Systems Programming Laboratory, Spring 2022

Programming bash

Abhijit Das
Arobinda Gupta

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

March 24, 2022

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"
```

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/abhij/spl/asgn" exists
"/home/abhij/spl/asgn" is not a regular file
"/home/abhij/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
$
```