Bourne Shell Programming

Janaka Jayawardena

The Bourne Shell

Not really a contender as an interactive shell

- Classic Bourne shell hopelessly outclassed as an interactive shell by the C shell and others
- Newer versions of the Bourne shell (and ksh) have most of the C shell's interactive usage features but these shells are not widely available

Still the shell of choice for scripting

- Available on every UNIX platform
- Perhaps less readable, but contains a very stable and predictable set of control flow structures
- Need to know for System Administration needs as most startup scripts on UNIX variants are all written in Bourne shell
- The predominantly used scripting language for the installation of software packages.

Bourne Shell

I/O Redirection

Redirecting standard output

```
command > filename
```

Redirecting standard output and standard error

```
command 2>&1 > filename
```

Redirecting standard error only

```
command 2> filename
```

Pipes

Pipe standard output

```
command | command
```

Pipe standard output and standard error

```
command 2>&1 | command
```

Background Jobs

Run a process in the background

```
command | command > file &
```

Bourne Shell Programming

Creating and running a shell script

The echo command

Variables (and environment variables)

The significance of quotes

Arithmetic operations

Manipulating path names

Intro to conditionals (the if statement)

The test command

Other forms of the if statement

The for and while loops

The case (switch) statement

Miscellaneous

Parsing command line arguments

Helpful tools

awk, sed, other misc shell utilities

Janaka Jayawardena

Creating and Running a Shell Script

Shell script

File that contains UNIX commands and shell directives that are to be executed in sequence.

Make a subshell execute the script

```
sh commandfile
```

or

```
chmod a+x commandfile
commandfile
```

To ensure that the right shell is used by system to process the file, the line:

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

must be the first line in the shell script (on most systems).

Make current shell execute the script

```
. commandfile
```

Comments in shell scripts

Anything that follows a pound sign (#) is considered to be a comment

```
# this is a line that is entirely a comment

ls  # everything from here on is a comment
```

The echo command

echo

Print out all arguments to the command

```
echo ”this is a message”

echo this is a message

echo ”this       is a       message”

echo $HOME

echo $PATH
```

set

Prints out the currently defined local variables to the shell.

env

Prints out the currently defined environment variables to the shell.

Variables (and environment variables)

Variables

By default, all variables are local (accessible only to the shell that is processing these commands). Variables hold string values.

```
variable=value

name=Jack

fullname="Jack Skellington"
```

Using Variables

The string contained in a variable is accessed by:

```
$variable
```

This is called variable expansion.

Environment Variables

All variables can be turned into environment variables by:

```
export variable
```

The values of environment variables are then visible to programs (commands, subshells, etc) run from the current shell.

```
export PATH
```

Some pre–defined environment variables:
HOME, PATH, SHELL, TERM, USER

The significance of quotes

Back quotes

Text surrounded by back quotes will be executed as through it were a command and the back quoted string will be replaced by the standard output of the command.

```
echo ‘who | wc –l‘
```

Single Quotes

Text enclosed in single quotes is treated as a single piece of text without any special processing.

```
echo ’This is a literal string’

echo ’My home directory is $HOME’
```

Double Quotes

Text enclosed in double quotes have shell variable expansions performed on them.

```
echo ”This is a literal string”

echo ”My home directory is $HOME”
```

Arithmetic Operations

Expr

Since variables are string quantities, the expr command is used in Bourne shell scripts to perform integer arithmetic operations.

```
a=5
b=6
c=‘expr $a + $b‘
echo $c
```

Some of the arithmetic operators supported by expr are:

```
exp + exp
exp – exp
exp * exp
exp / exp
exp % exp
( exp )
exp relop exp  (where relop can be: <, <=, =, !=, >=, >)
exp1 | exp2  (exp1 if it is not null or zero, exp2 if
                otherwise)
exp1 & exp2  (exp if neither exp is null or zero, 0 if
                otherwise)
```

Be careful to use back slashes to escape expr operators that clash with shell metacharacters.

Manipulating path names

basename

```
basename string suffix
```

Suffix is optional. Basename deletes any prefix ending in / and the optional suffix.

dirname

```
dirname string
```

Often found on System V only, dirname delivers all but the last level of the path name in string.

Embedding a variable in a string

Curly–braces may be used to embed a variable in a string.

```
var=value
echo prefix${var}suffix
```

Filename generation metacharacters

Used to provide wild card matching of filenames (at the shell level).

*	Match zero more characters
?	Match exactly one character
[abc]	Match exactly one of the characters
[a–z]	Match exactly one character in the specified range

Examples:

```
ls   *.c
cat   [a–f]*.?
cp   ../book/chap?.txt   .
```

Intro to conditionals (the if statement)

The basic if statement

```
if command
then
        commands
fi
```

Triggering a conditional

Most flow control constructs are triggered by the exit status of the conditional command.

```
exit status 0   ————————  OK (true)
exit status non zero ——  NO (false)
```

All UNIX commands return an exit status. Some commands use the exit status to indicate the success/failure of the command's mission.

Inquiring the exit status

The special shell variable $? contains the exit status of the most recently run command. /dev/null is a data hole, where anything sent to it disappears.

```
grep –si  sun /etc/motd  > /dev/null
echo $?
grep –s  zort /etc/motd > /dev/null
echo $?
echo $?
```

Command with known exit status

The command **true** returns an exit status of zero. The command **false** returns an exit status of 1.

The test command

test

The test command is used in Bourne Shell scripts to evaluate logical expressions. If the logical expression is true, test returns zero (0). If the logical expression is false, test returns one (1).

Compare strings

```
test string1 = string2

test string1 != string2
```

Compare integers

```
test int1 relop int2
```

where relop can be:

```
–gt, –ge, –eq, –ne, –le, –lt
```

Test path names (file/directory names)

```
test –r filename
test –w filename
test –f filename
test –d filename
test –s filename
```

Other forms of the if statement

The if–else statement

```
if command
then
        commands
else
      commands
fi
```

The if–elseif statement

```
if command
then
      commands
elif command
then
      commands
else
      commands
fi
```

There can be as many elif clauses as needed. There need not be an ending else clause.

The for and while loops

The for loop

```
for loop–index in args
do
        commands
done
```

For each string in the argument list, sets the value of the loop–index variable to that string and executes the command until the argument list if exhausted.

The while loop

```
while command
do
        commands
done
```

While the command in the while clause returns a zero exit status, execute the commands within the loop.

The case (switch) statement

```
case switch–string in
        pattern1)  commands–1 ;;
        pattern2)  commands–2;;
        pattern3)  commands–3;;
        .
        .
        .
        *)  commands ;;
esac
```

If the switch–string matches any of the patterns (1,2,3, etc), the specific command cluster is executed.

The patterns may contain filename generation metacharacters for wild-carding of strings.

The commands may be any number of commands that are ended by the ;;.

*) represents a default clause if nothing matches. This may be omitted.

Miscellaneous

Reading from the standard input into a variable

```
read variable
```

The next line from the standard input (up to and not including the newline) is read literally into the variable. Embedded spaces in the input line are preserved.

```
read variable1 variable2 variable3.....
```

The first word in the input line is stored in variable1, the next in variable2, the next in variable3, etc. If there are more words than variables, the last variables gets all the last words.

Terminating execution of the shell script

```
exit status-value
exit 0
```

When this statement is executed, the shell script will terminate, returning the specified exit status value to the calling process.

Parsing command line arguments

Command line arguments

Sometimes referred to as positional parameters, command line arguments are the space/tab separated strings that are given to the shell script when it is invoked.

```
scr foo bar
```

In the above, scr is the name of the file that is being executed and foo and bar are two command line arguments.

Accessing command line arguments from a script

A number of special variables hold the values of the command line arguments.

`$0` is the name of the script
`$1` is the value of the first command line argument
`$2` is the value of the second command line argument

The variable `$#` contains the number of arguments the script was invoked with. This number does not include the name of the script (`$0`).

The variable `$*` expands to all the command line arguments. Within a double quoted string, the variable $@ expands to all command line arguments as separately quoted strings.

Shift

The shift command moves the command line arguments to the left, losing the leftmost value. (ie. `$0` is lost, $1 becomes `$0`, `$2` becomes `$1`, and so on.)

Awk

A pattern scanning and processing language, it contains many constructs that are familiar to C programmers. Invocation is of the form:

```
awk [–f file] [–Fc] [program] [files...]
```

where:

–f file	The awk program is contained in file.
–F c	Make the character c the field separator.
program	Execute the awk commands in the argument string program.
files	Read input from the named files (standard input is the default)

Awk commands are of the form:

```
pattern {action}
```

where pattern maybe any regular expression (or complex awk expression which may also include regular expressions). When the pattern is matched in the input files, the accompanying action is executed. The special patterns BEGIN and END cause the associated actions to be executed at the beginning of and at the end of the awk run. Action is a valid awk program statement.

Awk program constructs

Statements

if (conditional) statement [else statement]
while (conditional) statement
for (variable in array) statement
for (expression ; conditional; expression) statement
break
continue
{ [statement]..... }
variable = expression
print [expression–list] [expression]
printf format [, expression–list] [expression]
next # skip remaining patterns on this input line
exit # skip the rest of the input

Expression Operators

= += –= *= /= %=	Assignment
||	OR
&&	AND
!	Negate value of expression
> >= < <=	Relational operators
== != ~ !~	Relational operators
string1 string2	String concatenation, resulting in string1string2
+ –	Plus, minus
* / %	Multiply, divide, remainder
++ —	Increment, decrement (prefix or postfix)

Awk program constructs

Some built–in variables

FILENAME	Name of current input file
FS	Input field separator character
NF	Number of fields in input record
NR	Number of input record
OFMT	Output format for numbers
OFS	Output field separator string
ORS	Output record separator string
RS	Input record separator character
$n	The nth field in the current input record
$0	The entire input record

Built–in functions

cos(expr)	Cosine of expr
exp(expr)	Exponential of expr
getline()	Reads next input line
index(s1, s2)	Position of string s2 in s1
int(expr)	Integer part of expr
length(s)	Length of string s
log(expr)	Natural logarithm of expr
sin(expr)	Sine of expr
split(s,a,c)	Split s into a[1]...a[n] on character c, return n
sprintf(fmt,....)	Format according to fmt
substr(s,m,n)	n–character substring of s beginning at position m

Sed

A stream editor. Invoked as:

```
sed [–e command ] [–f file] [–n]  [ files... ]
```

where

–e command	Command is a sed command. Multiple commands may be issued. If there is only one –e option and no –f options, the –e can be omitted.
–f file	Take the sed script from the file.
–n	Only print selected output.

Sed copies the named files (or the standard input by default) to the standard output, editing their contents with an ed–like sequence of commands of the form:

```
[ address [, address ] ]  command  [ arguments  ]
```

Address could be a regular expression or numeric line number.

Sed Commands

a\	Append lines to output until one not ending in \
b label	Branch to command :label
c\	Change lines to following text, as in the a command
d	Delete line
i\	Insert following text before next output
l	List line– make non–printing characters visible
p	Print line
q	Quit
r file	Read in file
s/old/new/f	Substitute new for old. Values for f include:
g	replace all occurrences
p	print
w file	write to file
t label	Test: branch to label if substitution made to current line
w file	Write line to file
y/str1/str2/	Replace each character from string1 with corresponding character from string2
=	Print current input line number
!cmd	Do sed command only if line not selected
:label	Set label for b and t commands
{	Treat commands up to matching } as a group