Reading Assignment: Chapter 1, 2, 3, 4, and 8 from the book by Sobell; also read Chapter 5 from the same book though I'll cover parts of it in class.

"UNIX was not designed to stop you from doing stupid things, because that would also stop you from doing clever things."

– Doug Gwyn

## Shellscript Programming

### Unix shell

- Program that interacts with the user to receive requests for running programs and executing them
- Can be used as an interpreted programming language
  - The programs written in shell command language do not have to be compiled (like the C programs) but can be executed directly
- Useful features of the shell
  - Metacharacters
  - Quoting
  - Creating new commands
  - Passing arguments to the commands
  - Variables
  - Control flow

### Command line structure

- Each command is a single word that names a file to be executed
- Example

```
$ who
```

- Command ends with a newline or a semicolon

```
$ date ; who
```

- Sending the output of the above through a pipe

```
$ date ; who | wc
```

- Pipe takes precedence over the semicolon
  - Semicolon separates two commands
  - Pipe connects two commands into one
- The precedence can be adjusted by using the parenthesis

```
$ ( date ; who ) | wc
```

- Data flowing through a pipe can be intercepted by the command `tee` to be saved in a file

  ```
  $ ( date ; who ) | tee save | wc
  ```

  `tee` copies the input to the named file, as well as to the output

- Commands can also be terminated by an ampersand (`&`)
  - Useful to run long running commands in the background

    ```
    $ ( sleep 5 ; date ) & date
    ```

- Precedence rules (again)
  - `&` has a lower priority compared to both `|` and `;`

- The special characters interpreted by the shell (`<`, `>`, `|`, `;`, and `&`) are not arguments to the programs being run, but control the running of those programs

- The command

  ```
  $ echo Hello > junk
  ```

  can also be written as

  ```
  $ > junk echo Hello
  ```

- Avoiding overwriting files by accident
  - In C shell, you can type the command

    ```
    % set noclobber
    ```

    to avoid overwriting a file by accident during output redirection
  - If you have already set noclobber and want to overwrite a file still, you have to follow the redirection symbol `>` with a bang (`!`) as

    ```
    % echo hello >! junk
    ```

**Metacharacters**

- Characters that have special meaning to the shell
- Most notable metacharacter is the asterisk or `*`
- An asterisk, by itself, matches all the files in the current working directory

  ```
  $ echo *
  ```

- Metacharacters can be protected from interpretation by the shell by using quotes

  ```
  $ echo '***'
  ```

- You can also use double quotes but shell peeks inside the double quotes to look for `$`, ` ... ` (back quotes), and `\`
- You can also protect metacharacters by preceding them with a backslash or escape character

  ```
  $ echo \*\*\*
  ```

Table 1: Other metacharacters

| | |
|---|---|
| `>` | redirect stdout to a file |
| `>>` | append stdout to a file |
| `<` | take stdin from a file |
| `\|` | pipeline |
| `<<str` | stdin follows, upto next `str` on a line by itself |
| `*` | match any string of zero or more characters |
| `?` | match any single character in the filenames |
| `[ccc]` | match any single character from `ccc` in the filenames |
| | ranges such as `[0-9]` and `[a-z]` are legal |
| `;` | terminate command |
| `&` | terminate command and run it in the background |
| `` `...` `` | run command(s) in `...` |
| `(...)` | run command(s) in `...` as a subshell |
| `{...}` | run command(s) in `...` in current shell |
| `$1`, `$2` | `$0 ... $9` replaced by arguments to shell file |
| `$var` | value of shell variable `var` |
| `${var}` | value of shell variable to be concatenated withe text |
| `\c` | take character `c` literally (suppress newline in `echo`) |
| `'...'` | take `...` literally |
| `"..."` | take `...` literally, but after interpreting `$`, `` `...` ``, and `\` |
| `#` | beginning of comment (ends with the end of line) |
| `var=value` | assignment (no spaces around operator) |
| $p_1$ `&&` $p_2$ | run $p_1$; if successful, run $p_2$ |
| $p_1$ `\|\|` $p_2$ | run $p_1$; if unsuccessful, run $p_2$ |

  - A backslash is the same as using the single quotes around a single character
  - The backslash character can be used to quote itself as `\\`

    ```
    $ echo abc\\def
    abc\def
    $
    ```

  - A backslash at the end of the line causes the line to be continued (as if the newline character is not there)

- Quoted strings can contain newlines

  ```
  $ echo 'hello
  > world'
  ```

- The > character above is secondary prompt stored in variable `PS2` and can be modified to preference
- The metacharacter `#` starts a comment only if it is placed at the beginning of a word (following whitespace)
- The newline following the `echo` command can be suppressed with the `-n` option

**Bourne shell**

- Developed by and named after Stephen R. Bourne at Bell Labs
- If a shell name is not specified, it is assumed that the reference is to Bourne shell
- Commonly known as `sh` or `/bin/sh`

**Shellscripts**

- Ordinary text files that contain commands to be executed by the shell
- First line of the script should identify the shell (interpreter) used by the script for execution
- For Bourne shell, the identifying line is

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

- If the first line is other than the identifying line, the script defaults to Bourne shell

**Creating shellscripts**

- Creating the script `nu` to count the number of users

```
$ echo 'who | wc -l' > nu
```

- The above script can be executed by

```
$ sh nu
```

- You can also change the permissions on the script to avoid typing `sh`

```
$ chmod +x nu
$ nu
```

- Child shell or subshell
- Putting the scripts in a separate directory

```
$ mkdir $HOME/scripts
$ PATH=$PATH:$HOME/scripts
$ mv nu scripts
$ nu
```

**Command arguments and parameters**

- Items are specified within the script by argument numbers as `$1` through `$9`
- The number of arguments itself is given by `$#`
- Writing a script to change mode to executable for a specified file
- Shellscript `cx` as a shorthand for `chmod a+x`

```
chmod +x $1
```

- What if there are multiple files (like more than 10)
  - The line in the shellscript can handle eight more arguments as

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

  - More arguments can be handled with

```
chmod +x $*
```

  - `$*` indicates operation on all the arguments

- Creating a phonebook
  - Let us have a personal telephone directory as

| | |
|---|---|
| dial-a-joke | 212-976-3838 |
| dial-a-prayer | 212-246-4200 |
| dial santa | 212-976-3636 |
| dow jones report | 212-976-4141 |

  - It can be searched by using the `grep` command
  - Let us call our shell script as `411`

```
grep $* phone-book
```

  - Problems if you type a two word pattern
  - Can be fixed by enclosing the `$*` in double quotes
  - `grep` itself can be made case insensitive by using the `-i` option
- The argument `$0` is the name of the program being executed

**Program output as argument**

- The output of any program can be included in the script by using the backward quotes

```
echo The current time is ‘date‘
```

- The backwards quotes are also interpreted within the double quotes but not in single quotes

**Difference between `sh nu` and `sh < nu`**[1]

- `sh nu` maintains the stdin of its invoking environment, while `sh < nu` forces stdin to come from the script, thus making non-redirected "read"s inside the script to behave oddly (and also affecting any program invoked by `nu` which read stdin)
- `$0` in the former is set to `nu`; in the latter it is `sh`

**Shell variables**

- `ksh` can work with numeric as well as string variables but Bourne shell is limited to string variables alone
- Variables in a shell are known as *parameters*
  - Strings like `$1` are called *positional parameters*
  - Positional parameters hold the arguments to a shell script file
- Other shell variables include `PATH` and `HOME`
  - `PATH` contains a colon-separated list of directories that are searched by the shell whenever a command is typed
- The value of a variable can be seen by preceding its name with a `$`
- The positional parameters cannot be changed as `$1` is just a compact notation to get the value of the first parameter

[1] Quoted from Ken Pazzini (`ken@halcyon.com`)

- Shell variables can be created, accessed, and modified

```
$ echo $PATH
$ PATH=$PATH:/usr/games
$ echo $PATH
```

- New variables (non-shell variables) can be created by assigning them values using the following syntax

```
color1=red
color2="Mulberry Red"
color3='Seattle Silver'
dir=`pwd`
cd /usr/lib
pwd
cd $dir
pwd
```

  - If there is a space in the string that is being assigned to the variable, enclose it in double quotes or single quotes as shown in examples above
  - Spacing on either side of the = operator is important; you can not have any

- The value in the variables can be accessed by preceding the variable name with the character $ as follows

```
$ echo $color1
red
$
```

- Shell variables are usually in upper case and non-shell variables can be defined in lower case to distinguish between the two
- The command `set` displays the values of all the defined variables (including the ones defined by the user)
- The value of a variable is associated with the shell that creates it and is not automatically passed to the new shell (or child)
  - A shell will keep its variables intact even if they are modified in a subshell (or child)
- A shellscript can be run quickly by using the `.` command
- If the value of a variable is to be used in subshells, you should use the `export` command

```
$ PATH=$PATH:/usr/local/bin
$ export PATH
```

**More on I/O redirection**

- The terminal as a file
  - Unix treats every device as a file
    - In effect, terminal and keyboard can be treated as files
    - Keyboard is an input file
    - Terminal monitor is an output file
  - The devices reside in the directory `/dev`
  - You can find the file associated with your terminal in the `/dev` directory by using the command `tty`

Table 2: Shell built-in variables

| | |
|---|---|
| `$#` | number of arguments |
| `$*` | all arguments to shellscript |
| `$@` | similar to `$*` |
| `$-` | options supplied to the shell |
| `$?` | return value of the last command executed |
| `$$` | pid of the shell |
| `$!` | pid of the last command started with `&` |
| `$HOME` | home directory of the user |
| `$IFS` | list of characters that separate words in arguments |
| `$MAIL` | system mail folder to keep incoming mail |
| `$PATH` | list of directories to search for commands |
| `$PS1` | prompt string, default value '`$ `' |
| `$PS2` | prompt string for continued command line, default value '`> `' |
| `EDITOR` | Name of your favorite editor |

- Every program has three default files – `stdin`, `stdout`, and `stderr`
- These files are respectively numbered as 0, 1, and 2
- Redirection
  - Sending the output of one program to a file, instead of `stdout` or `stderr`
  - Receiving the input to a program from a file instead of `stdin`
  - Based on operators <, >, <<, >> as well as pipes and `tee`
- Example – Command to determine the execution time

  ```
  $ time command
  ```

  The output of the time command is sent to `stderr`
- To capture the output in a file, use the following:

  ```
  $ time command > command.out 2> time.out
  ```

- The two output streams can be merged by any of the following two commands

  ```
  time command > command.out 2>&1
  time command > command.out 1>&2
  ```

- New directory program

  ```
  grep "$*" <<END
  dial-a-joke 212-976-3838
  dial-a-prayer 212-246-4200
  dial santa 212-976-3636
  dow jones report 212-976-4141
  END
  ```

**The exit statement/status**

Table 3: Shell I/O redirection

| | |
|---|---|
| `> file` | direct stdout to `file` |
| `>> file` | append stdout to `file` |
| `< file` | take stdin from `file` |
| `p1 \| p2` | connect `stdout` of `p1` to `stdin` of `p2` |
| `n> file` | direct output from file descriptor `n` to `file` |
| `n>> file` | append output from file descriptor `n` to `file` |
| `n>&m` | merge output from file descriptor `n` with file descriptor `m` |
| `n<&m` | merge input from file descriptor `n` with file descriptor `m` |
| `<<s` | here document; take `stdin` until next `s` at beginning of a line; substitute for `$`, `` `...` ``, and `\` |
| `<<\s` | here document with no substitution |
| `<<'s'` | here document with no substitution |

- Every Unix command runs `exit` upon termination
- As opposed to the standard C notation, in shellscripts, a zero signifies `true` and any positive integer represents `false`
- Why should we have a positive integer as `false` in shellscripts
  - The commands can fail for several reasons
  - The reason for failure can be encoded in the exit status
  - As an example, the exit status for `grep` command returns
    - 0 if there was a match
    - 1 if there was no match
    - 2 if there was an error in the pattern or filename
- The exit status for the last command is stored in the variable `$?`
- When writing your C programs, you return the exit status through `exit` system call (1 upon error)
- Using `cmp` command to compare two files and taking an action based on the exit status
- When commands are grouped, the exit status of the group is the exit status of the last command executed

**Command separators**

- Command is usually terminated by the end of line
- Other command separators are
  - Semicolon (`;`)
    - Separates commands for sequential execution
    - Command following the semicolon will not begin execution until the command preceding the semicolon has completed
    - Semicolon is equivalent to a newline but allows more than one command to be on the same line
  - Pipe (`|`)
    - Causes the standard output of the command before the vertical bar to be connected, or *piped*, into the standard input of the command following the vertical bar

  - The sequence of commands is called a *pipeline*
  - Each command in the pipeline is executed as a separate process and the commands are executed concurrently
  - The exit status of the pipeline is the exit status of the last command in the pipeline
  - The pipeline

    ```
    cmd1 | cmd2
    ```

    is equivalent to

    ```
    cmd1 > tmp; cmd2 < tmp
    ```

- Background execution operator (`&`)
  - Causes the command preceding it to be executed in the background
  - The process created to execute the command executes independent of the current process and the current process does not have to wait for it to complete
  - Ampersand is not considered to be a command separator
  - The standard input of the command run in the background is connected to `/dev/null` to prevent the current process and the background process from trying to read the same standard input
- ORed execution (`||`)
  - `cmd1 || cmd2` causes `cmd2` to be executed only if `cmd1` fails
  - `||` can be used to write conditional statements that resemble the C programming language as

    ```
    if cmd1 || cmd2
    then
        ...
    fi
    ```
- ANDed execution (`&&`)
  - `cmd1 && cmd2` causes `cmd2` to be executed only if `cmd1` executes successfully

**Command grouping**

- Grouping with parentheses
  - Grouping with parentheses makes the commands to execute in a separate shell process, or subshell
  - The shell creates the subshell to execute the commands in parentheses and waits for the subshell to complete before resuming
  - Subshell inherits the environment of the parent process but is not able to alter it
  - Useful when you do not want the environment of the shell to be altered as a result of some commands, for example, in `Makefile`, you may have

    ```
    ( cd $SUBDIR1; make )
    ```
  - An ampersand can be used after the right parenthesis to execute the subshell in the background
  - Parentheses can be nested to create more than one subshell
- Grouping with braces
  - Same as grouping with parentheses but the commands are executed in the current shell
  - The syntax is:

    ```
    { cmd1; cmd2; ...; }
    ```
  - The last command *must* be followed by a semicolon and there *must* be a space between commands and braces
  - Useful to redirect the combined standard input or standard output of the commands within the braces

        * In such a case, commands are executed in a subshell and braces have the same behavior as parentheses
    - Braces can be nested and may be followed by an ampersand to execute the commands within the braces in the background

**Looping in a shell program**

- The syntax for loops is

```
for x [in list]
do
    ...
done
```

    - The loop executes once for each value in `list`
        * `list` is a string that is parsed into words using the characters specified in the `IFS` (internal field separators) variable as delimiters
            · Initially, `IFS` variable is set to space, tab, and newline (the whitespace characters)
        * The part `[in list]` is optional
        * If `[in list]` is omitted, it is substituted by positional parameters
    - The value of `x` is accessed by `$x`

- Example – To display the number of lines in each file

```
for file in *.tex
do
    echo -n "$file "
    wc -l $file | awk '{printf "\t"$1"\n"}'
done
```

- You can use the `break` command to exit the loop early or the `continue` command to go to the next iteration by skipping the remainder of the loop body
- The output of the entire `for` command can be piped to another program, as the entire command is treated as a single entity

**The case statement**

- The syntax for the `case` statement is

```
case value in
    pattern1)    commands1 ;;
    pattern2)    commands2 ;;
    .
    .
    .
    *) commands for default ;;
esac
```

    - Patterns can use file generation metacharacters
    - Multiple patterns can be specified on the same line by separating them with the | symbol (not to be confused with the use of the same symbol for communicating between programs)

- A modified calendar program

```
#!/bin/sh
# newcal :  Nice interface to /bin/cal

case $# in
0) set `date`; m=$2; y=$6 ;; # no arguments; use today
1) y=$1; set `date`; m=$2 ;; # 1 argument; use this year
2) m=$1; y=$2 ;; # 2 arguments; month and year
*) echo "Too many arguments to $0" ;
   echo "Aborting ..." ;
   exit 1 ;;
esac

case $m in
[jJ]an* ) m=1 ;;
[fF]eb* ) m=2 ;;
[mM]ar* ) m=3 ;;
[aA]pr* ) m=4 ;;
[mM]ay* ) m=5 ;;
[jJ]un* ) m=6 ;;
[jJ]ul* ) m=7 ;;
[aA]ug* ) m=8 ;;
[sS]ep* ) m=9 ;;
[oO]ct* ) m=10 ;;
[nN]ov* ) m=11 ;;
[dD]ec* ) m=12 ;;
[1-9] | 0[1-9] | 1[0-2] ) ;; # numeric month
*)
esac

/bin/cal $m $y

exit
```

### The set statement

- A shell built-in command
- Can be used to assign values to variables
- Without arguments, `set` shows the values of variables in the environment
- Ordinary arguments reset the values of the variables `$1`, `$2`, and so on
  - Consider `set` `` `date` `` as used in the above shellscript
  - `$1` is set to the day of the week
  - `$2` is set to the name of the month
  - and so on
- You can also specify some options with `set`, such as `-v` and `-x` for echoing the commands to assist in debugging

### The if statement

- The `if` statement runs commands based on the exit status of a command
- The syntax is

```
if condition
then
    commands if the condition is true
else
    commands if the condition is false
fi
```

- The `else` part is optional
- Every `if` statement is terminated by an `fi` statement
- The condition is followed by the keyword `then` on a line by itself
  - `then` can be on the same line as the condition if it is preceded by a semicolon
- Example

```
if [ $counter -lt 10 ]
then
    number=0$counter
else
    number=$counter
fi
```

- Nesting of `if` with `else` using `elif`
  - You can use any number of `elif` statements in an `if` statement to check for additional conditions
  - Each `elif` statement contains a separate command list
  - The last `elif` statement can be followed by an `else` statement

```
if [ $counter -lt 10 ]
then
    number=00$counter
elif [ $counter -lt 100 ]
    then
        number=0$counter
else
    number=$counter
fi
```

**while and until loops**

- The syntax for `while` loop is

```
while condition
do
    commands
done
```

- Looking for someone to log in once every minute

```
while sleep 60
do
    if who | grep -s $1
    then
        echo "$1 is logged in now"
    fi
done
```

- You can also use the null statement (`:`) for an infinite loop; with null statement, the above example – with a `break` statement to terminate infinite loop – can be written as

```
while :
do
    sleep 60
    if who | grep -s $1 ; then
        echo "$1 is logged in now"
        break
    fi
done
```

  - The null command does nothing and always returns a successful exit status

- The syntax for `until` loop is

```
until condition
do
    commands
done
```

- Same example as above

```
until who | grep -s $1
do
    sleep 60
done

if [ $? ]
then
    echo "$1 is logged in now"
fi
```

**Things to watch for in shellscript writing**

- Specifying the `PATH` variable
  - Notice how specification of path alters the behavior of the script

```
#!/bin/ksh

PATH=/usr/ucb:/bin

for file in *.tex
do
    echo -n "$file"
    wc -l $file | awk -F '{printf "\t"$1"\n"}'
```

```
done

PATH=/bin:/usr/ucb

for file in *.tex
do
    echo -n "$file"
    wc -l $file | awk -F '{printf "\t"$1"\n"}'
done
```

- Specifying the usage of each command
- Using the exit status values

### Evaluation of shell variables

- Different symbols in the definition of shell variables and their meaning

Table 4: Shell variable evaluation

| | |
|---|---|
| `$var` | value of `var`<br>nothing, if `var` undefined |
| `${var}` | same; useful if alphanumerics follow variable name |
| `${var-value}` | value of `var` if defined; otherwise `value`<br>`$var` unchanged |
| `${var=value}` | value of `var` if defined; otherwise `value`<br>if undefined, `$var` set to `value` |
| `${var?message}` | value of `var` if defined; otherwise print `message` and exit shell<br>if `message` is not supplied, print the phrase<br>`parameter null or not set` |
| `${var+value}` | `value` if `$var` defined, otherwise nothing |

### Filters

- Family of programs that operate on some input (preferably `stdin`) and produce some output (preferably `stdout`)
- Exemplified by `grep`, `tail`, `head`, `sort`, `tr`, `uniq`, `wc`, `sed`, and `awk`
- `sed` and `awk` are derived from `grep` and are also known as *programmable filters*
  - `grep` uses a regular expression for the pattern to be matched
  - The regular expression is the same as used by `ed` – the underlying editor for `vi` and `ex`
  - `sed` and `awk` generalize the pattern as well as the action

### Handling interrupts

- One of the most important things in a shellscript is to properly handle interrupts
- You should delete any temporary files when the user interrupts the script
- In Korn/Bourne shell, the syntax for interrupt handling is

```
trap commands signal
```

- The commonly used signals for shellscripts are 2 (for `^C`), 14 (for alarm timeout), and 15 (for software termination via `kill`)
- If `$TMP` contains the name of a temporary file, you should remove it if the user hits `^C` or `kill` by

```
trap "rm -f $TMP ; exit 1" 2 15
```

**Parsing options**

- In Unix, the options are specified with a command line switch – followed by the option identifier
- The parameters are generally specified after the options have been entered on the command line
- A typical example of a Unix command is

```
ls [-altr] [filename]
```

where everything within the square brackets is optional

- The parsing of options should work in a way that the options and arguments, if any, are captured and then, the parameter list is captured
- This is achieved in Bourne shell by using the command `getopts` and its associated variables `OPTIND` and `OPTARG`
- The `getopts` command succeeds if there is an option left to be parsed and fails otherwise
- The `OPTIND` variable is set to the command line position of the next option as the options are parsed by `getopts`
- The `OPTARG` returns the expected option value from `getopts`
- Example

```
#!/bin/sh

while getopts abcdf:h OPTNAME
do
    case $OPTNAME in
    a)  echo Option a received
        ;;
    d)  set -x
        ;;
    f)  echo Option f received
        echo Optional argument is: $OPTARG
        ;;
    h | \?)  echo usage: $0 [-abc] [-f filename] parameter1 parameter2 ...
        echo Options are:
        echo "    -d  :  Turn debugging on"
        echo "    -h  :  Print help (this message)"
        exit 1
        ;;
    esac
done

shift `expr $OPTIND - 1`

echo The number of parameters is: $#
echo The parameters are: $*
```

**Filename generation**

- Shell can generate filenames to match the names of existing files by using metacharacters
- Filename generation metacharacters are also called wild cards
- The `?` metacharacter
  - Matches one and only one character in the name of an existing file
  - `READ?ME` will match `READ_ME`, `READ.ME`, and `READ-ME` but not `README`
- The `*` metacharacter
  - Matches any number of characters, including zero, in a filename
  - `READ*` will match `READ_ME`, `READ.ME`, `READ-ME`, and `README` but not `TOREAD`
  - You can combine the `?` and `*` to list all the hidden files by the expression `.??*`