

# Bourne Shell Quick Reference Card

## I. Introduction to Shell Scripts

- A. The shell is the program that you run when you log in
- B. It is a command interpreter
- C. There are three standard shells - C, Korn and Bourne
- D. Shell prompts users, accepts command, parses, then interprets command
- E. Most common form of input is command line input  
`cat file1 file2 file3`
- F. Most commands are of the format  
`command [- option list] [argument list]`
- G. Redirection and such
  1. `<` redirect input from standard input
  2. `>` redirect output from standard output
  3. `>>` redirect output and append
  4. | "pipes" output from one command to another  
`ls -l | more`
  5. tee "pipes" output to file and standard out  
`ls -l | tee rpt2 | more`
- H. Entering commands
  1. Multiple commands can be entered on the same line if separated by ;
  2. Command can span multiple lines if \R is typed at the end of each line except the last (R stands for carriage return, i.e. ENTER). This is escape sequence.
- I. Wild card characters can be used to specify file names in commands
  1. \* 0 or more characters
  2. ? one character of any kind
  3. [, , ] list of characters to match single character
- J. Simplest scripts combine commands on single line like  
`ls -l | tee rpt2 | more`
- K. Slightly more complex script will combine commands in a file
  1. Use any text editor to create file, say `my_sc`
  2. Type commands into file, one per line (unless you use ; to separate)
  3. Save file
  4. Make file readable and executable (more later on this)  
`chmod a+rx my_sc`
  5. run script by entering path to file  
`./my_sc`  
We will make this a little easier later
- L. See examples 1 and 2

## II. Variables

- A. The statement `name=value` creates and assigns value to variable  
`SUM=12`
- B. Traditional to use all upper case characters for names
- C. Access content of variable by preceding name with \$  
`echo $SUM`
- D. Arguments go from right to left
- E. Results of commands can be assigned to variables  
`SYS=`hostname``
- F. Strings are anything delimited by ""
- G. Variables used in strings are evaluated
- H. See example 3
- I. System/standard variables
  1. Command line arguments  
Accessed by \$1 through \$9 for the first 9 command line arguments. Can access more by using the `shift` command. This makes \$1 .. \$9 reference command line arguments 2-10. It can be repeated to access a long list of arguments.
  2. \$# number of arguments passed on the command line
  3. \$- Options currently in effect (supplied to `sh` or to `set`)
  4. \$\* all the command line arguments as one long double quoted string
  5. @\$ all the command line arguments as a series of double quoted strings
  6. \$? exit status of previous command
  7. \$\$ PID of this shell's process
  8. \$! PID of most recently started background job
  9. \$0 First word, that is, name of command/script

## III. Conditional Variable Substitution

- A. `${var:-string}` Use var if set, otherwise use string
- B. `${var:=string}` Use var if set, otherwise use string and assign string to var
- C. `${var:?string}` Use var if set, otherwise print string and exit
- D. `${var:+string}` Use string if var if set, otherwise use nothing

## IV. Conditional

- A. The *condition* part can be expressed two ways. Either as  
`test condition`  
or  
`[ condition ]`  
where the spaces are significant.
- B. There are several conditions that can be tested for
  1. `-s file` file greater than 0 length
  2. `-r file` file is readable

3. `-w file` file is writable
4. `-x file` file is executable
5. `-f file` file exists and is a regular file
6. `-d file` file is a directory
7. `-c file` file is a character special file
8. `-b file` file is a block special file
9. `-p file` file is a named pipe
10. `-u file` file has SUID set
11. `-g file` file has SGID set
12. `-k file` file has sticky bit set
13. `-z string` length of string is 0
14. `-n string` length of string is greater than 0
15. `string1 = string2` string1 is equal to string2
16. `string1 != string2` string1 is different from string2
17. `string` string is not null
18. `int1 -eq int2` integer1 equals integer2
19. `int1 -ne int2` integer1 does not equal integer2
20. `int1 -gt int2` integer1 greater than integer2
21. `int1 -ge int2` integer1 greater than or equal to integer2
22. `int1 -lt int2` integer1 less than integer2
23. `int1 -le int2` integer1 less than or equal to integer2
24. `! condition` negates (inverts) condition
25. `cond1 -a cond2` true if condition1 and condition2 are both true
26. `cond1 -o cond2` true if either condition1 or condition2 are true
27. `\( \)` used to group complex conditions

## V. Flow Control

- A. The if statement  
`if condition`  
`then`  
`commands`  
`else`  
`commands`  
`fi`
- B. Both the while and until type of loop structures are supported  
`while condition`  
`do`  
`commands`  
`done`  
  
`until condition`  
`do`  
`commands`  
`done`

- C. The case statement is also supported

```
case string in
pattern1)
commands
;;
```

```
pattern2)
commands
;;
```

```
esac
```

The pattern can either be an integer or a single quoted string

The \* is used as a catch-all default pattern

- D. The for command
- ```
for var [in list]
do
commands
done
```

where either a list (group of double quoted strings) is specified, or \$@ is used

## VI. Other Commands

- A. Output
1. Use the `echo` command to display data
  2. `echo "This is some data"` will output the string
  3. `echo "This is data for the file = $FILE"` will output the string and expand the variable first. The output from an `echo` command is automatically terminated with a newline.
- B. Input
1. The `read` command reads a line from standard input
  2. Input is parsed by whitespace, and assigned to each respective variable passed to the `read` command
  3. If more input is present than variables, the last variable gets the remainder
  4. If for instance the command was `read a b c` and you typed "Do you Grok it" in response, the variables would contain `$a="Do"`, `$b="you"` `$c="Grok it"`
- C. Set the value of variables \$1 thru \$n
1. If you do `set `command``, then the results for the command will be assigned to each of the variables \$1, \$2, etc. parsed by whitespace
- D. Evaluating expressions
1. The `expr` command is used to evaluate expressions

2. Useful for integer arithmetic in shell scripts `i=`expr $i +1``
- E. Executing arguments as shell commands

  1. The `eval` command executes its arguments as a shell command

---

## VII. Shell functions

- A. General format is
- B. `function_name ()`
- C. `{`
- D. `commands`
- E. `}`

---

## VIII. Miscellaneous

- A. `\n` at end of line continues on to next line
- B. Metacharacters
1. \* any number of characters
  2. ? any one character
  3. [.] list of alternate characters for one character position
- C. Substitution
1. `delimit with ``` (back quote marks, generally top left corner of keyboard)
  2. `executes what is in ``` and substitutes result in string
- D. Escapes
1. `\` single character
  2. `'` groups of characters
  3. `"` groups of characters, but some special characters processed (`$\`)
- E. Shell options
1. Restricted shell `sh -r`
    - a. can't `cd`, modify `PATH`, specify full path names or redirect output
    - b. should not allow write permissions to directory
  2. Changing shell options
    - a. Use `set` option `+/-` to turn option on/off
    - b. `e` interactive shell
    - c. `f` filename substitution
    - d. `n` run, no execution of commands
    - e. `u` unset variables as errors during substitution
    - f. `x` prints commands and arguments during execution

## Examples

### Single Line Script

```
#!/bin/sh
# Script lists all files in current
directory in decending order by size
```

```
ls -l | sort -r -n +4 -5
```

---

### Multiline Script

```
#!/usr/bin/ksh
# Lists 10 largest files in current
directory by size
```

```
ls -l > /tmp/f1
sort -r -n +4 -5 /tmp/f1 > /tmp/f2
rm /tmp/f1
head /tmp/f2 > /tmp/f3
rm /tmp/f2
more /tmp/f3
rm /tmp/f3
```

---

```
#!/usr/bin/ksh
# Uses variables to store data from
commands
```

```
SYS=`hostname`
ME=`whoami`
W="on the system"
echo "I am $ME $W $SYS"
```