOME" is not set. Without the question mark, you might get a strange error. In this case, the program *cat* would complain, saying file "/Welcome" does not exist. Change the script to be

```
#!/bin/sh
cat ${HOME?}/Welcome
```

and execute it, and you will get the following message instead:

```
script: HOME: parameter null or not set
```

As you can see, changing all variables of the form "$variable" to "${variable?}" provides a simple method to improve the error reporting. Better still is a message that tells the user how to fix the problem. This is done by specifying a word after the question mark. Word? Yes, the manual pages says a word. In a typical UNIX-like way, that word is very important. You can place a single word after the question mark. But only one word. Perfect for one-word insults to those who forget to set variables:

```
cat ${HOME?Dummy}/Welcome
```

This is a perfect error message if you wish to develop a reputation. Some programmers, however, prefer to keep their jobs and friends. If you fall into that category, you may prefer to give an error message that tells the user how to fix the problem. How can you do that with a single word?

Remember my discussion earlier on quoting? And how the shell will consider a whitespace to be the end of the word unless quoted? The solution should be obvious. Just quote the string, which makes the results one word:

```
cat ${HOME?"Please define HOME, and try again"}/Welcome
```

Simple, yet this makes a shell script more user-friendly.

${variable-default} - Use default if undefined

The next variation doesn't generate an error. It simply provides a variable that didn't have a value. Here is an example, with user commands in boldface:

```
$ echo Y is $Y
Y is
$ echo Y is ${Y-default}
Y is default
$ Y=new
$ echo Y is ${Y-default}
Y is new
$
```

Think of the hyphen as a mnemonic for an optional value, as the hyphen is used to specify an option on a UNIX command line. Like the other example, the word can be more than a single word. Here are some examples:

```
${b-string}
${b-$variable}
${b-"a phrase with spaces"}
${b-"A complex phrase with variables like $HOME or `date`"}
${b-`command`}
${b-`wc -l </etc/passwd`}
${b-`ypcat passwd | wc -l`}
```

Any command in this phrase is only executed if necessary. The last two examples counts the number of lines in the password file, which might indicate the maximum number of users. Remember - you can use these forms of variables in place of the simple variable reference. So instead of the command

```
echo Maximum number of users are $MAXUSERS
```

change it to

```
echo Maximum number of users are ${MAXUSERS-`wc -l </etc/passwd`}
```

If the variable is set, then the password file is never checked.

${variable+value} - Change if defined

The third variation uses a plus sign, instead of a minus. The mnemonic is "plus is the opposite of the minus." This is appropriate, as the command does act the opposite as the previous one. In other words, if the variable is set, then ignore the current value, and use the new value. This can be used as a debug aid in Bourne shell scripts. Suppose you wanted to know when a variable was set, and what the current value is. A simple way to do this is to use the *echo* command, and echo nothing when the variable has no value by using:

```
echo ${A+"Current value of A is $A"}
```

This command does print a blank line if A does not have a value. To eliminate this, use either the Berkeley version of echo, or the System V version of echo:

```
/usr/bin/echo ${A+"A = $A"}"\c"
/usr/ucb/echo -n ${A+"A = $A"}
```

${variable=value} - Redefine if undefined

Don't forget that these variations are used when you reference a variable, and do not change the value of the variable. Well, the fourth variation is different, in that it **does** change the value of the variable, if the variable is undefined. It acts like the hyphen, but if used, redefines the variable. The mnemonic for this action? The equals sign. This should be easy to remember, because the equals sign is used to assign values to variables:

```
$ echo Y is $Y
Y is
$ echo Y is ${Y=default}
Y is default
$ echo Y is $Y
Y is default
$
```

Undefining Variables

As you use these features, you may wish to test the behavior. But how do you undefine a variable that is defined? If you try to set it to an empty string:

```
A=
```

you will discover that the above tests do not help. As far as they are concerned, the variable is defined. It just has the value of nothing, or null as the manual calls it. To undefine a variable, use the *unset* command:

```
unset A
```

or if you wish to unset several variables

```
unset A B C D E F G
```

${x:-y}, ${x:=y}, ${x:?y}, ${x:+y} forms

As you can see, there is a different between a variable that has a null value, and a variable that is undefined. While it might seem that all one cares about is defined

or undefined, life is rarely so simple. Consider the following:

```
A=$B
```

If B is undefined, is A also undefined? No. Remember, the shell evaluates the variables, and then operates on the results. So the above is the same as

```
A=
```

which defines the variable, but gives it an empty, or null value. I think most scripts don't care to know the difference between undefined and null variables. They just care if the variables have a real value or not. This makes so much sense, that later versions of the Bourne shell made it easy to test for both cases by creating a slight variation of the four forms previously described: a colon is added after the variable name:

Form	Meaning
${variable:?word}	Complain if undefined or null
${variable:-word}	Use new value if undefined or null
${variable:+word}	Opposite of the above
${variable:=word}	Use new value if undefined or null, and redefine.

Notice the difference between "${b-2}" and "${b:-2}" in the following example:

```
$ # a is undefined
$ b=""
$ c="Z"
$ echo a=${a-1}, b=${b-2}, c=${c-3}
a=1, b=, c=Z
$ echo a=${a:-1}, b=${b:-2}, c=${c:-3}
a=1, b=2, c=Z
```

Order of evaluation

One last point - the special word in one of these formats is only evaluated if necessary. Therefore the *cd* and *pwd* commands in the following: is only executed if the word is executed:

```
echo ${x-`cd $HOME;pwd`}
```

Also - the evaluation occurs in the current shell, and not a sub-shell. The command above will change the current directory, but the one below will not, as it executes the commands in a new shell, which then exits.

```
echo `cd $HOME;pwd`
```

Special Variables in the Bourne Shell

Earlier, I discussed Bourne shell variables, and various ways to use them. So far I have only given you the foundation of shell programming. It's time for discussing special Bourne shell variables, which will allow you to write useful scripts. These special variables are identified by the dollar sign, and another character. If the character is a number, it's a positional parameter. If it's not a letter or number, it's a special purpose variable.

Positional Parameters $1, $2, ..., $9

The most important concept in shell scripts is passing arguments to a script. A script with no options is more limited. The Bourne shell syntax for this is simple, and similar to other shells, and *awk*. As always, the dollar sign indicates a variable. The number after the dollar sign indicates the position on the command line. That is, "$1" indicates the first parameter, and "$2" indicates the second. Suppose you wanted to create a script called *rename* that takes two arguments. Just create a file with that name, that contains the following:

```
#!/bin/sh
# rename: - rename a file
# Usage: rename oldname newname
mv $1 $2
```

Click here to get file: rename0.sh

Then execute "chmod +x rename" and you have a new UNIX program. If you want to add some simple syntax checking to this script, using the techniques I discussed earlier, change the last line to read:

```
mv ${1?"missing: original filename"} ${2?"missing new filename"}
```

This isn't very user friendly. If you do not specify the first argument, the script will report:

```
rename: 1: missing: original filename
```

As you can see, the missing variable, in this case "1," is reported, which is a little confusing. A second way to handle this is to assign the positional variables to new names:

```
#!/bin/sh
# rename: - rename a file
```

```
# Usage: rename oldname newname
oldname=$1
newname=$2
mv ${oldname:?"missing"} ${newname:?"missing"}
```

Click here to get file: rename.sh

This will report the error as follows:

```
rename: oldname: missing
```

Notice that I had to add the colons before the question mark. Earlier I mentioned how the question mark tests for undefined parameters, while the colon before the question mark complains about empty parameters as well as undefined parameters. Otherwise, the *mv* command might have complained that it had insufficient arguments.

The Bourne shell can have any number of parameters. However, the positional parameters **variables** are limited to numbers 1 through 9. You might expect that $10 refers to the tenth argument, but it is the equivalent of the value of the first argument with a zero appended to the end of the value. The other variable format, ${10}, ought to work, but doesn't. The Korn shell does support the ${10} syntax, but the Bourne shell requires work-arounds. One of these is the *shift* command. When this command is executed, the first argument is moved off the list, and lost. Therefore one way to handle three arguments follows:

```
#!/bin/sh
arg1=$1;shift;
arg2=$1;shift;
arg3=$1;shift;
echo first three arguments are $arg1 $arg2 and $arg3
```

The *shift* command can shift more than one argument; The above example could be:

```
#!/bin/sh
arg1=$1
arg2=$2
arg3=$3;shift 3
echo first three arguments are $arg1 $arg2 and $arg3
```

This technique does make it easier to add arguments, but the error message is unfriendly. All you get is "cannot shift" as an error. The proper way to handle syntax errors requires a better understanding of testing and branching, so I will postpone this problem until later.

$0 - Scriptname

There is a special positional parameter, at location zero, that contains the name of the script. It is useful in error reporting:

```
echo $0: error
```

will report "rename: error" when the *rename* script executes it. This variable is not affected by the *shift* command.

$* - All positional parameters

Another work-around for the inability for specifying parameters 10 and above is the "$*" variable. The "*" is similar to the filename meta-character, in that it matches all of the arguments. Suppose you wanted to write a script that would move any number of files into a directory. If the first argument is the directory, the following script would work:

```
#!/bin/sh
# scriptname: moveto
# usage:
# moveto directory files.....
directory=${1:?"Missing"};shift
mv $* $directory
```

Click here to get file: moveto.sh

If this script was called "moveto" then the command

```
moveto /tmp *
```

could easily move hundreds of files into the specified directory. However, if any of the files contain a space in the name, the script would not work. There is a solution, however, using the $@ variable.

$@ - All positional parameters with spaces

The "$@" variable is very similar to the the "$*" variable. Yet, there is a subtle, but important distinction. In both cases, all of the positional parameters, starting with $1, are listed, separated by spaces. If there are spaces inside the variables, then "$@" retains the spaces, while "$*" does not. An example will help. Here is a script, called *EchoArgs*, that echoes its arguments:

```
#!/bin/sh
# Scriptname: EchoArgs
# It echoes arguments
#First - make sure we are using the Berkeley style echoes
PATH=/usr/ucb:$path;export PATH
E="echo -n"
# echo the name of the script
${E} $0:
# now echo each argument, but put a space
# before the argument, and place single quotes
# around each argument
${E} " '${1-"?"}'"
${E} " '${2-"?"}'"
${E} " '${3-"?"}'"
${E} " '${4-"?"}'"
${E} " '${5-"?"}'"
${E} " '${6-"?"}'"
${E} " '${7-"?"}'"
echo
```

Click here to get file: EchoArgs.sh

Second, here is a script that tests the difference:

```
#!/bin/sh
EchoArgs $*
EchoArgs $@
EchoArgs "$*"
EchoArgs "$@"
```

Click here to get file: TestEchoArgs.sh

Now, let's execute the script with arguments that contain spaces:

```
./TestEcho "a b c" 'd e' f g
```

The script outputs the following:

```
./EchoArgs: 'a' 'b' 'c' 'd' 'e' 'f' 'g'
./EchoArgs: 'a' 'b' 'c' 'd' 'e' 'f' 'g'
./EchoArgs: 'a b c d e f g' '?' '?' '?' '?' '?' '?'
./EchoArgs: 'a b c' 'd e' 'f' 'g' '?' '?' '?'
```

As you can see, $* and $@ act the same when they are not contained in double quotes. But within double quotes, the $* variable treats spaces within variables, and spaces between variables the same. The variable $@ retains the spaces. Most of the time $* is fine. However, if your arguments will ever have spaces in them, then the $@ is required.

$# - Number of parameters

The "$#" variable is equal to the number of arguments passed to the script. If *newscript* returned $# as a results, then both

```
newscript a b c d
```

and

```
newscript "a b c" 'd e' f g
```

would report 4. The command

```
shift $#
```

"erases" all parameters because it shifts them away, so it is lost forever.

$$ - Current process ID

The variable "$$" corresponds to the process ID of the current shell running the script. Since no two processes have the same identification number, this is useful in picking a unique temporary filename. The following script selects a unique filename, uses it, then deletes it:

```
#!/bin/sh
filename=/tmp/$0.$$
cat "$@" | wc -l >$filename
echo `cat $filename` lines were found
/bin/rm $filename
```

Click here to get file: CountLines0.sh

Another use of this variable is to allow one process to stop a second process. Suppose the first process executed

```
echo $$ >/tmp/job.pid
```

A second script can kill the first one, assuming it has permissions, using

```
kill -HUP `cat /tmp/job.pid`
```

The *kill* command sends the signal specified to the indicated process. In the above case, the signal is the hang-up, or HUP signal. If you logged into a system from home, and your modem lost the connection, your shell would receive the HUP signal.

I hope you don't mind a brief discourse into signals, but these concepts are closely related, so it is worth while to cover them together. Any professional-quality script should terminate gracefully. That is, if you kill the script, there should be no extra files left over, and all of the processes should quit at the same time. Most people just put all temporary files in the */tmp* directory, and hope that eventually these files will be deleted. They will be, but sometimes the temporary files are big, and can fill up the */tmp* disk. Also some people don't mind if it takes a while for a script to finish, but if it causes the system to slow down, or it is sending a lot of error messages to the terminal, then you should stop all child processes of your script when your script is interrupted. This is done with a *trap* command, which takes one string, and any number of signals as an argument. Therefore a script that kills a second script could be written using:

```
#!/bin/sh
# execute a script that creates /tmp/job.pid
newscript &
trap 'kill -HUP `cat /tmp/job.pid`' 0 HUP INT TERM
# continue on, waiting for the other to finish
<rest of the script deleted>
```

Signals are a very crude form of inter-process communication. You can only send signals to processes running under your user name. HUP corresponds to a hang-up, INT is an interrupt, like a control-C, and TERM is the terminate command. You can use the numbers associated with these signals if you wish, which are 1, 2, and 15. Signal number zero is special. It indicates the script is finished. Therefore setting a trap at signal zero is a way to make sure some commands are done at the end of the script. Signal 1, or the HUP signal, is generally considered to be the mildest signal. Many programs use this to indicate the program should restart itself. Other signals typically mean stop soon (15), while the strongest signal (9) cannot be trapped because it means the process should stop immediately. Therefore if you kill a shell script with signal 9, it cannot clean up any temporary files, even if it wanted to.

$! - ID of Background job

The previous example with $$ requires the process to create a special filename. This is not necessary if your script launched the other script. This information is

returned in the "$!" variable. It indicates the process ID of the process executed with the ampersand, which may be called an asynchronous, or background process. Here is the way to start a background process, do something else, and wait for the background job to finish:

```
#!/bin/sh
newscript &
trap "kill -TERM $!" 0 1 2 15
# do something else
wait $!
```

I used the numbers instead of the names of the signals. I used double quotes, so that the variable $! is evaluated properly. I also used the *wait* command, which causes the shell to sleep until that process is finished. This script will run two shell processes at the same time, yet if the user presses control-C, both processes die. Most of the time, shell programmers don't bother with this. However, if you are running several processes, and one never terminates (like a "tail -f)" then this sort of control is required. Another use is to make sure a script doesn't run for a long time. You can start a command in the background, and sleep a fixed about of time. If the background process doesn't finish by then, kill it:

```
#!/bin/sh
newscript &
sleep 10
kill -TERM $!
```

The $! variable is only changed when a job is executed with a "&" at the end. The C shell does not have an equivalent of the $! variable. This is one of the reasons the C shell is not suitable for high-quality shell scripts. Another reason is the C shell has a command similar to *trap*, but it uses one command for all signals, while the Bourne shell allows you to perform different actions for different signals.

The wait command does not need an argument. If executed with no arguments, it waits for all processes to be finished. You can launch several jobs at once using

```
#!/bin/sh
job1 &
pid=$!
job2 &
pid="$pid $!"
job3 &
pid="$pid $!"
trap "kill -15 $pid" 0 1 2 15
wait
```

$? - error status

The "$?" variable is equal to the error return of the previous program. You can remember this variable, print it out, or perform different actions based on various errors. You can use this to pass information back to the calling shell by exiting a shell script with the number as an argument. Example:

```
#!/bin/sh
# this is script1
exit 12
```

Then the following script

```
#!/bin/sh
script1
echo $?
```

would print 12.

$- Set variables

The variable "$-" corresponds to certain internal variables inside the shell. I'll discuss this next.

Options and debugging

The Bourne Shell *set* command is somewhat unusual. It has two purposes: setting certain shell options, and setting positional parameters. I mentioned positional parameters earlier. These are the arguments passed to a shell script. You can think of them as an array, of which you can only see the first nine values, by using the special variables $1 through $9. As I mentioned earlier, you can use the *shift* command to discard the first one, and move $2 to $1, etc. If you want to see all of your variables, the command

```
set
```

will list all of them, including those marked for export. You can't tell which ones are marked. But the external command *env* will do this.

You can also explicitly set these variables, by using the *set* command. Therefore the Bourne shell has one array, but only one. You can place anything in this array, but you lose the old values. You can keep them, however, by a simple assignment:

```
old=$@
set a b c
```

```
# variable $1 is now equal to "a", $2=b, and $3=c
set $old
# variable $1 now has the original value, as does $2, etc.
```

This isn't perfect. If any argument has a space inside, this information isn't retained. That is, if the first argument is "a b," and the second is "c," then afterwards the first argument will be "a," the second "b," and the third "c." You may have to explicitly handle each one:

```
one=$1;two=$2;three=$3
set a b c
# argument $1 is "a", etc.
set "$one" "$two" "$three"
# argument $1, $2 and $3 are restored
```

If you wanted to clear all of the positional parameters, try this:

```
set x;shift
```

Special options

As you recall, the dollar sign is a special character in the Bourne shell. Normally, it's used to identify variables. If the variable starts with a letter, it's a normal variable. If it starts with a number, it's a positional parameter, used to pass parameters to a shell script. Earlier, I've discussed the $*, $@, $#, $$, and $! special variables. But there are another class of variables, or perhaps the proper term is flags or options. They are not read. That is, you don't use them in strings, tests, filenames, or anything like this. These variables are boolean variables, and are internal to the shell. That is, they are either true or false. You cannot assign arbitrary values to them using the "=" character. Instead, you use the *set* command. Also, you can set them and clean them, but you cannot read them. At least, not like other variables. You read them by examining the "$-" variable, which shows you which ones are set.

Excuse me, but I am going to fast. I'm teaching you how to run, before I explained walking. Let's discuss the first flag, and why it's useful.

X - Bourne Shell echo flag

If you are having trouble understanding how a shell script works, you could modify the script, adding echo commands so you can see what is happening. Another solution is to execute the script with the "x" flag. There are three ways to set this flag. The first, and perhaps easiest, is to specify the option when executing the script: To demonstrate, assume the file *script* is:

```
#!/bin/sh
a=$1
echo a is $a
```

Then if you type

```
sh -x script abc
```

the script will print out

```
a=abc
+ echo a is abc
a is abc
```

Notice that built-in commands are displayed, while external commands are displayed with a "+" before each line. If you have several commands separated by a semicolon, each part would be displayed on its own line.

The "x" variable shows you each line before it **ex**ecutes it. The second way to turn on this variable is to modify the first line of the script, i.e.:

```
#!/bin/sh -x
```

As you can see, the first way is convenient if you want to run the script once with the variable set, while the second is useful if you plan to repeat this several times in a row. A large and complex script, however, is difficult to debug when there are hundreds of lines to watch. The solution is to turn the variable on and off as needed, inside the script. The command

```
set -x
```

turns it on, while

```
set +x
```

turns the flag off again. You can, therefore, turn the "echo before execute" flag on or off when convenient.

V - Bourne Shell verbose flag

A similar flag is the "v," or verbose flag. It is also useful in debugging scripts. The difference is this: The "v" flag echoes the line as it is read, while the "x" flag causes each command to be echoed as it is executed. Let's examine this in more detail. Given the script:

```
#!/bin/sh
```

```
# comment
a=${1:-`whoami`};b=${2:-`hostname`}
echo user $a is using computer $b
```

typing "sh -x script" causes:

```
+ whoami
a=barnett
+ hostname
b=grymoire
+ echo user barnett is using computer grymoire
user barnett is using computer grymoire
```

However, "sh -v script" reports

```
#!/bin/sh
# comment
a=${1:-`whoami`};b=${2:-`hostname`}
echo user $a is using computer $b
user barnett is using computer grymoire
```

As you can see, the comments are echoed with the verbose flag. Also, each line is echoed before the variables and the commands in backquotes are evaluated. Also note the "x" command echoes the assignment to variables a and b on two lines, while the verbose flag echoed one line. Perhaps the best way to understand the difference is the verbose flag echoes the line before the shell does anything with it, while the "x" flag causes the shell to echo each **command.** Think of it as a case of Before and After.

Combining flags

You can combine the flags if you wish. Execute a script with

```
sh -x -v script
```

or more briefly

```
sh -xv script
```

Inside, you can use any of these commands

```
set -x -v
set -xv
set +x +v
set +xv
```

The first line of a script has an exception. You can use the format

```
#!/bin/sh -xv
```

but the following will not work:

```
#!/bin/sh -x -v
```

UNIX systems only pass the first argument to the interpreter. In the example above, the shell never sees the "-v" option.

U - unset variables

Another useful flag for debugging is the "u" flag. Previously, I mentioned how the variable form "${x:?}" reports an error if the variable is null or not set. Well, instead of changing every variable to this form, just use the "-u" flag, which will report an error for **any** unset variable.

N - Bourne Shell non-execute flag

A simple way to check a complex shell script is the "-n" option. If set, the shell will read the script, and parse the commands, but not execute them. If you wanted to check for syntax errors, but not execute the script, use this command.

E - Bourne Shell exit flag

I haven't discussed the exit status much. Every external program or shell script exits with a status. A zero status is normal. Any positive value is usually an error. I normally check the status when I need to, and ignore it when I don't care. You can ignore errors by simply not looking at the error status, which is the "$?" variable I mentioned last time. (If the program prints error messages, you have to redirect the messages elsewhere). Still, you may have a case where the script isn't working the way you expect. The "-e" variable can be used for this: if any error occurs, the shell script will immediately exit. This can also be used to make sure that any errors are known and anticipated. This would be very important if you wanted to modify some information, but only if no errors have happened. You wouldn't want to corrupt some important database, would you? Suppose the following script is executed:

```
#!/bin/sh
word=$1
grep $word my_file >/tmp/count
count=`wc -l </tmp/count`
echo I found $count words in my_file
```

The script searches for a pattern inside a file, and prints out how many times the

pattern is found. The *grep* program, however, exits with an error status if no words are found. If the "e" option is set, the shell terminates before executing the *count* program. If you were concerned about errors, you could set the "e" option at the beginning of the script. If you find out later that you want to ignore the error, bracket it with instructions to disable the option:

```
set +e # ignore errors
grep $word my_file >/tmp/count
set -e
```

T - Bourne Shell test one command flag

Another way to make a script exit quickly is to use the "t" option. This causes the shell to execute one more line, then exit. This would be useful if you wanted to check for the existence of a script, but didn't want it to complete. Perhaps the script takes a long time to execute, and you just care if it's there. In this case, executing

```
sh -t script
```

will do this for you.

A - Bourne Shell mark for export flag

Previously, I mentioned you had to explicitly export a variable to place it in the environment, so other programs can find it. That is, if you execute these commands

```
a=newvalue
newscript
```

The script *newscript* will now know the value of variable "a."

in the environment with

```
export a
```

A second way to do this is to assign the variable right before executing the script:

```
a=newvalue newscript
```

This is an unusual form, and not often used. There is no semicolon on the line. If there was a semicolon between the assignment and *myscript*, the variable "a" would not be made an environment variable.

Another way to do this is to set the "a" option:

```
set -a
```

If set, **all** variables that are modified or created will be exported. This could be very useful if you split one large script into two smaller scripts, and want to make sure all variables defined on one script are known to the other.

K - Bourne Shell keyword flag

While many of the options I have discussed are useful for debugging, or working around problems, other options solve subtle problems. An obscure option is the "k" switch. Consider the following Bourne shell command

```
a=1 myscript b=2 c d=3
```

When *myscript* executes, four pieces of information are passed to the program: The environment variable "a" has the value 1. Three arguments are passed to the script: "b=2," "c," and "d=3."

Any assignment on the same line as a command is made an environment variable, but only if it's before the command. The "-k" options changes this All three assignments become environment variables, and the script only sees one argument.

H - Bourne Shell hash functions flag

I've read the manual page, and was unclear. This seems to be a way to speed up program executings by pre-storing the paths for each command. The Bash manpage says this is enabled by default. "-h" option.

The $- variable

As I mentioned, you can use the *set* command to change the value of these flags. However, it cannot be used to check the values. There is a special variable, called "$-," which contains the current options. You can print the values, or test them. It has another use. Suppose you had a complex script that called other scripts. Suppose you wanted to debug all of the scripts. You could modify every script, add the option you wanted. That is, assume, *newscript* might contain

```
#!/bin/sh
myscript arg1 arg2
```

If this is replaced by

```
#!/bin/sh
```

```
sh -$- myscript arg1 arg2
```

then if you typed "sh -x newscript," *myscript* would also see the "-x" option.

- - Bourne Shell hyphen option

I should mention that you can set options as well as positional parameters on the same *set* command. That is, you can type

```
set -xvua a b c
```

There is another special option, that isn't really an option. Instead, it solves a special problem. Suppose you want one of these parameters to start with a hyphen? That is, suppose you have the following script, called *myscript*:

```
#!/bin/sh
# remember the old parameters
old=$@
set a b c
# $1, $2, $3 are changed.
# now - put them back
set $old
```

Looks simple. But what happens if you execute this script with the following arguments:

```
myscript -d abc
```

Can you see what will happen? You will get an error, when the system reports

```
-d: bad option(s)
```

The *set* command thinks the "-d" argument is a shell option. The solution is to set the special hyphen-hyphen flag. This tells the shell that the rest of the arguments are not options, but positional parameters:

```
#!/bin/sh
# remember the old parameters
old=$@
set a b c
# $1, $2, $3 are changed.
# now - put them back, NOTE the change
set -- $old
```

Other options

There are three to five additional options that can be passed to the shell, but not

changed with the *set* command. The number of options depends on the version of the operating system and shell. The "$-" variable will display some of these options. I'll discuss them individually.

C - Bourne Shell command option

The "-c" option is used if you want the shell to execute a command. This can be useful if you normally use the C shell. Simply type

```
sh -c "command"
```

You can test if the Bourne shell "-a" option really does cause variables to be exported by typing:

```
sh -ac "a=1;env"
```

Since *env* prints all environment variables, you will see that variable "a" is indeed exported, and visible to the external program *env*.

S - Bourne Shell shell-session option

When you use the shell interactively, the programs reads from standard input, not from a file. If you execute the shell with no arguments, it normally behaves like this. That is,

```
echo "echo a" | sh
```

will echo "a," while

```
echo "echo a" | sh myscript
```

will ignore standard input. The "s" option forces the shell to read standard input for commands. That is,

```
echo "echo a" | sh -s myscript
```

never executes the script *myscript*. Instead, it echoes "a" like the first example.

I - Bourne Shell shell-interactive option

Normally, the shell checks standard input, and checks to see if it's a terminal or a file. If it is a terminal, then it ignores the TERMINATE signal, which is associated with signal zero in the *trap* command. Also INTERRUPT is ignored. However, if the shell is reading from a file, these signals are not ignored. The "-i" option tells the shell to not ignore these traps.

R - Bourne Shell restricted shell option

Adding the "-r" option on some Solaris systems makes the shell a restricted shell, */usr/lib/rsh*. See the rsh(1M) manual page.

P - Bourne Shell privileged shell option

The "-p" option will not change the effective user and group to the real user and group.

unset

Once you export a variable, it becomes an environment variable. The only way to undo this is to use the *unset* command, followed by the name of the variable. Therefore

```
export JUNK
unset JUNK
```

has no effect, and the two commands cancel.

Bourne Shell: Status, Pipes and branches

Suppose you have a directory for all local executables called */usr/local/bin*. Also suppose this directory is shared by more than one machine. No problem, as long as all of the machines are the same type. However, if some are running Solaris, and others are running SunOS, you may have problems. Some executables only work for certain types of architectures (like the public domain program called *top*). You may also have different types of UNIX systems. This can be a real problem. One solution is to create a wrapper script that calls the appropriate script for the machine. That is, suppose you had a directory called */usr/local/SunOS/5.4/sun4m/bin* that contains executables for Solaris 2.4 (SunOS 5.4). If you move the executable from */usr/local/bin*, and place it in the proper directory, and replaced it with a shell script that contained:

```
#!/bin/sh
/usr/local/`uname -s`/`uname -r`/`uname -m`/bin/`basename $0` "$@"
```

Your problem would be solved. Sort of. There are four problems with this script, or rather say, four ways to improve the script.

Unnecessary process execution

The first problem is that the program *uname* is executed three times every time this script is executed. A simple solution is to use environment variables. That is, if the environment variables are set, then there is no need to execute the *uname* program:

```
#!/bin/sh -a
OS=${OS=`uname -s`}
REV=${REV=`uname -r`}
ARCH=${ARCH=`uname -m`}
CMD=`basename $0`
/usr/local/$OS/$REV/$ARCH/bin/$CMD "$@"
```

The "-a" Bourne shell option says to mark all modified variables for auto-export. This script tests three environment variables, and if not defined, it sets these variables. You could, therefore, set these variables once when you log in. Thereafter, the *uname* program need not be executed.

$@ versus ${1+$@}

Earlier, I suggested using "$@" to pass all arguments to another shell script. This works on current SunOS and Solaris machines, but other UNIX systems might not work. If you think about what I've said about quoting for a while, you should begin to see something magical is happening. If you take any other variables, and surround it by double quotes, you will get one argument. If the variable has not been set, you will get one argument containing nothing. Therefore, "$@" can't possible work correctly. Yet it does. The reason is that the shell considers the "$@" a special case. However, not all versions of the Bourne shell have the same behavior. Some convert

```
"$@"
```

to

```
""
```

if no arguments are provided. For a general purpose wrapper program, this is wrong. The *cat* program would complain that it cannot open the file provided, and would not print out the filename because there is one.

The fix for these systems is to use the pattern:

```
${1+"$@"}
```

To explain, if "$1" is defined, then replace this by "$@." If it is not defined, then do nothing. Therefore the more portable form is:

```
/usr/local/$OS/$REV/$ARCH/bin/$CMD ${1+"$@"}
```

Status and Wasted Processes

There are two more flaws in the script: one minor and one major. The first is that the script creates a new process unnecessarily. It's a small point, but if you want to optimize a commonly executed script, worthwhile. The second problem is the script does not properly return the exit status. The fix is simple. Place *exec* before the instruction:

```
exec /usr/local/$OS/$REV/$ARCH/bin/$CMD ${1+"$@"}
```

Normally, the shell creates a copy of itself for each line, and then executes the command on the line with the new process. The *exec* command does the second step, without requiring the shell to copy itself. If the *exec* command succeeds, the shell never executes the next line.

The second problem with the script is the exit status. The version without the "exec" script does not execute the *exit* command. Therefore the exit status is zero. The one with the "exec" command never exits (unless the program isn't found). Instead, the program it executes exits, and the exit status is passed to the script that called the wrapper script. Why is this important? Well, the value of the status is the basis of flow control in the Bourne shell.

Simple Flow Control

This is the other topic for this section. There are some subtle points, so bear with me. The simplest form is one of these variations

```
command1 && command2
command1 || command2
```

As I have mentioned before, a program only has two ways to pass information to another process: a file/pipe, or the exit status. A program cannot use environment variables to pass information back to the process that created it. Most of the time a process does file I/O. The exit status is another quick and convenient method.

The status is an integer from 0 to 255. The shell can either examine the integer value of an exit status, or treat the value as a boolean. Zero is true, all other values are false. If you do not provide an exit status, the system returns with the status of the last command executed.

UNIX comes with two programs called *true* and *false*, which is simply the

command "exit 0" and "exit 255." Well, the *true* program doesn't even have an exit command. Since no commands are executed, the exit status is zero. So, in essence, the *true* command does absolutely nothing, but takes nine lines, including the copyright notice, to do nothing. I should warn you, however, that if you ever create a shell script that does nothing, you risk the legal fury of the AT&T legal department for reverse engineering a program that took untold hours to develop.

That's the scoop. Using the status gives you flow control. The "&&" is often called the "and" operator. It executes the next command if the first command is true. The "||" operator is the "or" command, which executes if the command is false. To illustrate:

```
true  && echo this line IS printed
false || echo this line IS printed
true  || echo this line is NOT printed
false && echo this line is NOT printed
```

Substitute the words "and" or "or" in the above examples, and read them quietly to yourself to understand why. You can, if you wish, combine these operators one one line:

```
command && echo command succeeded || echo command failed
```

Allow me a brief discursion. The pipe character "|" is special. Several commands can be combined using pipes into something called a *pipeline*. The shell has five different mechanisms to combine pipelines into a list. You all know that the end of line character is one of the five. Here are examples of the other four:

```
cmd1 ; cmd2 ; cmd3 ; cmd4
cmd1 & cmd2 & cmd3 & cmd4
cmd1 && cmd2 && cmd3 && cmd4
cmd1 || cmd2 || cmd3 || cmd4
```

The semicolon tells the shell to operate sequentially. First "cmd1" is executed, then "cmd2," etc. Each command starts up, and runs as long as they don't need input from the previous command. The "&" command launches each process in a detached manner. The order is not sequential, and you should not assume that one command finishes before the other. The last two examples, like the first, execute sequentially, as long as the status is correct. In the "&&" example, "cmd4" is executed if all three earlier commands pass. In the "||" example, "cmd4" is executed if the first three fail. The "&&" and "||" have higher precedence than ";" and "&," but lower than "|." Therefore

```
a | b && c ; d || e | f ;
```

is evaluated as

```
(a | (b && c)) ; ((d || e) | f) ;
```

The || and && commands can be used for a simple if-then-else. Note that

```
cmd1 && cmd2 || cmd3
```

if cmd1 succeeds, and cmd2 fails, then cmd3 will be executed. To prevent this, one can use

```
cmd1 && { cmd2;exit 0; } || cmd3
```

Changing Precedence

If you want to change the precedence, or order of evaluation, you can use either curly braces or parenthesis. There are some subtle differences between these two. Syntactically, there are two differences:

```
(cmd1; cmd2) | cmd3
{ cmd1; cmd2; } | cmd3
```

Notice the semicolon at the end of the list in the curly brace. Also notice a space is required after the first "{." There is another difference: the parenthesis causes the shell to execute a new process, while the curly brace does not. You can set variables in a curly brace, and it will be known outside the braces. In the example below, the first echo prints "OLD" and the second prints "NEW:"

```
a=OLD
(a=NEW) ; echo $a
{ a=NEW;} ; echo $a
```

Putting it all together

I've explained the pieces. Now I'll try to put everything together. If you want to temporarily ignore a series of commands, without adding a "#" before each line, use the following:

```
false && {
command1
command2
command3
}
```

Change the *false* to *true*, and the commands get executed. Braces and parenthesis can be nested:

```
A && {
        B && {
                echo "A and B both passed"
        } || {
                echo "A passed, B failed"
        }
} || echo "A failed"
```

The parenthesis and curly brace are useful when you want to merge standard output of multiple commands. For instance, suppose you want to add a line containing "BEGIN" to the middle of a pipeline, before the "C" command:

```
A | B | C | D
```

The first thing a novice might try is this:

```
A | B | echo "BEGIN" | C | D
```

It doesn't work. The *echo* command is not a filter. Any input to *echo* command is discarded. Therefore the program "C" sees the word "BEGIN" but doesn't see the output of program "B," which is lost. Changing the line to

```
A | B | echo "BEGIN" ; C | D
```

also doesn't work. The program "C" doesn't have its input connected to a pipe. Therefore the shell uses the user's terminal for standard input. The fix is to use one of these forms:

```
A | B | { echo "BEGIN" ; C;} | D
A | B | (echo "BEGIN" ; C) | D
```

It takes a while to learn when this is necessary. You have to know which commands read standard input, and which generate output, and which commands have the flexibility to allow you to do either. The *cat* command allows you to specify a hyphen as an option which indicates standard input. Therefore another way to do the above is:

```
echo BEGIN >/tmp/begin
A | B | cat /tmp/begin - | C | D
```

The parenthesis is useful when you want to change a state of the process without affecting the other processes. For example - changing the current

directory. A common method of copying directory trees can be done with the *tar* command:

```
tar cvf . | ( cd newdirectory; tar xfBp - )
```

Parenthesis can also be used to change terminal permissions. If you have a printer connected to a port on the computer, and want to change the baud rate of the port to 2400, while printing a file, one method is to type:

```
(stty 2400;cat file) >/dev/printer
```

One of the more powerful uses of these techniques is combining tests with pipes. You see, taking different branches does not automatically disconnect pipelines. The C shell doesn't do this well, but the Bourne shell does. Suppose you wanted a filter to read standard input, and if the input contains a special pattern, you want the filter to add a line before the stream of information. Otherwise, you wanted to add another line after the input stream. A simple way to do this is to use the shell's features:

```
#!/bin/sh
tee /tmp/file | \
(grep MATCH /tmp/file >/dev/null && cat header - || cat - trailer )
/bin/rm /tmp/file
```

As you can see, a temporary file is created, and then the *grep* program searches the copy for a pattern. If found, the shell output's first the header, then the rest of the input stream. If not found, the stream is output, and then the trailer is appended. This works because the *grep* command reads standard reads standard input if no filename is provided as an argument. However, if a filename is provided, then standard input is ignored.

I should mention that most people test conditions using the *if* command instead of the "&&" and "||" operators. The structure might seem unusual at first, but the more you use it, the more uses you will find for it.

Bourne Shell Flow Control Commands: If, While and Until

Some might find the sequence of this tutorials very strange. All these words, and nothing on the *if* statement. Strange, but necessary if the basics are properly explained, which is my goal. Too many tutorials skim over the basics, and people soon find themselves in trouble. Previously I discussed the status returned by the commands, and introduced the various brackets used for constructing lists.

What's a list? There are three ways commands can be grouped together:

A simple command
A pipeline
A list

A *simple command* is a collection of words separated by spaces.
A *pipeline* is a group of simple commands separated by a "|" character. Earlier versions of the shell used the "^" character instead of the pipe character. Older keyboards didn't have the pipe character. Newer versions of the shell use both. The exit status of the last command is the status the pipeline returns.
A *list* is a series of pipelines, separated by &, ;, &&, or ||, and terminated by a semicolon, ampersand, or new line character.

A command can either be a simple command, or a complex command. The complex commands can be one of those listed below:

if *list* **then** *list* **fi**
if *list* **then** *list* **else** *list* **fi**
if *list* **then** *list* **elif** *list* **then** *list* **fi**
if *list* **then** *list* **elif** *list* **then** *list* **elif** *list* **then** *list* **fi**
if *list* **then** *list* **elif** *list* **then** *list* **else** *list* **fi**
if *list* **then** *list* **elif** *list* **then** *list* **elif** *list* **else** *list* **fi**
while *list* **do** *list* **done**
until *list* **do** *list* **done**

The "if" commands can be nested. Therefore the above only lists some of the combinations. Also note than *fi* is *if* backwards.

Commands that must be first on the line

One point that gave me trouble at first with the Bourne shell was this concept of list. It's a simple concept. The following words must be the first word on a line:

if
then
else
elif
fi
case
esac
for
while
until
do
done
{
}

The line doesn't have to start on the actual beginning of the line. If you have a semicolon or ampersand before the word, this does the same thing. Let me explain this another way. One way to write an *if* statement is:

```
if true
then
echo it was true
fi
```

(The indentation is done for convenience.) The same complex command may also be written:

```
if true; then echo it was true; fi
```

Both are equivalent to the description I gave earlier:

if *list* **then** *list* **fi**

A list is simply a command that ends with a semicolon **or** ends with a new line character.

The "if" command executes a list if the list after the "if" command is true. A list can have more than one command. The last one is used to make a decision:

```
if
false
false
true
then
echo this will print
fi
```

An "else" list will execute if the test condition is false. Adding an "elif" allows multiple tests. Also, lists can contain further "if" statements:

```
if testa; then
        echo testa is true
elif testb; then
        echo testb is true
else
        if testc; then
                echo testc is true
        else
                echo all tests fail
        fi
fi
```

Some times you want to perform a test if the condition is false. You might try

the following, but a syntax error would exist:

```
if condition
then
# ignore this
else
        echo condition is false
fi
```

The "if," "then" and "else" statements **must** be followed by a list. A comment is not a list. There must be a command. In this case, the ":" command, which does nothing, can be used:

```
if condition; then :; else
echo condition is false
fi
```

You may notice how I vary the indentation style. Any form is correct. Pick the one you are more comfortable with. As an aside, the C shell will allow a comment as the only statement inside a conditional block.

While - loop while true

The "if" test is performed once. If you want to loop while a test is true, then use the "while" command:

```
while mytest
do
        echo mytest is still true
done;
```

The "while" command is useful in reading input, combined with the "read" command, which reads standard input, assigns the words seen to its arguments, and returns 0, or false, when the end of file is reached. The following, therefore, echoes every line:

```
while read a
do
echo $a
done
```

This can be used to ask for a list of arguments:

```
echo "Please type the arguments"
echo "Type Control-D (EOF) when done"
args="":
echo '?'
```

```
while read a
do
args="$args $a"
echo '?'
done
echo "The arguments are $args"
```

A lot of people forget a list follows the "while" command. They assume a single command must follow the "while" keyword. Not true. The above fragment could be written:

```
echo "Please type the arguments"
echo "Type Control-D (EOF) when done"
args="":
while
echo '?'
read a
do
args="$args $a"
done
echo "The arguments are $args"
```

This places the prompt right before the query, which is good programming style. The "while" command can be combined with input redirection

```
cat file | while read a
do
file contains line $a
done
```

The "while" command can get its input from a subshell, and be placed inside a subshell. The following script

```
#!/bin/sh
(echo a b c;echo 1 2 3) | (while read a; do
echo $a $a
echo $a $a
done
) | tr a-z A-Z
```

generates the following output:

```
A B C A B C
A B C A B C
1 2 3 1 2 3
1 2 3 1 2 3
```

The following also works:

```
while read a
```

```
do
echo a=$a
done <file
```

The C shell doesn't allow this flexibility in redirecting standard input.

There are a few subtle points about the "read" and "while" commands. The following is a syntax error:

```
while true
do
# comment
done
```

There must be a command in the list between the ""do" and "done" commands. The null command ":" will fix the syntax error:

```
while true
do
:
done
```

I expected

```
echo 1 2 3 | read a; echo a is $a
```

to work, but it doesn't. At least on the systems I tried, it doesn't work. However, the following does:

```
echo 1 2 3 | ( read a; echo a is $a )
```

The following seems strange, but does work:

```
<file (
read a ;echo a is $a
read a ;echo a is $a
read a ;echo a is $a
)
```

as does:

```
( read a ;echo a is $a
read a ;echo a is $a
read a ;echo a is $a
) <file
```

Exactly three lines of the file are read. I should emphasize the that C shell cannot do this easily. It does not have a built-in mechanism to read exact one line. Also the C shell command to read a single line only reads from the

controlling terminal, and cannot be redirected to get input from a file, or a pipe, while the Bourne shell can.

What happens if the "read" command has more than one argument? The command breaks the input line into words, using whitespace as the characters between words. Each word is assigned to a different variables. If there are not enough words, the last variables are assigned an empty value. If there are too many, the last variable gets the leftovers. What gets printed if the following is executed?

```
echo 1 2 3 4 5 | (
read a b c
echo first $a
echo second $b
echo third $c
)
```

The first variable is "1," the second is "2," and the last is "3 4 5."

Together, these two commands provide a lot of flexibility, more so than what appears on the surface.

If you want to loop forever, you could do what most people do:

```
while true
do
echo never stop
done
```

This executes the empty script "true." Another way to do the same, without needing to execute another process, is to use the ":" command:

```
while :
do
echo never stop
done
```

Until - loop until true

The until command acts the same as the "while" command, except the test is inverted.

```
until testa
do
echo not yet
done
```

will echo "not yet" until "testa" becomes true (exits with a status of zero).

Next, the "for" and "case" commands.

Bourne Shell Flow Control Commands

I previously discussed the Bourne Shell commands "if," "while" and "until." Next I will discuss the other two commands used for controlling flow -- "for" and "case." The templates for these commands are:

for *name* **do** *list* **done**
for *name* **in** *word* ... **do** *list* **done**
case *word* **in esac**
case *word* **in** *pattern*) *list*;; **esac**
case *word* **in** *pattern* | *pattern*) *list*;; **esac**

For - Repeating while changing a variable

The "for" command executes the commands in a list, but changes the value of a variable for each loop. To illustrate, the command

```
for i in 1 2 3 4 5 6 7 8 9 10
do
echo $i
done
```

prints out ten lines, with a number on each line. The variable "i" is assigned the values of each of the ten numbers. This could also be written as:

```
numbers="1 2 3 4 5 6 7 8 9 10"
for i in $numbers
do
echo $i
done
```

Change

```
for i in $numbers
```

to

```
for i in $numbers $numbers
```

and the script will output twice as many lines. The variables on the are evaluated as all variables are. Change

```
for i in $numbers
```

to

```
for i in "$numbers"
```

and it will print out one line with all ten numbers, because there is only one argument after the "in" command. There are many ways to specify the list of arguments in the "for" command. You can use a Clever Thing to modify two variables in a loop, by combining the "for" command with the "set" command:

```
for args in "a A" "b B" "c C"
do
set $args
echo lower case is $1, upper case is $2
done
```

Of course you can use back-quotes to use the output of a command as the creator of the list:

```
for i in `ls file*`
```

The line-at-a-time information is lost, as all newline characters are removed. Therefore you cannot easily use the "set" command to modify two or more variables at a time. If you want to change multiple arguments from a file, you either have to use the "while read" sequence I described earlier, or read a variable, and split one variable into multiples using a command like "tr" to convert some character into spaces, and then using "set" to change several variables at once:

```
#!/bin/sh
# read the /etc/password file
# use `cat /etc/passwd`, but spaces are treated like new lines
# therefore, change spaces into _

for i in `tr ' ' '_' </etc/passwd`
do
        set `echo $i | tr ':' ' '`
        echo user: $1, UID: $3, Home Directory: $6
done
```

You can use standard input to get the list of values for the variable:

```
for a in `cat`
```

Lastly, you can combine variables, constants, and program executions:

```
for i in BEGIN $a "$b $c" `cat file` END
do
echo i is $i
```

```
done
```

Of course any variable besides "i" will work. You don't have to specify the list of values after the "in" command. If not specified, the positional arguments are used. To illustrate, if you had a script called "script1" and executed

```
script1 a b c
```

and wanted to count the number of words in each file, then you could use

```
#!/bin/sh
echo $1 has `wc -l <$1` words
echo $2 has `wc -l <$2` words
echo $3 has `wc -l <$3` words
```

Another way to do the same is as follows:

```
#!/bin/sh
for file
do
echo $file has `wc -l <$file` words
done
```

Case - Checking multiple cases

The case statement functions like a complex "if" statement, with multiple clauses. The template is:

case *word* **in** *pattern* | *pattern*) *list*;; **esac**

Let's suppose you wanted to get a yes or no answer to a question. An example is:

```
echo answer yes or no
read word
case $word in
        yes | YES )
                echo you answered yes
                ;;
        no | NO )
                echo you answered no
                ;;
esac
```

The "case" statement works on patterns, like filename matching. The above example won't let you specify a single character. You must type the full word. If you type "Yes," it will not recognize it as a yes because the first letter is upper case, and the other letters are lower case. There is a fix. The patterns

are filename patterns. Therefore you can modify the above script to be:

```
echo answer yes or no
read word
case $word in
        [Yy]* )
                echo you answered yes
                ;;
        [Nn]* )
                echo you answered no
                ;;
        * )
                echo you did not say yes or no;;
esac
```

You will notice the last test, containing an asterisk. This is the standard method of specifying a default condition for the "case" statement. The last pattern in this case will always match.

There can only be one word between "case" and "in." The following is a syntax error:

```
case a b c in
```

To prevent this error, especially when the item you are checking comes from another program, use a variable.

```
#!/bin/sh
arg="a b c"
case $arg in
[aAbBcC] ) echo this never matches;;
"a b c") echo this will match;;
esac
```

You could put quotes around "$arg," but this is not needed on my Sun system. Perhaps it is a good idea, in case some older version of the Bourne shell has a bug. You will also notice the doubled up semicolon. This is required. Consider this fragment:

```
case $arg in
a ) echo yes; b ) echo no;;
esac
```

The "b" looks like a command, as far as the shell knows, followed by an illegal ")." The ";;" is needed to tell the shell the next word found is a pattern, not a command. By the way, the following is legal:

```
case $arg in
```

```
esac
```

It doesn't do anything, but the shell accepts this syntax.

Break and continue

The "for" and "while" commands execute each portion of code more than once. If you want to test a condition, and exit from this loop, you can. You can control this by using the "break" or "continue" commands. "Break" causes the control to exit the "for" or "while" statement. The "continue" statement causes the loop to immediately start the next loop. Earlier, I used a "case" statement to check if the input is a yes or no. If neither, an error occurs. If you want to wait until you have a proper answer, the "break" statement can be used:

```
while :
do
echo "Type yes or no"
read answer
case "$answer" in
[yYnN]*) break;;
ecas
done
```

Notice I used the null command ":" instead of "/bin/true."" Both have the same function, but the null command is built into the shell.

The "continue" command causes the "for" or "while" command to go to the next loop immediately The following loop only prints the odd numbers:

```
for number in 1 2 3 4 5 6 7 8
do
case "$number" in
2|4|6|8) continue ;;
esac
echo $number
done
```

The, "for" and "while" commands can be nested. Which loops do the the "break" and "continue" commands operate on? They work on the innermost loop, but you can override this. If you place a number after the command, the number determines the depth of the loop. Take the following:

```
for number in 1 2 3 4 5 6
do
for letter in a b c d e f g
do
```

```
case $number in
3) break
esac
echo $number $letter
done
done
```

This will print every combination of letters and numbers, except it will not print any line with the number "3." However, change the break to be "break 2," and the script will only print only combinations that contain 1 or 2.

Expr - Bourne Shell Expression Evaluator

This section is on the *expr* command, used to perform expression evaluation.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic. In older versions of UNIX, it used external programs, either *awk* or the must simpler program *expr*. In UNIX System V, and therefore in Solaris, *expr* was added to the Bourne shell, which increases the speed of any Bourne shell script that uses the built-in version. If the full path-name is used, the external version is executed, causing a slight speed penalty. There are four types of operations *expr* performs:

Arithmetic
Logical
Relational
String

Most of the operators are obvious:

Operator	Type	Meaining
+	Arithmetic	Addition
-	Arithmetic	Subtraction
*	Arithmetic	Multiplication
/	Arithmetic	Division
%	Arithmetic	Remainder
=	Relational	Equal to
>	Relational	Greater Than
>=	Relational	Greater Than or Equal to
<	Relational	Less Than
<=	Relational	Less than or Equal to
!=	Relational	Not Equal to
|	Boolean	Or

&	Boolean	And
:	String	Match or substitute

Not so obvious is how to use these operators. The general form is

expression operator expression

First of all, there must be spaces between operators and expressions. The example below is incorrect:

expr 2+2

The proper form is

expr 2 + 2

which prints "4" to standard output. The second concern is the shell, which treats some of the characters as meta characters. Therefore if you use any of these characters "*&|><()" you must place a backslash before them, or put quotes around them. Examples:

```
# 2 times 10
expr 2 "*" 10
# or
expr 2 \* 10
# a complex expression
# true if both $a and $b are greater than zero
expr $a \'>' 0 \'&' $b \'>' 0
# But this gives you an error if $a or $b are undefined
# so you should use
expr "$a" '>' 0 '&' "$b" '>' 0
```

The third potential problem is quoting too many characters. The expression

expr '2 + 2'

is not "4" but "2 + 2" which is five characters long. This allows strings with spaces to be compared:

expr $a = '2 + 2'

If only one expression is given, then *expr* simply echos it.

expr no

prints the word "no."

Another potential problem is variables that contain operators. The following example, which looks like it might test if a variable is equal to an "=," generates an error:

```
A='='
# is $A an '='?
expr $A = '='
# WRONG - syntax error - same as 'expr = = ='
```

The fix is similar to the one used to fix *test*, i.e. add an extra character before the variable:

```
expr X$A = X=
```

Arithmetic Operators

Expr uses 32-bit, signed integers. Therefore negative numbers are acceptable. A common use is loops:

```
# count from 1 to 10
A=1
while [ $A -le 10 ]
do
echo $A
A=`expr $A + 1`
done
```

Relational Operators

The relational operators check to see if both expressions are integers. If so, the comparison is numeric. If one or more expressions are non-numeric, the comparison is lexical. Therefore you can compare integers or strings. The confusing part of the relational operators is deciding how to use the information. You see, the operators output to standard output, and also return an exit status. Remember how the value of true is typically non-zero, but a true exit status is zero? Well, *expr* continues this tradition. Consider the following test:

```
expr 1 = 1
```

This outputs "1," or "true" to standard output, but returns an exit status of zero. To be precise, if the expression has a value that is not equal to zero, and not equal to a null string, then the exit status is zero. An exit status of one is the opposite, i,e, the output is a null string or zero. *Expr* will return an exit status of 2 if an error

occurs. The exit status can be used in simple tests:

```
expr $A = 1 > /dev/null && echo A = 1
```

Notice how the standard output had to be discarded, otherwise the script would print out a "1." The expression can also be used in a loop:

```
A=1;
while expr $A '<=' 10 >/dev/null
do
A=`expr $A + 1`
echo $A
done
```

Relational operators always print a zero or one.

Boolean Operators

The two boolean operators operate on strings and integers. A null string and a zero are false. Anything else is true. I mentioned the status before. This is a "side-effect," because the program also sends a string to standard output. In other words, the designers of the *expr* program tried to make it as versatile as possible. Consider the following:

```
expr $a \& $b
```

If either variable is an empty string, or zero, the program prints zero. (The status is one.) If not, the program exits with a status of zero, but it does not print a one. instead, it prints the value of variable "a." Imagine Garry Owens or Don Pardo saying

> Not only does *expr* return a status, but it prints too! And it doesn't just print zero or one. It prints either zero, or the actual value of the expression, saving you time **and** money.

On the other hand, they may have simply felt printing a one would throw away information for no reason. The "|" operator also returns zero if both expressions are zero or a null string. Otherwise, it returns either the first expression, or the second. More precisely, *expr* looks at the first expression, and if not a zero or null, it returns the value, otherwise it returns the second expression, not even checking its value. This is a little confusion. A table might help. The first column is the expression. The second is the printout. The third column is the exit status:

Expression	Output	Exit Status
expr " | "	0	1

expr " | 0	0	1
expr " | 1	1	0
expr " | A	A	0
expr 0 | "	0	1
expr 0 | 1	1	0
expr 0 | A	A	0
expr 1 | anything	1	0
expr A | anything	A	0
expr " & anything	0	1
expr 0 & anything	0	1
expr 1 & "	0	1
expr 1 & 0	0	1
expr 1 & A	1	0
expr A & "	0	1
expr A & 0	0	1
expr A & 1	A	0
expr A & B	A	0

The string operator

The last operator is the string operation. It is also the most complicated. The syntax is

string : *regular-expression*

The regular expression is assumed to have a "^" in front of it, so it always is aligned with the first character of the string. The output is the numbers of characters matched in the regular expression. That is,

```
expr abc : abc
```

outputs "3," while

```
expr abc : abd
```

outputs "0." As you can see, if it doesn't match, a zero is returned, the equivalent of "false." If a match occurs, a true value (non-zero) is returns, and you know how many characters match. This can be used to count characters, letters or numbers:

```
# prints the number characters in variable a
expr "$a" : '.*'
# prints the number of lower case letters
expr "$a" : '[a-z]*'
```

```
# prints the number of lower case letters
expr "$a" : '[0-9]*'
```

Remember, the regular expression starts at the first character, so the last example will only count numbers at the beginning of the variable. If variable "a" has the value of "123abc" the expression would return a value of three. *Expr* returns the number of characters matched. Therefore, *expr* can be used to find the location of the first letter in a string:

```
expr "$x" : '[^a-zA-Z]*[a-zA-Z]'
```

Parenthesis can be used in the regular expression, like the substitute command in *sed*. If a match is found, the substring within the parenthesis is returned. Therefore, if the variable "a" has the value "123abc," the two commands below output the same string, which is "abc."

```
# if $a = 123abc, then output 'abc'
expr "$a" : '[0-9]*\([a-z]*\)'
# same as above
echo $a | sed 's/^[0-9]*\([a-z]*\)/\1/'
```

This feature is often used to extract part of a command line option. You should know that the parenthesis must have a back-slash before them, and *expr* must see them. That is, the backslash is not a signal to the shell that the following character is not a meta-character. The reason it must be excaped is to match the same syntax as *sed*, etc. Suppose you had a shell script with the following option

```
myscript -x123
```

Now suppose you wanted to extract the number from the option. You could use a *sed* script like the one above. You could also use the *expr* command:

```
x=`expr "$1" : '-x\(.*\)'`
```

The *expr* command can also be used to test if a variable is a number:

```
echo "Type in a number"
read ans
number=`expr "$ans" : "\([0-9]*\)"`
if [ "$number" != "$ans" ]; then
echo "Not a number"
elif [ "$number" -eq 0 ]; then
echo "Nothing was typed"
else
echo "$number is a fine number"
fi
```

You can combine tests. For instance, the following will truncate a string to four

characters:

```
expr "$x" : '\(....\)' \| "$x"
```

The results of each expression is another expression, so you can combine expressions. The following prints 32:

```
expr 2 + 3 \* 10
```

The *expr* command can even be used as a simple version of basename. The following will output the last part of a filename:

```
expr "$x" : '.*/\(.*\)' '|' "$x"
```

However, there is a bug in this code. If the variable "x" is equal to "/," then this evaluates to

```
expr / : '.*/\(.*\)' '|' /
```

If you remember, I pointed out this problem earlier. The shell assumes the slash is an operator, and not an expression. It complains about the syntax error. The solution is to place slashes before the variable:

```
expr //$x : '.*/\(.*\)'
```

Precedence of the Operators

You can place parenthesis around expressions to construct more complex expressions. Make sure you quote them, as the shell treats parenthesis as special characters..

```
expr \( 2 + 3 \) \* 10
```

The parenthesis overrides the default precedence, and the results is 50, to 32. These parenthesis are different than the ones used in regular expressions to perform a substitute operation. *Expr* should see the parenthesis without backslashes. The backslashes are for the shell. You could use quotes instead of backslashes. The natural precedence is grouped into six different levels, which are:

Highest Precedence
expr : expr
expr * expr

expr / expr
expr % expr
expr + expr
expr - expr
expr = expr
expr > expr
expr >= expr
expr < expr
expr <= expr
expr != expr
expr & expr
expr | expr
Lowest Precedence

Berkeley Extensions

The Berkeley version of *expr* has three special functions:

match
substr
length

The *match* operator acts like the colon. The substring operator acts like the *awk* function. The length operator returns the length of a string. I suggest you do not use them, for portability reasons. You do get this functionality if you place the directory "/usr/ucb" before either "/usr/bin" on Solaris, or "/usr/5bin" on SunOS. It doesn't matter if you use the build-in version, or the external version. Both versions support the Berkeley extensions, if the path has the "ucb" directory first. "Ucb," by the way, stands for the Univerity of California at Berkeley.

I hope I've given you some insight on how to use the *expr* command. Many people only know about the numeric calculation features. It isn't as powerful as *sed* or *awk*, but if it can do the job, you may get a performance gain because you are using an built-in function instead of an external program.

Bourne Shell -- Functions and argument checking

The original version of the Bourne shell didn't have functions. If you wanted to

perform an operation more than once, you either had to duplicate the code, or create a new shell script. There is a slight penalty for each script called, as another process must be created. You also have to know where the script is, if it isn't in the path.

The C shell has aliases, but these are limited to one line, and parsing arguments is extremely confusing. The Bourne shell solved this problem with the concept of functions. Here is an example that counts from 1 to 10, incrementing variable A in a function:

```
#!/bin/sh
inc_A() {
# Increment A by 1
        A=`expr $A + 1`
}
A=1
while [ $A -le 10 ]
do
        echo $A
        inc_A
done
```

If you wish, you can define the same function on one line:

```
inc_A() { A=`expr $A + 1`; }
```

Note that you need to add a semicolon before the curley brace, as the shell expects a list between the curley braces. Another way to remember this is "}" must be the first command on a line.

Passing an argument to a function is simple: it is identical to the mechanism used to pass an argument to a shell script. All you have to do is remember that the positional arguments are relative to the function, not the script. That is, if you have the following script:

```
#!/bin/sh
func() {echo $1;}
func $2
```

and you call it with two arguments, the script prints the second argument, because that is the first argument in the function.

Passing values by name

You can pass names of variables to functions. However, this adds a lot of complexity. You must bypass the normal shell evaluation of variables. Also, strings like "$$" have special meanings. Here is a function that increments a

specified variable

```
#!/bin/sh
inc() { eval $1=`expr $$1 + 1`; }


A=10
inc A
# variable A now has the value of 11
```

The *eval* command operates on the string

```
a=`expr $a + 1`
```

and the backslashes were added to prevent the shell from interpreting the backquotes and dollar sign too soon.

Exiting from a function

What happens if you execute an *exit* command inside a function? The same as if you executed it from anywhere else from a script. It aborts the script, and passes the value to the calling script.

Suppose you want to return a value from a function? If it's a variable, just put the value in a variable. But that's not what I'm talking about. I've shown you how to use the exit status in commands like *if* and *while*. What happens if you create a function and use it in a *if* statement? You have two choices. Normally, the function returns with the exit status of the last command. If you want to control explicitly the value, the Bourne shell has a special command called *return* that sets the status value to the value specified. If no value is specified, the status of the last command is used.

You can write a simple script to loop forever:

```
always_true() { return 0;}

while always_true
do
        echo "I just can't control myself."
        echo 'Someone stop me, please!'
done
```

Remember that the exit status of zero is a true condition in shell programming. Now that I've described functions, I'll show you how to use them to parse arguments in a shell script.

Checking the number of arguments

Let's say you have a shell script with three arguments. There are many ways to solve this problem. I'll try to provide a sampler of mechanisms, so you can use the one right for your application. One way to make sure your script will work without the right number of arguments is to use default values for the variables. This example moves a file from one directory to another. If you don't specify any arguments, it moves file "a.out" from the current directory to your home directory:

```
#!/bin/sh -x
arg1=${1:-a.out}
arg2=${2:-`pwd`}
arg3=${3:-$HOME}
mv $arg2/$arg1 $arg3
```

Short, and simple. Not many people use this form, and it can seem daunting to the beginning shell programmer. It is handy at times. If you want to demand that the first argument be specified, use the form that reports an error if an argument is missing:

```
#!/bin/sh
file_to_be_moved="$1"
arg1=${file_to_be_moved:?"filename missing"}
arg2=${2:-`pwd`}
arg3=${3:-$HOME}
mv $arg2/$arg1 $arg3
```

Notice how I added another variable with a long name. Without it, I'd get the following error:

```
1: filename missing
```

With the extra variable, I now get:

```
file_to_be_moved: filename missing
```

This is better, but perhaps is a little obscure. Perhaps a better mechanism is:

```
#!/bin/sh
arg1="a.out"
arg2=`pwd`
arg3=$HOME
if [ $# -gt 3 ]
then
    :
fi
```

```
if [ $# -eq 0 ]
then
        echo must specify at least one argument
        exit 1
else if [ $# -eq 1 ]
then
        arg1="$1";
else if [ $# -eq 2 ]
then
        arg1="$1";
        arg2="$2";
else if [ $# -eq 3 ]
then
        arg1="$1";
        arg2="$2";
        arg3="$3";
else
        echo too many arguments
        exit 1
fi

mv $arg2/$arg1 $arg3
```

Click here to get file: ShCmdChk1.sh
or perhaps:

```
#!/bin/sh
arg1="a.out"
arg2=`pwd`
arg3=$HOME
if [ $# -gt 3 ]
then
        echo too many arguments
        exit 1
fi

if [ $# -eq 0 ]
then
        echo must specify at least one argument
        exit 1
fi
[ $# -ge 1 ] && arg1=$1 # do this if 1, 2 or 3 arguments
[ $# -ge 2 ] && arg2=$2 # do this if 2 or 3 arguments
[ $# -ge 3 ] && arg3=$3 # do this if 3 arguments

mv $arg2/$arg1 $arg3
```

Click here to get file: ShCmdChk2.sh
Another way to solve the problem is to use the shift command. I'll add a function this time:

```
#!/bin/sh

usage() {
        echo `basename $0`: ERROR: $* 1>&2
        echo usage: `basename $0` 'filename [fromdir] [todir]' 1>&2
        exit 1
}
arg1="a.out"
arg2=`pwd`
arg3=$HOME

[ $# -gt 3  -o $# -lt 1 ] && usage "Wrong number of arguments"

arg1=$1;shift
[ $# -gt 0 ] && { arg2=$1;shift;}
[ $# -gt 0 ] && { arg3=$1;shift;}

mv $arg2/$arg1 $arg3
```

Click here to get file: ShCmdChk3.sh
Here is still another variation using the *case* command:

```
#!/bin/sh

usage() {
        echo `basename $0`: ERROR: $* 1>&2
        echo usage: `basename $0` 'filename [fromdir] [todir]' 1>&2
        exit 1
}
arg1="a.out"
arg2=`pwd`
arg3=$HOME

case $# in
        0) usage "must provide at least one argument";;
        1) arg1=$1;;
        2) arg1=$1;arg2=$2;;
        3) arg1=$1;arg2=$2;arg3=$3;;
        *) usage "too many arguments";;

esac
```

```
mv $arg2/$arg1 $arg3
```

Click here to get file: ShCmdChk4.sh
As you can see, there are many variations. Notice the use of "basename $0" in the *usage* function. It is very important to put the name of the script in an error message. It can be very frustrating to get an error report, but have no idea where the error is. When scripts call other scripts in a chaotic fashion, finding the culprit can be time-consuming. I also like putting the *usage* function in the beginning of a script, to help someone quickly learn how to use the script by examing the code. Also note the "1>&2" when reporting an error. This forces the output to go to standard error.

UNIX conventions for command line arguments

There are standards for command-line arguments in the UNIX world. Some commands, like *tar*, *find*, and *dd* do not follow those conventions. I could explain why they don't follow the convention, but it's ancient history. One must deal with life's disappointments, grasshopper.

If possible, you should go along with the conventions. Most shell scripts do not follow all of the conventions. Perhaps a refresher course is a good idea. Still here? I'll be brief. Optional arguments always start with a hyphen, and are a single letter or number. If an option has a value following it, the space is optional. For example, if the "o" variable takes a value, the following two examples should do the same thing:

```
program -ofile
program -o file
```

Options that do not take arguments can be combined with a hyphen. Order doesn't matter. Therefore the following examples should be equivalent:

```
program -a -b -c
program -c -b -a
program -ab -c
program -cb -a
program -cba
program -abc
```

If an option is provided that doesn't match the list of suggested options, an error should occur. Another convention that is popular is the hyphen-hyphen convention. If you want to pass a filename to a script that starts with a hyphen, use a hyphen-hyphen before it:

program -- -a

Checking for optional arguments

Let's assume we need a script that uses the letters a through c as single letter options, and "-o" as an option that requires a value. Let's also assume there are any number of arguments after the options. A simple solution might be:

```
#!/bin/sh

usage() {
        echo `basename $0`: ERROR: $* 1>&2
        echo usage: `basename $0` '[-a] [-b] [-c] [-o file]
                [file ...]' 1>&2
        exit 1
}

a= b= c= o=

while :
do
    case "$1" in
        -a) a=1;;
        -b) b=1;;
        -c) c=1;;
        -o) shift; o="$1";;
        --) shift; break;;
        -*) usage "bad argument $1";;
        *) break;;
    esac
    shift
done
# rest of script...
```

Click here to get file: ShCmdArgs3.sh
This script does not allow you to combine several single-letter options with one hyphen. Also, a space is mandatory after the "-o" option. These options can be fixed, but at a cost. The code becomes much more complicated. It does work, however:

```
#!/bin/sh

usage() {
        echo `basename $0`: ERROR: $* 1>&2
        echo usage: `basename $0` '[-[abc]] [-o file]' '[file ...]' 1>&2
```

```
        exit 1
}


inside() {
# this function returns a TRUE if $2 is inside $1
# I'll use a case statement, because this is a built-in of the shell,
# and faster.
# I could use grep:
#    echo $1 | grep -s "$2" >/dev/null
# or this
#    echo $1 | grep -qs "$2"
# or expr:
#   expr "$1" : ".*$2" >/dev/null && return 0 # true
# but case does not require another shell process
    case "$1" in
        *$2*) return 0;;
    esac
    return 1;
}


done_options=
more_options() {
        # return true(0) if there are options left to parse
        # otherwise, return false

        # check the 'short-circuit' flag
        test $done_options && return 1  # true

        # how many arguments are left?
        [ $# -eq 0 ] && return 0

        # does the next argument start with a hyphen
        inside "$1" '-' && return 0;

        # otherwise, return false
        return 1        # false
}
a= b= c= o=

while more_options "$1"
do
    case "$1" in
        --) done_options=1;;
        -[abc]*)
                inside "$1" a && a=1;
                inside "$1" b && b=1;
                inside "$1" c && c=1;
                inside "$1" "[d-zA-Z]" &&
                        usage "unknown option in $1";
```

```
                ;;
        -o?*)
                # have to extract string from argument
                o=`expr "$1" : '-o(.*)'`
                ;;
        -o)
                [ $# -gt 1 ] || usage "-o requires a value";
                shift
                o="$1";;
        -*) usage "unknown option $1";;
    esac
    shift       # each time around, pop off the option
done
# continue with script
# ...
```

Click here to get file: ShCmdArgs4.sh
One of my earlier versions of this script used *expr* instead of the *case* statement. This provided a smaller, shorter script, but the *expr* command is an external command, and the script took longer to execute. This version is more efficient, but more complicated. The question you have to ask, is it worth it? There is another option. Many UNIX systems have the *getopt* program. This allows you to handle command-line options with the standard conventions. Here is how to use it:

```
#!/bin/sh

usage() {
        echo `basename $0`: ERROR: $* 1>&2
        echo usage: `basename $0` '[-[abc]] [-o file]' '[file ...]' 1>&2
        exit 1
}

set -- `getopt "abco:" "$@"` || usage

a= b= c= o=
while :
do
        case "$1" in
        -a) a=1;;
        -b) b=1;;
        -c) c=1;;
        -o) shift; o="$1";;
        --) break;;
        esac
        shift
```

```
done
shift # get rid of --
# rest of script...
```

Click here to get file: ShCmdArgs.sh
getopt command does much of the work. It automatically reports illegal arguments. The first argument specifies the legal arguments, and any letter with a colon after it requires a value. You should note that it always places a double-hyphen after the options and before the arguments. I used a *shift* command to get rid of it. There is one bad side-effect of the *getopt* command. If you have any null arguments, or arguments with spaces, the *getopt* command loses this information. The other options do not.
I've given you a very complete collection of scripts that you can modify to handle your needs. There are many other ways to solve the problems presented here. There is no single best method to use. Hopefully, you now know several options, and can decide on the method best for you.

Job Control

I added this section afterwards, because it's so powerful, yet confusing to some people.
The Bourne shell has a very powerful way to control background processes in a script. Here are some examples.
This script lauches three jobs. If the parent job gets a HUP or TERM signal, it sends it to the child processes. Therefore if you interrupt the parent process, it will pass this signal to the child processes.

```
PIDS=
program1 & PIDS="$PIDS $!"
program2 & PIDS="$PIDS $!"
program3 & PIDS="$PIDS $!"
trap "kill -1 15 $PIDS" 1 15
```

Here's another example that lets you run three processes, but terminate them if 30 seconds elapses.

```
MYID=$$
PIDS=
(sleep 30; kill -1 $MYID) &
(sleep 5;echo A) & PIDS="$PIDS $!"
(sleep 10;echo B) & PIDS="$PIDS $!"
(sleep 50;echo C) & PIDS="$PIDS $!"
trap "echo TIMEOUT;kill $PIDS" 1
echo waiting for $PIDS
```

```
wait $PIDS
echo everything OK
```

Here's another example where you run "prog1" and "prog2" in the background, and run "prog3" several times until either "prog1" or "prog2" terminates.

```
#!/bin/sh
MYPID=$$
done=0
trap "done=1" USR1
(prog1;echo prog1 done;kill -USR1 $$) & pid1=$!
trap "done=1" USR2
(prog2;echo prog2 done;kill -USR2 $$) & pid2=$!
trap "kill -1 $pid1 $pid2" 1 15

while [ "$done" -eq  0  ]
do
        prog3
done
```

I could have simplified the example above by combining USR1 and USR2 But this way you could modify it to test for both jobs to be complete instead of just one.

Also note the trap of signal 1, which allows you to terminate the parent, and have it terminate the children. You don;t want long-running jobs hanging around, especially if you are debugging the script.

You can also use it to pass other signals to the child processes, and have them do something special. They may wait untul they get a signal, for instance. This way you can start all of the processes at the same time (nearly).
This document was translated by troff2html v0.21 on September 22, 2001.