Conditional statements

Summary of conditional statements

- `if condition; then c1; c2; ... cm; fi`
- `if condition; then c1; c2; ... cm; else d1; d2; ... dn; fi`
- Each semi-colon (`;`) can be substituted by a new line.
- ```
  if condition
  then
      c1
      c2
      ...
      cm
  else
      d1
      d2
      ...
      dn
  fi
  ```
  ```

Summary of conditional statements (continued)

- **`if condition1; then block_1; elif condition2; then block_2; ... else block_r; fi`**
- **`case value in val1) block1 ;; val2) block2 ;; ... valn) blockn ;; *) dftblock ;; esac`**
- Note the use of semi-colons.
  - Every condition must be followed by a semi-colon.
  - No need to have a semi-colon after **`then`**, **`else`**, or **`elif`**.
  - Use a semi-colon before **`then`**, **`else`**, or **`elif`** if it appears in the same line as the preceding block.
  - A double-semi-colon is needed before every **`case`** option (starting from the second).
- Multiple values in **`case`** can be separated by **`|`**.
- For checking the conditions, bash silently looks at the special variable **`$?`**, and proceeds accordingly.

Example of conditional statements

checkfile.sh

```
#!/bin/bash

if [ $# -eq 0 ]; then
   echo "Run with one command-line argument"
   exit 1
fi
fname=$1
if [ ! -e "$fname" ]; then
   echo "\"$fname\" does not exist"
   exit 2
else
   if [ -f "$fname" ]; then echo "\"$fname\" is a regular file"
   elif [ -d "$fname" ]; then echo "\"$fname\" is a directory"
   else echo "\"$fname\" is neither a regular file nor a directory"
   fi
   echo -n "Permissions:"
   if [ -r "$fname" ]; then echo -n " read"; fi
   if [ -w "$fname" ]; then echo -n " write"; fi
   if [ -x "$fname" ]; then echo -n " execute"; fi
   echo ""
fi
```

Run checkfile.sh

```
$ ./checkfile.sh
Run with one command-line argument
$ ./checkfile.sh checkfile.sh
"checkfile.sh" is a regular file
Permissions: read write execute
$ ./checkfile.sh /usr/
"/usr/" is a directory
Permissions: read execute
$ ./checkfile.sh ~/spl/
"/home/foobar/spl/" is a directory
Permissions: read write execute
$ ./checkfile.sh /dev/null
"/dev/null" is neither a regular file nor a directory
Permissions: read write
$
```

Example of case

```
#!/bin/bash
if [ $# -eq 0 ]; then
   echo "Usage: $0 FILENAME WORD1 [WORD2 [WORD3]]"
   exit 1
fi
fname=$1
case $# in
   1) echo "You should specify one, two, or three word(s)" ;;
   2) c=`grep -c -e $2 $fname` ;;
   3) c=`grep -c -e $2 -e $3 $fname` ;;
   4) c=`grep -c -e $2 -e $3 -e $4 $fname` ;;
   *) echo "Too many words. Giving up..." ;;
esac
if [ $c ]; then echo "$c lines matched"; fi
```

```
$ ./options.sh
Usage: ./options.sh FILENAME WORD1 [WORD2 [WORD3]]
$ ./options.sh textfile.txt
You should specify one, two, or three word(s)
$ ./options.sh textfile.txt problem
2 lines matched
$ ./options.sh textfile.txt problem algorithm
7 lines matched
$ ./options.sh textfile.txt problem algorithm method
12 lines matched
$ ./options.sh textfile.txt problem algorithm method protocol
Too many words. Giving up...
$
```

An inefficient Fibonacci-number calculator

fib.sh returns Fibonacci numbers by echoing

```
#!/bin/bash

function FIB () {
   local n=$1
   if [ $n -le 1 ]; then echo "$n"; return 0; fi
   echo $(( `FIB $((n-1))` + `FIB $((n-2))` ))
}

echo -n "Enter n: "; read n
echo "F($n) = `FIB $n`"
```

```
$ ./fib.sh
Enter n: 0
F(0) = 0
$ ./fib.sh
Enter n: 1
F(1) = 1
$ ./fib.sh
Enter n: 5
F(5) = 5
$ ./fib.sh
Enter n: 10
F(10) = 55
$ ./fib.sh
Enter n: 16
F(16) = 987
$
```

Calculating Fibonacci numbers with memoization: A failed attempt

fibmemobad.sh attempts to store the calculated Fibonacci numbers in the array F[ ]

```
#!/bin/bash
function FIB () {
   local n=$1
   if [ ! ${F[$n]} ]; then
      F[$n]=$(( `FIB $((n-1))` + `FIB $((n-2))` ))
   fi
   echo ${F[$n]}
}
echo -n "Enter n: "; read n
declare -ai F=([0]=0 [1]=1)
echo "${F[@]}"
echo "${!F[@]}"
echo "F($n) = `FIB $n`"
echo "${F[@]}"
echo "${!F[@]}"
```

Output is correct, but equally slow

```
$ ./fibmemobad.sh
Enter n: 16
0 1
0 1
F(16) = 987
0 1
0 1
$
```

A repaired script that actually does memoization

fibmemo.sh makes all the calculations in the current shell, so F[ ] is correctly modified

```
#!/bin/bash
function FIB () {
   local n=$1
   if [ ! ${F[$n]} ]; then
      FIB $((n-1))
      FIB $((n-2))
      F[$n]=$(( F[n-1] + F[n-2] ))
   fi
}
echo -n "Enter n: "; read n
declare -ai F=([0]=0 [1]=1)
echo "${F[@]}"
echo "${!F[@]}"
FIB $n
echo "F($n) = ${F[$n]}"
echo "${F[@]}"
echo "${!F[@]}"
```

```
./fibmemo.sh
Enter n: 16
0 1
0 1
F(16) = 987
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
$
```

A script to reverse strings

reversal.sh

```
#!/bin/bash

function reverse () {
   local S=$1
   local Slen=${#S}
   local T
   case $Slen in
      0|1) echo "$S" ;;
      *) T=${S:0:-1}; T=`reverse "$T"`; echo "${S: -1}$T" ;;
   esac
}

echo -n "Enter a string:  "; read S
echo -n "reverse($S) = "
S=`reverse "$S"`
echo "$S"
```

```
$ ./reversal.sh
Enter a string: a bc def ghij klmno pqrstu
reverse(a bc def ghij klmno pqrstu) = utsrqp onmlk jihg fed cb a
$
```

Loops

The loop structures at a glance

- **`for item in list; do block; done`**
- **`for arg do block; done`**

  This is equivalent to

  `for arg in $@; do block; done`
- **`while condition; do block; done`**

  While loops are repeated so long as **`condition`** is true.
- **`until condition; do block; done`**

  Until loops are repeated so long as **`condition`** is false.
- Semi-colons may be replaced by new lines.

```
for item in list
do
   block
done
```

- **`break`**, **`continue`**, and **`exit`** work in the usual way.

A simple example

- Each of the following loops prints $0, 1, 2, \ldots, 9$ in that order, one in each line.
  - `for n in 0 1 2 3 4 5 6 7 8 9; do echo $n; done`
  - `for n in ` echo {0..9}` ; do echo $n; done`
  - `n=0; while [ $n -lt 10 ]; do echo $n; n=$((n+1)); done`
  - `n=0; until [ $n -eq 10 ]; do echo $n; n=$((n+1)); done`
- Do not quote the list in the **`for`** loop, because the quoted list stands for a list of a single item.
- The range **`{i..j}`** can be specified only for constant values of **`i`** and **`j`**.

An iterative Fibonacci number calculator

- The shell script **`fibiter.sh`** maintains an array F to store $F(0), F(1), F(2), \ldots, F(N)$ for some N.
- N is initialized to 1, and F is initialized to store $0, 1$ only.
- In a loop, the script asks for entering n.
- The loop continues until the user enters a non-negative value for n.
- If $n \leqslant N$, $F(n)$ is read from the array, and displayed.
- If $n > N$, then the array F is appended by $F(N+1), F(N+2), \ldots, F(n)$, and N is set to n. Subsequently, $F(n)$ is displayed.
- After this, the user is asked whether (s)he wants to continue. If not, the script terminates.

The script fibiter.sh

```
#!/bin/bash

function computerest () {
   local n=$1
   while [ $n -le $2 ]; do F[$n]=$((F[n-1]+F[n-2])); n=$((n+1)); done
}

declare -ia F=([0]=0 [1]=1); N=1

while true
do
   echo -n "Enter n: "; read n
   if [ $n -lt 0 ]; then echo "Enter a positive integer please"; continue; fi
   if [ $n -gt $N ]; then
      echo "Computing F($((N+1))) through F($n)"
      computerest $((N+1)) $n
      N=$n
   fi
   echo "F($n) = ${F[$n]}"
   until false
   do
      echo -n "Repeat (y/n)? "; read resp
      case $resp in
         y|Y) break ;;
         n|N) echo "Bye..."; exit 0 ;;
         *) echo "Invalid response. Retry...";;
      esac
   done
done
```

Running fibiter.sh

```
$ ./fibiter.sh
Enter n: 16
Computing F(2) through F(16)
F(16) = 987
Repeat (y/n)? y
Enter n: 32
Computing F(17) through F(32)
F(32) = 2178309
Repeat (y/n)? Y
Enter n: 64
Computing F(33) through F(64)
F(64) = 10610209857723
Repeat (y/n)? U
Invalid response. Retry...
Repeat (y/n)? Y
Enter n: -40
Enter a positive integer please
Enter n: 40
F(40) = 102334155
Repeat (y/n)? n
Bye...
$
```

A script to recursively print directory trees

dirtree.sh

```
#!/bin/bash

function exploredir () {
   local currentdir=$1
   local currentlev=$2
   local lev=0

   while [ $lev -lt $currentlev ]; do echo -n "        "; lev=$((lev+1)); done
   echo -n $currentdir
   if [ ! -r "$currentdir" ] || [ ! -x "$currentdir" ]; then
      echo " [Unable to explore further]"
   else
      echo ""
      for entry in `ls "$currentdir"`; do
         if [ -d "$currentdir/$entry" ]; then
            exploredir "$currentdir/$entry" $((currentlev+1))
         fi
      done
   fi
}

if [ $# -eq 0 ]; then rootdir=.; else rootdir=$1; fi
if [ ! -d "$rootdir" ]; then echo "$rootdir is not a directory"; exit 1; fi
exploredir "$rootdir" 0
```

Running dirtree.sh

```
$ ./dirtree.sh /usr/local
/usr/local
        /usr/local/bin
        /usr/local/etc
        /usr/local/games
        /usr/local/include
        /usr/local/lib
                /usr/local/lib/python3.8
                        /usr/local/lib/python3.8/dist-packages
        /usr/local/man
        /usr/local/sbin
        /usr/local/share
                /usr/local/share/ca-certificates
                /usr/local/share/emacs
                        /usr/local/share/emacs/site-lisp
                /usr/local/share/fonts
                /usr/local/share/man
                /usr/local/share/sgml
                        /usr/local/share/sgml/declaration
                        /usr/local/share/sgml/dtd
                        /usr/local/share/sgml/entities
                        /usr/local/share/sgml/misc
                        /usr/local/share/sgml/stylesheet
                /usr/local/share/texmf
                /usr/local/share/xml
                        /usr/local/share/xml/declaration
                        /usr/local/share/xml/entities
                        /usr/local/share/xml/misc
                        /usr/local/share/xml/schema
        /usr/local/src
        /usr/local/WinFIG
                /usr/local/WinFIG/Documentation [Unable to explore further]
                /usr/local/WinFIG/plugins [Unable to explore further]
                /usr/local/WinFIG/Scripts [Unable to explore further]
$
```

Writing C-style loops

- Use `((...))`.
- `for (( i = 0; i < 10; ++i )) do echo $i; done`
- `i=0`
  `while (( i < 10 )); do echo $i; (( ++i )); done`

Reading a file line-by-line in an array

file2array.sh

```
#!/bin/bash

[ $# -ge 1 ] || exit 1;

for fname in $@; do
   if [ ! -f $fname ] || [ ! -r $fname ]; then
      echo "--- Unable to read $fname"
      continue
   fi
   echo -n "+++ Reading file $fname: "
   L=()
   while read -r line; do
      L+=("$line")
   done < $fname
   echo "${#L[@]} lines read"
done
```

```
$ ./file2array.sh ~/spl/prog/libstaque/*
+++ Reading file /home/foobar/spl/prog/libstaque/Makefile: 17 lines read
--- Unable to read /home/foobar/spl/prog/libstaque/shared
--- Unable to read /home/foobar/spl/prog/libstaque/static
$ ./file2array.sh ~/spl/prog/libstaque/static/*.?
+++ Reading file /home/foobar/spl/prog/libstaque/static/defs.h: 11 lines read
+++ Reading file /home/foobar/spl/prog/libstaque/static/queue.c: 82 lines read
+++ Reading file /home/foobar/spl/prog/libstaque/static/queue.h: 17 lines read
+++ Reading file /home/foobar/spl/prog/libstaque/static/stack.c: 79 lines read
+++ Reading file /home/foobar/spl/prog/libstaque/static/stack.h: 14 lines read
+++ Reading file /home/foobar/spl/prog/libstaque/static/staque.h: 8 lines read
$
```

Finding all matches of a regular expression in a file

allmatches.sh

```
#!/bin/bash

if [ $# -lt 2 ]; then echo "Usage: $0 filename regexp"; exit 1; fi
pattern="(.*)($2)(.*)"
nmatch=0
while read -r line; do
   T=(); M=()
   while true; do
      if [[ ! $line =~ $pattern ]]; then
         T+=("$line")
         break
      fi
      T+=("${BASH_REMATCH[3]}")
      M+=("${BASH_REMATCH[2]}")
      line="${BASH_REMATCH[1]}"
   done
   l=$(( ${#T[@]} - 1))
   nmatch=$(( nmatch + 1 ))
   echo -n "${T[$l]}"
   for (( i=l-1; i>=0; --i)) do
      echo -n "[${M[$i]}]${T[$i]}"
   done
   echo
done < $1
echo "+++ Total number of matches = $nmatch"
```

Running allmatches.sh

```
$ ./allmatches.sh textfile.txt '[A-Z][^A-Z]*[a-z]'
[Abstract]

[This tutorial focuses on algorithms for factoring large composite integers]
and for computing discrete logarithms in large finite fields. [In order to]
make the exposition self-sufficient, [I start with some common and popular]
public-key algorithms for encryption, key exchange, and digital signatures.
[These algorithms highlight the roles played by the apparent difficulty of]
solving the factoring and discrete-logarithm problems, for designing
public-key protocols.

[Two exponential-time integer-factoring algorithms are first covered]:
trial division and [Pollard's rho method]. [This is followed by two]
sub-exponential algorithms based upon [Fermat's factoring method]. [Dixon's]
method uses random squares, but illustrates the basic concepts of the
relation-collection and the linear-algebra stages. [Next], [I introduce the]
[Quadratic] [Sieve] [Method] (QS[M) which brings the benefits of using small]
candidates for smoothness testing and of sieving.

[As the third module], [I formally define the discrete-logarithm problem] (DLP)
and its variants. [As a representative of the square-root methods for solving]
the DL[P, the baby-step-giant-step method is explained]. [Next], [I introduce the]
index calculus method (IC[M) as a general paradigm for solving the] DLP.
[Various stages of the basic] IC[M are explained both for prime fields and]
for extension fields of characteristic two.

+++ Total number of matches = 25
```

Note: Compare this with the output of: `grep '[A-Z][^A-Z]*[a-z]' textfile.txt`

Sample exercises

1. Write a bash script that takes multiple arguments, and checks which of these arguments is/are Fibonacci number(s).

2. Write a bash script that takes a positive integer as an argument, and add commas to separate thousands, lakhs, and crores. For example, for input 123456, the output should be 1,23,456, and for input 123456789, the output should be 12,34,56,789.

3. Write a bash script (similar to filetree.sh) that recursively lists the entire file tree under a given directory. For non-directories, the recursive call is not made. If no directory is supplied, take . as the root directory.

4. Write a bash script that takes two directories dir1 and dir2 as arguments, and prints the common names of files in the two directories.

5. Write a bash script that takes a directory as an argument, and keeps on listing every minute only the names of the files in that directory, that are modified (after the last print). The first listing is that of all the files in the directory. Modification includes new files, deleted files, and updated files.

6. Write a bash script that finds all the regular (that is, non-system) users in the system. Assume that regular users have user IDs $\geqslant 1000$. (Read **`/etc/passwd`**.)