CS29206 Systems Programming Laboratory
Spring 2024

Programming bash

Abhijit Das
Pralay Mitra

Why shell programming?

- You can write C/C++/Java/Python/. . . programs for every doable thing.
- Precompiled libraries make your job easier.
- C programs are naturally good for number crunching, data structuring, . . .
- Specially written programs can do special tasks with little programming efforts.
    - grep specializes in pattern matching.
    - gawk specializes in text data processing.
    - A shell like bash specializes in many types of file handling.
- C programs for these special jobs are often huge and difficult to write.
- A shell script is an all-in-one solution to simplify a programmer's life.
    - A shell itself does whatever it is naturally good at.
    - For special tasks, it can call the specialists with little effort.
    - Shell scripts are very useful for system administration.

What have you seen, and what next?

- What you already know:
  - How bash can execute commands.
  - How bash can manage variables and arrays.
  - How bash can define functions.
  - How bash can do arithmetic operations using `$((...))`.
  - How bash can store the complete outputs (not the return values) produced by other programs using back-quotes or `$(...)`.
  - How bash can do pattern-based substitutions in command lines.
- What remains for you to know is the control structures.
  - Condition checking
  - Conditional execution
  - Loops

Introductory concepts

Your first shell script

- First line: The hash-bang or she-bang notation specifies the interpreter.
- Then, write the shell commands and directives.
- Add execute permission to the shell script.
- Run the script.

File hello.sh

```
#!/bin/bash
echo "Hello, world!"
```

Running hello.sh

```
$ chmod 755 hello.sh
$ ./hello.sh
Hello, world!
$
```

An interactive shell script to list all files of an extension

The script findall.sh

```
#!/bin/bash
echo -n "*** Enter an extension (without the dot): "
read extn
echo "*** Okay, finding all files in your home area with extension $extn"
ls -R ~ | grep "\.$extn$"
echo "*** That's all you have. Bye."
```

Running findall.sh

```
$ chmod a+x findall.sh
$ ./findall.sh
*** Enter an extension (without the dot): tif
*** Okay, finding all files in your home area with extension tif
centralimage-1500.tif
formulas-hires.tif
frontcover-hires.tif
Crypto.tif
left.tif
dataconv2.tif
ICDCN_DD.tif
ICDCN_LNCS.tif
ICDCN_REGN.tif
lncs-logo_4c.tif
*** That's all you have. Bye.
$
```

You can supply regular expressions in extension

```
$ ./findall.sh
*** Enter an extension (without the dot): [A-Z]
*** Okay, finding all files in your home area with extension [A-Z]
LABTEST.C
gf2n.S
test.S
template17.Z
*** That's all you have. Bye.
$ ./findall.sh
*** Enter an extension (without the dot): [a-z]*[^a-zA-Z]
*** Okay, finding all files in your home area with extension [a-z]*[^a-zA-Z]
crypto.toc7
cfp.html~
bwedit3.0
words.2
2021-11-15.mp4
MontgomeryLadder.gp~
Numberlink.mp3
*** That's all you have. Bye.
$
```

Running another interpreter

rungawk.sh

```
#!/bin/bash
echo -n "Enter dinosaur database file: "
read dbfile
gawk '
   BEGIN { FS=":"; print "Theropod dinosaurs" }
   {
      if ($2 ~ "theropod") { print "\t" $1; n++ }
   }
   END { print n " theropods found" }
' $dbfile
```

Note:

- `$1` and `$2` have different meanings in bash and gawk.
- Since the commands of gawk are within single quotes, bash does not expand `$1` and `$2`.

Output of rungawk.sh

```
$ ./rungawk.sh
Enter dinosaur database file: ../awk/dinosaurs.txt
Theropod dinosaurs
        Albertosaurus
        Allosaurus
        Archaeopteryx
        Baryonyx
        Carcharodontosaurus
        Carnotaurus
        Ceratosaurus
        Chindesaurus
        Coelophysis
        Deinocheirus
        Deinonychus
        Dilophosaurus
        Giganotosaurus
        Indosuchus
        Majungasaurus
        Megalosaurus
        Microraptor
        Monolophosaurus
        Oviraptor
        Sinraptor
        Spinosaurus
        Tarbosaurus
        Tyrannosaurus
        Utahraptor
        Velociraptor
25 theropods found
$
```

Using here documents

rungawkfile.sh

```
#!/bin/bash
echo -n "Enter dinosaur database file: "
read dbfile
cat << EOP > thero.awk
   BEGIN { FS=":"; print "Theropod dinosaurs" }
   {
      if (\$2 ~ "theropod") { print "\t" \$1; n++ }
   }
   END { print n " theropods found" }
EOP
gawk -f thero.awk $dbfile
```

Notes:

- `echo` (in place of `cat`) does not work here. Why?
- Here documents expand the variables. To prevent this from happening, you should use `\$1` and `\$2`.

Output of rungawkfile.sh

```
$ ./rungawkfile.sh
Enter dinosaur database file: ../awk/dinosaurs.txt
Theropod dinosaurs
        Albertosaurus
        Allosaurus
        Archaeopteryx
        Baryonyx
        Carcharodontosaurus
        Carnotaurus
        Ceratosaurus
        Chindesaurus
        Coelophysis
        Deinocheirus
        Deinonychus
        Dilophosaurus
        Giganotosaurus
        Indosuchus
        Majungasaurus
        Megalosaurus
        Microraptor
        Monolophosaurus
        Oviraptor
        Sinraptor
        Spinosaurus
        Tarbosaurus
        Tyrannosaurus
        Utahraptor
        Velociraptor
25 theropods found
$
```

Storing the output of another program in a string

rungawkstore.sh

```
#!/bin/bash
echo -n "Enter dinosaur database file: "
read dbfile
cat << EOP > thero.awk
   BEGIN { FS=":"; print "Theropod dinosaurs" }
   {
      if (\$2 ~ "theropod") { print "\t" \$1; n++ }
   }
   END { print n " theropods found" }
EOP
gawkop=`gawk -f thero.awk $dbfile`
echo "gawk produced the following output..."
echo $gawkop
```

Running the script

```
$ ./rungawkstore.sh
Enter dinosaur database file: ../awk/dinosaurs.txt
gawk produced the following output...
Theropod dinosaurs Albertosaurus Allosaurus Archaeopteryx Baryonyx Carcharodontosaurus Carnotaurus Ceratosaurus Chindesaurus
Coelophysis Deinocheirus Deinonychus Dilophosaurus Giganotosaurus Indosuchus Majungasaurus Megalosaurus Microraptor
Monolophosaurus Oviraptor Sinraptor Spinosaurus Tarbosaurus Tyrannosaurus Utahraptor Velociraptor 25 theropods found
$
```

Note: Use `echo "$gawkop"` to see the correctly formatted output.

Processing the stored output

rungawkgrep.sh prints only the theropod dinosaur names not ending with s

```
#!/bin/bash
echo -n "Enter dinosaur database file: "
read dbfile
cat << EOP > thero.awk
   BEGIN { FS=":"; print "Theropod dinosaurs" }
   {
      if (\$2 ~ "theropod") { print "\t" \$1; n++ }
   }
   END { print n " theropods found" }
EOP
gawkop=`gawk -f thero.awk $dbfile`
echo "Output of gawk is filtered through grep..."
echo "$gawkop" | grep "^[^a-zA-Z0-9].*[^s]$" -
```

Running the script

```
$ ./rungawkgrep.sh
Enter dinosaur database file: ../awk/dinosaurs.txt
Output of gawk is filtered through grep...
        Archaeopteryx
        Baryonyx
        Microraptor
        Oviraptor
        Sinraptor
        Utahraptor
        Velociraptor
$
```

Return modes revisited

- Every command returns a value.
- Your shell functions also run as commands.
- The return value is to be treated as a status.
- The status is usually a small integer in the range [0, 255].
- For returning other things (larger integers, floating-point values, and strings), you have to use other mechanisms.
- Status is to be treated as status, not as value.
- Use one of the following mechanisms.
    - Returning by setting global variable(s).
    - Returning by echoing.

Return values through global variables

hypo1.sh

```
#!/bin/bash

function hypotenuse () {
   local a=$1;
   local b=$2;
   a=$((a*a))
   b=$((b*b))
   csqr=$((a+b))
   c=`echo "scale=10; sqrt($csqr)" | bc`
}

echo -n "Enter a and b: "
read a b
hypotenuse $a $b
echo "a = $a, b = $b, c = $c, csqr = $csqr"
```

Returning by echoing

Running the script

```
$ ./hypo1.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759, csqr = 61
$
```

Return values by echoing

hypo2.sh

```
#!/bin/bash

function hypotenuse () {
   local a=$1;
   local b=$2;
   a=$((a*a))
   b=$((b*b))
   csqr=$((a+b))
   echo `echo "scale=10; sqrt($csqr)" | bc`
}

echo -n "Enter a and b: "
read a b
c=`hypotenuse $a $b`
echo "a = $a, b = $b, c = $c"
```

Running the script

```
$ ./hypo2.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759
$
```

You have a price to pay

hypo3.sh

```
#!/bin/bash

function hypotenuse () {
   local a=$1;
   local b=$2;
   a=$((a*a))
   b=$((b*b))
   csqr=$((a+b))
   echo `echo "scale=10; sqrt($csqr)" | bc`
}

echo -n "Enter a and b: "
read a b
csqr="Not yet computed"
c=`hypotenuse $a $b`
echo "a = $a, b = $b, c = $c, csqr = $csqr"
```

Running the script

```
$ ./hypo3.sh
Enter a and b: 5 6
a = 5, b = 6, c = 7.8102496759, csqr = Not yet computed
$
```

What happened to csqr?

- Whenever you run a command using `` `...` `` or `$(...)`, **a sub-shell is opened**.
- A function call also works like a command.
- Any changes in the global variables of **this** shell, that you make in the **sub-shell**, have **no effect** in **this** shell.
- This happens even if you export your variables.
- This is the difference between

  ```
  cmd arg1 arg2 ...
  ```

  and

  ```
  storedop=`cmd arg1 arg2 ...`
  echo "$storedop"
  ```

- In the first case, `cmd` is executed in **this** shell, and in the second case, in a **sub-shell**.

Logical conditions

Overview

- Needed for conditional execution of blocks, and in loops.
- Unlike C, 0 means True, and non-zero means False.
- A command returns a status.
- The return status indicates true (successful completion) or false (unsuccessful completion).
- The return status can be accessed as `$?`.
- Conditions can be logically joined by `||` or `&&`, or negated by `!`.
- Use parentheses `(` and `)` for disambiguation (if needed).
- Other types of conditions
    - Results of numeric comparisons
    - Results of string comparisons
    - Conditions on file attributes
- These other conditions can be checked as `test condition` or as `[ condition ]`.
- Note the space after `[` and before `]`.

Checking return status

Note: `&&` and `||` are short-circuit operators.

```
$ ls ~/[a-z].* ; echo $?
ls: cannot access '/home/foobar/[a-z].*': No such file or directory
2
$ ls ~/*.[a-z] ; echo $?
/home/foobar/assignment1.c   /home/foobar/assignment2.c   /home/foobar/assignment3.c
0
$ ls ~/[a-z].* && ls ~/*.[a-z] ; echo $?
ls: cannot access '/home/foobar/[a-z].*': No such file or directory
2
$ ls ~/[a-z].* || ls ~/*.[a-z] ; echo $?
ls: cannot access '/home/foobar/[a-z].*': No such file or directory
/home/foobar/assignment1.c   /home/foobar/assignment2.c   /home/foobar/assignment3.c
0
$ ls ~/*.[a-z] && ls ~/[a-z].* ; echo $?
/home/foobar/assignment1.c   /home/foobar/assignment2.c   /home/foobar/assignment3.c
ls: cannot access '/home/foobar/[a-z].*': No such file or directory
2
$ ls ~/*.[a-z] || ls ~/[a-z].* ; echo $?
/home/foobar/assignment1.c   /home/foobar/assignment2.c   /home/foobar/assignment3.c
0
$ ! ls ~/[a-z].* ; echo $?
ls: cannot access '/home/foobar/[a-z].*': No such file or directory
0
$ ! ls ~/[a-z].* && ls ~/*.[a-z] ; echo $?
ls: cannot access '/home/foobar/[a-z].*': No such file or directory
/home/foobar/assignment1.c   /home/foobar/assignment2.c   /home/foobar/assignment3.c
0
$
```

Numeric comparisons

- **Syntax:** `[ EXPR1 -comp_op EXPR2 ]`
- Numeric comparisons apply to integer values only.
- Fractional/non-numeric/undefined values lead to errors.
- The comparison operators are as follows. Here, “if” means “if and only if”.

`-eq`	True if the two expressions are equal.
`-ne`	True if the two expressions are unequal.
`-lt`	True if the first expression is less than the second.
`-le`	True if the first expression is less than or equal to the second.
`-gt`	True if the first expression is greater than the second.
`-ge`	True if the first expression is greater than or equal to the second.

Examples of numeric comparison

```
$ x=3; y=4; z=5
$ [$y -eq 3]; echo $?
[4: command not found
127
$ [ $y -eq 3]; echo $?
bash: [: missing ‘]’
2
$ [ $y -eq 3 ]; echo $?
1
$ [ $y -gt $x ]; echo $?
0
$ [ $y -gt $z ]; echo $?
1
$ [ $y -gt $x ] && [ $y -gt $z ]; echo $?
1
$ [ $y -gt $x ] && [ ! $y -gt $z ]; echo $?
0
$ [ $y -gt $x ] || [ $y -gt $z ]; echo $?
0
$ [ $((x**2 + y**2)) -eq $((z**2)) ]; echo $?
0
$ w=‘echo "scale=10; sqrt($z)" | bc‘; echo $w
2.2360679774
$ [ $w -le $x ]; echo $?
bash: [: 2.2360679774: integer expression expected
2
$ [ ! $w -le $x ]; echo $?
bash: [: 2.2360679774: integer expression expected
2
$
```

String comparisons

- Strings can be compared for equality/inequality.
- A string with space(s) should be quoted.

`[ STR1 = STR2 ]`	True if the two strings are equal.
`[ STR1 == STR2 ]`	True if the two strings are equal.
`[ STR1 != STR2 ]`	True if the two strings are unequal.
`[ -z STR ]`	True if `STR` is an empty/undefined string.
`[ -n STR ]`	True if `STR` is a non-empty string.

Examples of string comparisons

```
$ x="Foolan"; y="Foolan Barik"
$ [ $x = $y ]; echo $?
bash: [: too many arguments
2
$ [ "$x" == "$y" ]; echo $?
1
$ [ ! "$x" == "$y" ]; echo $?
0
$ [ "$x" != "$y" ]; echo $?
0
$ [ -z "$z" ]; echo $?
0
$ z=""; [ -z "$z" ]; echo $?
0
$ z=" "; [ -z "$z" ]; echo $?
1
$ z=" "; [ -z $z ]; echo $?
0
$
```

Conditions based on file attributes

`[ -e FILE ]`	True if `FILE` exists
`[ -f FILE ]`	True if `FILE` exists and is a regular file
`[ -s FILE ]`	True if `FILE` exists and is non-empty
`[ -d FILE ]`	True if `FILE` exists and is a directory
`[ -r FILE ]`	True if `FILE` exists and has read permission
`[ -w FILE ]`	True if `FILE` exists and has write permission
`[ -x FILE ]`	True if `FILE` exists and has execute permission
`[ FILE1 -nt FILE2 ]`	True if `FILE1` is newer than `FILE2`
`[ FILE1 -ot FILE2 ]`	True if `FILE1` is older than `FILE2`

File conditions: Example

filecheck.sh

```
#!/bin/bash

[ $# -eq 0 ] && { echo "Run with a command-line argument"; exit 1; }
[ ! -e "$1" ] && { echo "\"$1\" does not exist"; exit 0; }
echo "\"$1\" exists"
[ -f "$1" ] && echo "\"$1\" is a regular file"
[ ! -f "$1" ] && echo "\"$1\" is not a regular file"
[ -d "$1" ] && echo "\"$1\" is a directory"
[ ! -d "$1" ] && echo "\"$1\" is not a directory"
echo -n "Permissions:"
[ -r "$1" ] && echo -n " read"
[ -w "$1" ] && echo -n " write"
[ -x "$1" ] && echo -n " execute"
echo ""
```

Run filecheck.sh

```
$ ./filecheck.sh
Run with a command-line argument
$ ./filecheck.sh filecheck.sh
"filecheck.sh" exists
"filecheck.sh" is a regular file
"filecheck.sh" is not a directory
Permissions: read write execute
$ ./filecheck.sh /usr/
"/usr/" exists
"/usr/" is not a regular file
"/usr/" is a directory
Permissions: read execute
$ ./filecheck.sh /dev/null
"/dev/null" exists
"/dev/null" is not a regular file
"/dev/null" is not a directory
Permissions: read write
$ ./filecheck.sh /etc/passwd
"/etc/passwd" exists
"/etc/passwd" is a regular file
"/etc/passwd" is not a directory
Permissions: read
$ ./filecheck.sh ~/spl/*
"/home/foobar/spl/asgn" exists
"/home/foobar/spl/asgn" is not a regular file
"/home/foobar/spl/asgn" is a directory
Permissions: read write execute
$
```

Disambiguation of logical expressions

- Let $A = F$, $B = F$, and $C = T$.
- So $AB + C = (AB) + C = F + T = T$, whereas $A(B + C) = F(F + T) = FT = F$.
- In bash, `&&` and `||` have the **same precedence**.
- **Left-to-right associativity** is used for disambiguation.
- $A + BC$ is interpreted as $(A + B)C$ which evaluates to $(T + F)F = TF = F$.
- If you mean $A + (BC)$, use parentheses, so it evaluates to $T + (FF) = T + F = T$.
- `true` and `false` are the constant values T and F.

```
$ [ "abc" == "a b c" ] && [ 5 -eq $((3+4)) ] || [ ! -f /dev/null ] ; echo $?
0
$ [ "abc" == "a b c" ] && ( [ 5 -eq $((3+4)) ] || [ ! -f /dev/null ] ) ; echo $?
1
$ [ ! -f /dev/null ] || [ "abc" == "a b c" ] && [ 5 -eq $((3+4)) ] ; echo $?
1
$ [ ! -f /dev/null ] || ( [ "abc" == "a b c" ] && [ 5 -eq $((3+4)) ] ) ; echo $?
0
$
```

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, . . . , 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/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/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 + l ))
   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`

Practice 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 dirtree.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 the 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`.)