Shell Reference Card 1.0

Shell commands may be invoked via an interactive terminal, or batched from within a file (called a shell script).

There are 2 methods for invoking shell commands from a file; the first is to pass the filename containing the shell commands to the shell interpreter:

```
sh <file
```

the second, (more convenient) is to give the file containing the commands execute permissions (`chmod a+x file`) and indicating the interpreter on the the first line of the file:

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

Comment Lines

A word beginning with `#` causes that word and all the following characters up to a newline to be ignored.

```
# This is a comment
```

Command Substitution

Commands my be executed, and the standard output stored in a variable by enclosing the command between two grave accents (back-quotes ‘ ‘).

This example executes `grep` and stores the resulting output in the variable `OUTPUT`:

```
OUTPUT=‘grep Tom /usr/dict/words‘
```

Parameter Substitution (Variables)

The character $ is used to denote variables. There are two types of variables: positional and keyword. If a digit follows the $ character, the variable is a positional parameter (command-line arguments) which may be assigned values by `set`. Variables may be assigned values using:

name=*value* [*name*=*value*] ...

Some examples:

```
MONTH="January" YEAR="2000"
MESSAGE="Hello World."
```

Input/Output Redirection

Three file descriptors (0, 1 and 2) are automatically opened when a shell in invoked. They represent:

0	standard input (stdin)
1	standard output (stdout)
2	standard error (stderr)

A command's input and output may be redirected using the following notation:

`<file`	take input from *file*
`>file`	write output to *file* (truncate to zero if it exists)
`>>file`	append output to *file* else create
`<<word`	"here" document; read input until line matches *word*
`<>file`	open *file* for reading and writing
`<&digit`	use file descriptor *digit* as input (`>&digit` for output)
`<&-`	close standard input (`>&-` close output)
`cmd1|cmd2`	stdout of *cmd1* is piped to stdin of *cmd2*

If any of the above redirect operators are preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1).

The first example saves the output of `ls -l` to a file called `listing`; the second example pipes the output of `ls -l` directly to the printer:

```
ls -l >listing
ls -l | lpr
```

This example redirects stdout (file descriptor 1) of `truss` to a file called `out` and stderr (file descriptor 2, error messages) to a file called `err`:

```
truss ls >out 2>err
```

This example redirects both stdout and stderr to a file called `both` by first redirecting stdout to the file and then stderr to stdout (which goes to the file):

```
truss ls 1>both 2>&1
```

In this example, `lpr` reads from stdin (file descriptor 0):

```
lpr <&0
```

This example pipes the output of `zcat` (uncompresses a file to stdout) into `tar` and requests the contents of the tar archive:

```
zcat file.tar.Z | tar tvf -
```

Special Variables

Braces are required (${parameter}) only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is `*` or `@`, all the positional parameters, starting with `$1`, are substituted (separated by spaces). Parameter `$0` is set from argument zero, the program name, when the shell is invoked.

```
PROG=$0
ALL=$*
echo "arg0=$PROG arg1=$1 arg2=$2 all=$ALL"
```

The following variables are automatically set by the shell:

`$#`	the number of positional parameters
`$-`	flags supplied to the shell
`$?`	value returned by the last executed command
`$$`	process id of this shell
`$!`	process id of the last background command

Some general conventions of shell programming include using `$$` as a name for temporary files and `$#` in **case** statements (see below) to validate command line arguments.

case

case *word* **in** [*pattern* [| *pattern*]) *list* **;;**] ... **esac**

This example validates the number of command-line arguments; if no arguments are provided it displays the usage to stderr, otherwise it falls through and the script continues executing.

```
USAGE="Usage: $0 userid"
case $# in
    0)  echo $USAGE >&2
        exit 1;;
    *)  ;;
esac
```

This example sets the variable `NUM` to a number from 1 to 12 depending on the contents of `MONTH`. If `MONTH` is a number rather than an abbreviation it simply uses that number.

```
case $MONTH in
    Jan|jan) NUM=1;;
    Feb|feb) NUM=2;;
    :
    :
    Dec|dec) NUM=12;;
    1|2|3|4|5|6|7|8|9|10|11|12) NUM=$MONTH;;
esac
```

for

for *name* [**in** *word* . . .] **do** *list* **done**

This example prints: `1`, `2`, `3`, `a`, `b` and `c`:

```
for i in 1 2 3 a b c
do
    echo $i
done
```

This example copies every file with a `.c` extension to a file with a `.bak` extension, effectively making backup copies:

```
for i in ‘ls *.c‘
do
    cp $i $i.bak
done
```

while

while *list* **do** *list* **done**

This example monitors the size of a growing file by continuously performing an `ls` every 5 seconds (the command ":" evaluates as true):

```
while :
do
    ls -l download.tgz
    sleep 5
done
```

getopts

`getopts` is used to parse command-line options. Option names can be single character only. Options with arguments are indicated with a trailing ":".

In the following example, `-B` does not have an option while `-K` (number of copies), and `-P` (printer name) do:

```
while getopts BK:P: options
do
  case $options in
      K) COPIES=$OPTARG;;
      P) PRINTER=$OPTARG;;
      B) BANNERFLAG=1;;
      ?) echo "lpr [-B -K copies -P printer] file"
         exit 2;;
  esac
done
shift ‘expr $OPTIND - 1‘
```

expr

`expr` is used to perform integer math in shell scripts. Standard mathematical symbols behave as you would expect them to.

In this example, the **while** loop counts up to 10:

```
a=0
while [ "$a" -lt 10 ]
do
   a=‘expr $a + 1‘
   echo $a
done
```

Miscellaneous

`basename`, `cut` are useful commands in shell scripting.

`http://www.ee.ryerson.ca/~elf/sh/`

if

if *condition* ; **then** *action* ; [**elif** *condition2* ; **then** *action2* ;] [**else** *action3* ;] **fi**

If *condition* evaluates as true (i.e. returns a 0 exit status) then *action* is executed. Otherwise *condition2* is evaluated and if it returns 0 then *action2* is executed. If neither *action* nor *action2* are executed, then *action3* is executed.

The following primitives are used to construct *condition*:

`-r file`	True if *file* exists and is readable
`-w file`	True if *file* exists and is writeable
`-x file`	True if *file* exists and is executable
`-f file`	True if *file* exists and is a regular file
`-d file`	True if *file* exists and is a directory
`-h file`	True if *file* exists and is a symlink
`-c file`	True if *file* exists and is a character special file
`-b file`	True if *file* exists and is a block special file
`-p file`	True if *file* exists and is a named pipe (fifo)
`-u file`	True if *file* exists and is setUID
`-g file`	True if *file* exists and is setGID
`-k file`	True if *file* exists and sticky bit is set
`-s file`	True if *file* exists and its size is greater than 0
`-t fd`	True if open file desc. *fd* is assoc. with a terminal
`-z str`	True if the length of *str* is zero
`-n str`	True if the length of *str* is non-zero
`s1 = s2`	True if strings *s1* and *s2* are identical
`s1 != s2`	True if strings *s1* and *s2* are different
`s1`	True *s1* is not the null string (empty)
`n1 -eq n2`	True if integers *n1* and *n2* are equal
`n1 -ne n2`	True if integers *n1* and *n2* are not equal
`n1 -gt n2`	True if integer *n1* is greater than *n2*
`n1 -ge n2`	True if integer *n1* is greater than or equal to *n2*
`n1 -lt n2`	True if integer *n1* is less than *n2*
`n1 -le n2`	True if integer *n1* is less or equal to *n2*
`-L link`	True if the file pointed by *link* exists

Primitives may be combined using the following operators:

`!`	Unary negation
`-a`	Binary *and*
`-o`	Binary *or* (lower precedence than `-a`)
(*condition*)	Grouping parentheses (must be quoted)

This example stores the output of `grep` into `ENTRY` then checks if it contains any output (`if [ -z` . . .) and exits if it doesn't:

```
ENTRY=‘/usr/5bin/grep -s -i "$*" $DATABASE‘
if [ -z "$ENTRY" ]; then
    exit 2
fi
```

This example `ghostviews` the resulting `.ps` file if both the preceeding `dvips` and the `latex` commands were successful:

```
latex file.tex && dvips file.dvi && gv file.ps
```

This example tests to see if either of two directories (passed as command-line arguments) exist:

```
test -d "$1" || test -d "$2"
```