
Review

- **' and '\$'**
 - These symbols indicate the start and the end of a string, respectively.
- **'*', '+', and '?'**
 - The symbols '*', '+', and '?' denote the number of times a character or a sequence of characters may occur. What they mean is: "zero or more", "one or more", and "zero or one."
- **Braces {}**
 - Bounds, which appear inside braces, indicate ranges in the number of occurrences
- **! OR operator:**
 - Works as an OR operator:
- **('.');**
 - A period (') stands for any single character:
- **Bracket [] expressions**
 - specify which characters are allowed in a single position of a string:

Shell Scripting and Regular Expression

Week 2

Review

- **Environmental Variables (LC_*)**
- **Text Processing Tools (sort, wc, head, tail)**
- **Basic I/O Redirection (<, >, >>, 2>) and pipeline (|)**
- **Search for text (grep)**
- **File comparison (diff, comm, cmp)**
- **File Processing: tr, sed**

CS 35L- Software Construction Laboratory

Fall 2018

TA: Guangyu Zhou

Lab 3

Shell

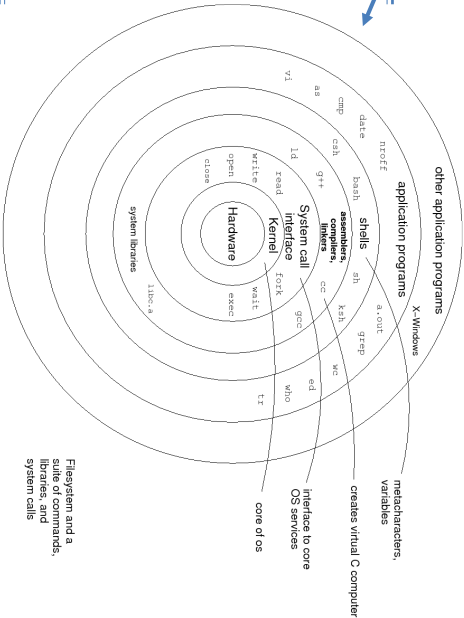
- The shell is the user's interface to the OS
- From it you run programs.
- Example of shells
 - bash, zsh, csh, sh, tcsh
- Allow more complex functionality than interacting with OS directly
 - Tab complete, easy redirection

Shell Scripting

- The ability to enter **multiple commands** and **combine** them logically
- Must specify the shell you use in the first line
 - **#!/bin/bash**
 - (# itself can lead comments)
- You can create easiest shell script by listing commands in separate lines
- (shell will process commands in order)

Outline

- Advanced Linux Commands
- Regular Expression
- The Shell Scripting



Conceptual Architecture of UNIX SYSTEMS

From: *The Design of the UNIX Operating System*

Why Shell

- Simplicity
 - Far easier to write and debug a shell script than a C/C++ program. Especially for system administration tasks which include execution of **external commands, creating and removing files and directories, redirecting output**, etc.
 - C/C++ programs are better for a much lower level of operation, such as invoking system calls, manipulating data structures, etc.
- Portability
 - A **shell script** can be transferred to other Unix and Unix-like operating systems and executed (if the shell itself is present).
 - Even when transferring a shell script from different architectures such as x86, MIPS, shell scripts are much more portable than C/C++ programs.

Scripting Languages VS. Compiled Languages

- Compiled languages (e.g. C++, Java)
 - Programs are translated from their original source code into object code that is executed by hardware [human-readable -> machine-readable]
 - Efficient
 - Work at low level, dealing with bytes, integers, floating points, etc
- Scripting languages (e.g. Ruby, Perl)
 - Interpreted
 - Interpreter reads program, translates it into internal form, and execute programs
 - Relatively inefficient (translation on the fly)

Example

```
$ who | grep john           where is john?
john pts/3 Dec 27 11:07 (flags-r-us.example.com)

Script:
# i /bin/sh
# finduser --- see if user named by john is logged in
who | grep john
```

```
Run it:
$ chmod +x finduser      Make it executable
$ ./finduser             Test it: find john
john pts/3 Dec 27 11:07 (flags-r-us.example.com)
```

Example:

- How to write a shell script that can find the logged in user with username "john"?
 - What command to use
 - How to write and run the script

Variables

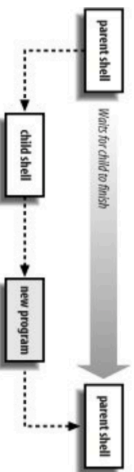
- Allows you to temporarily store info and use it later
- Two general types
 - Environment variables
 - User defined variables (UDV)
- **Environment variables**
 - Created and maintained by Linux itself
 - Track specific system info
 - defined in CAPITAL LETTERS
 - ex: \$PATH, \$PWD
- **User defined variables (UDV)**
 - Created and maintained by user
 - defined in lower letters

Variables

- When refer a variable to assign a value to it, do not use dollar sign (**no space around =**)
`myvar=helloworld`
- **export:**
 - puts variables into the environment. Environment is a list of name-value pairs that is available to every running program
- **env:**
 - Displays the current environment
- **unset:**
 - remove variable and functions from the current shell

The #! First Line

- A shell script is just a file with shell commands.
- When the shell runs a program (e.g. finduser), it asks the kernel to start a new **“child process” and run the given program in that process.**
- First line is used to state which “child shell” to use:
 - `#!/bin/sh`
 - `#!/bin/csh -f`
 - `#!/bin/ah`



Variables

- Start with a letter or underscore and may contain any number of following letters, digits, or underscores
- Declared/assigned using =
 - `Var=hello world`
- Referenced with \$
 - `echo $PATH`
- Reminder - echo prints to screen

Example with Command Line Parameters

```
$ who | grep john           where is john?
john pts/3 Dec 27 11:07 (flags-r-us.example.com)

Script:
#! /bin/sh
# finduser --- see if user named by john is logged in
who | grep john $1

Run it:
$ chmod +x finduser        Make it executable
$ ./finduser john          Test it: find john
john pts/3 Dec 27 11:07 (flags-r-us.example.com)
```

Special Variables

- \$0 The filename of the current script.
- \$n The input arguments. the first is \$1, the second is \$2, etc
- \$? Show exit status of previous command
 - - 0 means exit normally
 - - Otherwise exit with some errors
- \$PATH
- Other: see supplement materials

POSIX Built-in Shell Variables

Variable	Meaning
#	Number of arguments given to current process.
@	Command line arguments to current process. Inside double quotes, expands to individual arguments.
*	Command line arguments to current process. Inside double quotes, expands to a single argument. Option given to aid on invocation.
- (hyphen)	Exit status of previous command.
?	Process ID of shell process.
\$	Process ID of shell process.
0 (zero)	The name of the shell program.
!	Process ID of last background command. Use this to save process ID numbers for later use with the wait command.
ENV	Used only by interactive shells upon invocation, the value of ENV is parameter expanded. The result should be a list of names for a file to be read and executed as scripts. This is an shell requirement.
HISTFILE	Name of history file.
HISTSIZE	How many lines of history are kept. Default is 500 lines. Maximum is 5000 lines. Note: some shells allow a larger value.
IFS	Default values of current locale, controlled by the shell C.L. variables.
LANG	Default locale.
LC_ALL	Force the use of a specific locale completely. Overrides LC_* variables.
LC_COLLATE	Names of collating sequences used for string comparison.
LC_CTYPE	Names of character sets used for character classification.
LC_MESSAGES	Names of current language for default messages.
LINENO	Line number in script or function of the line that last ran.
MAIL	The location of message catalogs for messages in the language given by \$LC_MESSAGES (XSI).
PATH	Search path for commands.
PPID	Process ID of parent process.
PS1	Primary command prompt string. Default is "\$ ".
PS2	Prompt string for line continuations. Default is "> ".
PS4	Prompt string for execution tracing with set -x. Default is "> ".
PWD	Current working directory.

Exit: Return value

Value	Meaning
0	Command exited successfully.
> 0	Failure during redirection or word expansion (file, variable, command, and arithmetic expansions, as well as word splitting).
1-125	Command exited unsuccessfully. The meanings of particular exit values are defined by each individual command.
126	Command found, but file was not executable.
127	Command not found.
> 128	Command died due to receiving a signal.

Arithmetic Operators

Operator	Meaning	Associativity
++ --	Increment and decrement, prefix and postfix	Left to right
+ - ! ~	Unary plus and minus, logical and bitwise negation	Right to left
* / %	Multiplication, division, and remainder	Left to right
+ -	Addition and subtraction	Left to right
<< >>	Bit-shift left and right	Left to right
< <= > >=	Comparisons	Left to right
= !=	Equal and not equal	Left to right
&	Bitwise AND	Left to right
^	Bitwise Exclusive OR	Left to right
	Bitwise OR	Left to right
&&	Logical AND (short-circuit)	Left to right
	Logical OR (short-circuit)	Left to right
?:	Conditional expression	Right to left
= += -= *= /= %> &= ^> =	Assign ment operators	Right to left

Structured command

- Alter the flow of operations based conditions
- If statement
- For statement
- While loops
- Case statement
- Break statement
- Continue statement

Arithmetic expression

- Let
- ((math expression))
- Ex:
let z=5; echo \$z
let z=z+5; echo \$z
((z = z + 5)); echo \$z
- (()) also used as numerical Boolean expression in control constructs
- Ex:
 - If ((z > 0)); then
 - echo "\$z is positive"
 - fi

Quote

- Single quote '
 - Literal meaning of everything within ''
 - echo '\$PATH'
 - Double quote "
 - Literal meaning of everything except \$ \ `
 - echo "the current directory is \$PWD"
 - The backtick `
 - Execute the command
 - Allow you to assign the output of a shell command to a variable
 - testing `date`
- ```
$ echo "this is $PATH"
$ echo 'this is $PATH'
$ echo `ls`
```

---

## IF-THEN Statement

---

- If the exit status of **command** is **zero** (complete successfully), the command listed under then ***then*** section are **executed**

```
#!/bin/bash
testing the if statement
if date
then
 echo "it worked"
fi

if grep pattern myfile > /dev/null
then
 ... Pattern is there
else
 ... Pattern is not there
fi
```

---

## Test command

---

- Three classes of conditions
  - - Numeric comparisons
  - - String comparisons
  - - File comparisons

---

## if-elif-else-fi

---

```
if command
then
 statements-if-true-1
[elif command
then
 statements-if-true-2
...]
[else
 statements-if-all-else-fails]
fi
```

---

## Test command

---

- The ability to evaluate any condition other than the exit code of a status (ie. evaluate true/false)

```
if test condition
then
 commands
fi

if [condition]
then
 commands
fi
```
- If the **condition** listed in the test command is **true**, the test command **exits with 0**

## Test command: String comparisons

- The greater-than and less-than symbols must be escaped (otherwise will be interpreted as redirection)
- The greater-than and less-than order is not the same as sort (ASCII vs. locale language)
- Ex: \$USER = \$testuser

| Comparison                  | Description                                                          |
|-----------------------------|----------------------------------------------------------------------|
| <code>str1 = str2</code>    | Check if <code>str1</code> is the same as string <code>str2</code> . |
| <code>str1 != str2</code>   | Check if <code>str1</code> is not the same as <code>str2</code> .    |
| <code>str1 &lt; str2</code> | Check if <code>str1</code> is less than <code>str2</code> .          |
| <code>str1 &gt; str2</code> | Check if <code>str1</code> is greater than <code>str2</code> .       |
| <code>in str2</code>        | Check if <code>str1</code> has a length greater than zero            |
| <code>-z str1</code>        | Check if <code>str1</code> has a length of zero                      |

## Test command: Numeric comparisons

- Evaluate both numbers and variables
- Ex: \$var -eq 1, \$var1 -ge \$var2

| Comparison             | Description                                                            |
|------------------------|------------------------------------------------------------------------|
| <code>n1 -eq n2</code> | Check if <code>n1</code> is equal to <code>n2</code> .                 |
| <code>n1 -ge n2</code> | Check if <code>n1</code> is greater than or equal to <code>n2</code> . |
| <code>n1 -gt n2</code> | Check if <code>n1</code> is greater than <code>n2</code> .             |
| <code>n1 -le n2</code> | Check if <code>n1</code> is less than or equal to <code>n2</code> .    |
| <code>n1 -lt n2</code> | Check if <code>n1</code> is less than <code>n2</code> .                |
| <code>n1 -ne n2</code> | Check if <code>n1</code> is not equal to <code>n2</code> .             |

## Conditions Example

- Comparison
  - eq equal to
  - usage: if [ "\$X" -eq "1" ]     In C++, it is: if(x == 1)
  - (())
  - usage: if ((x > 1))

- File
  - f     is a file
  - usage: if [ -f "\$dir" ]     if variable \$dir is a file
  - Other notations: see supplement materials

## Test command: File comparisons

- Test the status of files and directories in linux file system

| Comparison                   | Description                                                                               |
|------------------------------|-------------------------------------------------------------------------------------------|
| <code>-d file</code>         | Check if <code>file</code> exists and is a directory.                                     |
| <code>-e file</code>         | Checks if <code>file</code> exists.                                                       |
| <code>-f file</code>         | Checks if <code>file</code> exists and is a file.                                         |
| <code>-r file</code>         | Checks if <code>file</code> exists and is readable.                                       |
| <code>-s file</code>         | Checks if <code>file</code> exists and is not empty.                                      |
| <code>-w file</code>         | Checks if <code>file</code> exists and is writable.                                       |
| <code>-x file</code>         | Checks if <code>file</code> exists and is executable.                                     |
| <code>-0 file</code>         | Checks if <code>file</code> exists and is owned by the current user.                      |
| <code>-G file</code>         | Checks if <code>file</code> exists and the default group is the same as the current user. |
| <code>file1 -nt file2</code> | Checks if <code>file1</code> is newer than <code>file2</code> .                           |
| <code>file1 -ot file2</code> | Checks if <code>file1</code> is older than <code>file2</code> .                           |



---

## for loops

---

```
for i in atlbroschure*.xml
do
 echo $i
 mv $i $i.old
 sed 's/Atlanta/&', the capital of the South/' < $i.old > $i
done
```

---

## while and until loops

---

```
while condition
do
 statements
done

until condition
do
 statements
done
```

---

## case Statement

---

```
case $1 in
-f)
 ... Code for -f option
;;
-d | --directory) # long option allowed
 ... Code for -d option
;;
*)
 echo $1: unknown option >&2
 exit 1 # ;; is good form before `esac', but not required
esac
```

---

## Q: how does computer know how to split the list?

---

- **\$IFS** Internal field separator
- Define a list of characters the bash shell uses as field separators
- Default values: space, tab, newline
- Can change value of \$IFS to split list in different ways
- Better store original values and restore later

```
IFS.OLD=$IFS
IFS=$'\n'
<use the new IFS value in code>
IFS=$IFS.OLD
```

---

# Functions

---

- Must be defined before they can be used
- Can be done either at the top of a script or by having them in a separate file and source them with the "dot" (.) command.

---

## break and continue

---

- Pretty much the same as in C/C++
- Break: jump out of a loop
- Continue: jump to the beginning of a loop

---

## Example

---

```
wait_for_user --- wait for a user to log in
#
usage: wait_for_user user [sleeptime]
wait_for_user () {
 until who | grep "$1" > /dev/null
 do
 sleep ${2:-30}
 done
}

Functions are invoked the same way a command is
wait_for_user tolstoy Wait for tolstoy, check every 30 seconds
wait_for_user tolstoy 60 Wait for tolstoy, check every 60 seconds

The position parameters ($1, $2, etc) refer to the function's arguments.
The return command serves the same function as exit and works the same way
answer_the_question () {
 ...
 return 42
}
```