# CS 35L- Software Construction Laboratory

Fall 2018

TA: Guangyu Zhou

Lab 3

# 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

# Review

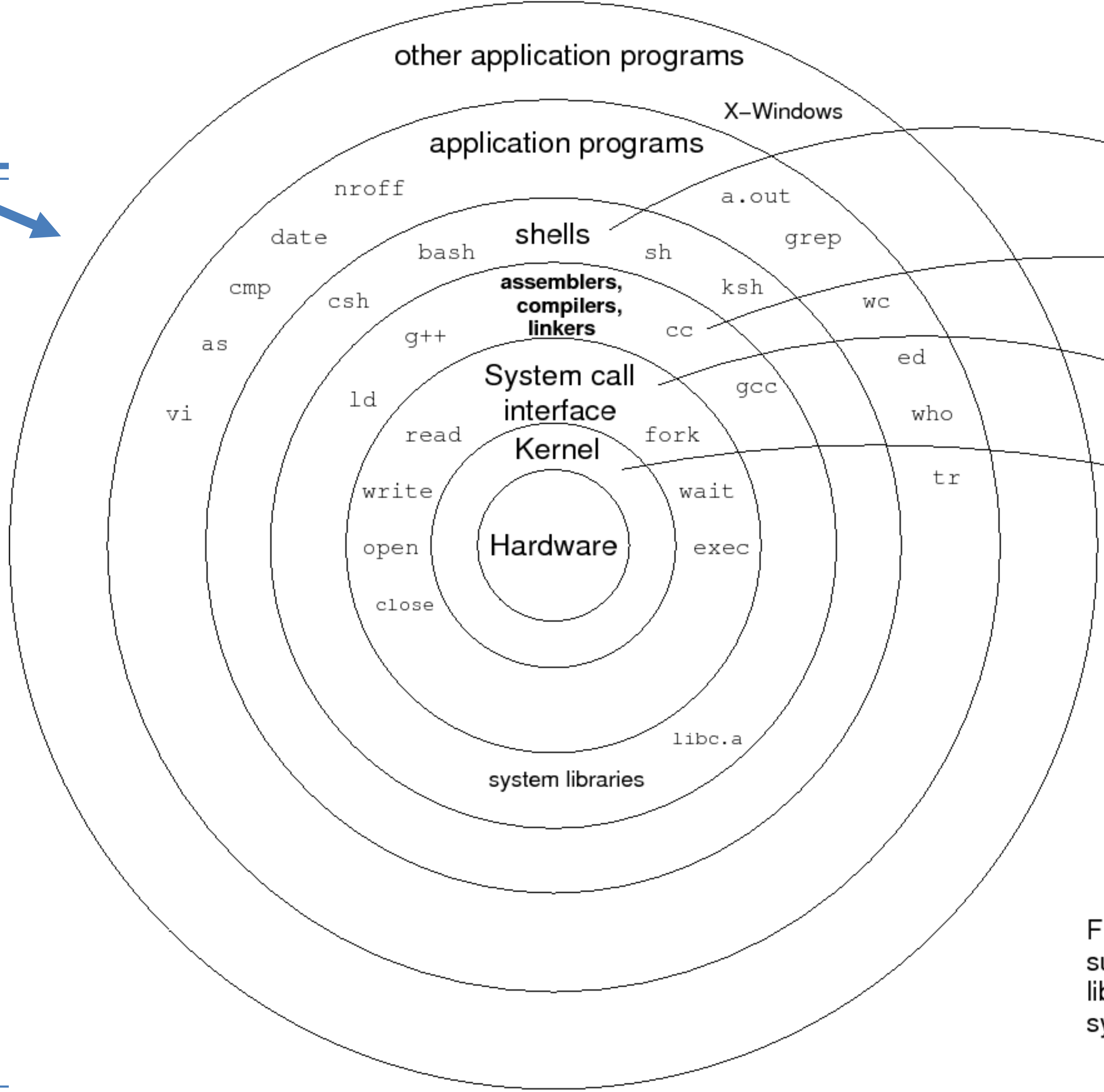
- **^' and '$'**
  - These symbols indicate the start and the end of a string, respectively.
- **'*', '+', and '?'**
  - The symbols '*', '+', and '?', denote the number of times a character or a sequence of characters may occur. What they mean is: "zero or more", "one or more", and "zero or one."
- **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:

# Outline

- Advanced Linux Commands
- Regular Expression
- **The Shell Scripting**

# 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 then interacting with OS directly
  - Tab complete, easy redirection



metacharacters, variables

creates virtual C computer

interface to core OS services

core of os

Filesystem and a suite of commands, libraries, and system calls

Conceptual Architecture of UNIX SYSTEMS

*From: The Design of the UNIX Operating System*

# 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)

# 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)

# 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.

# 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

# Example

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

**Script:**

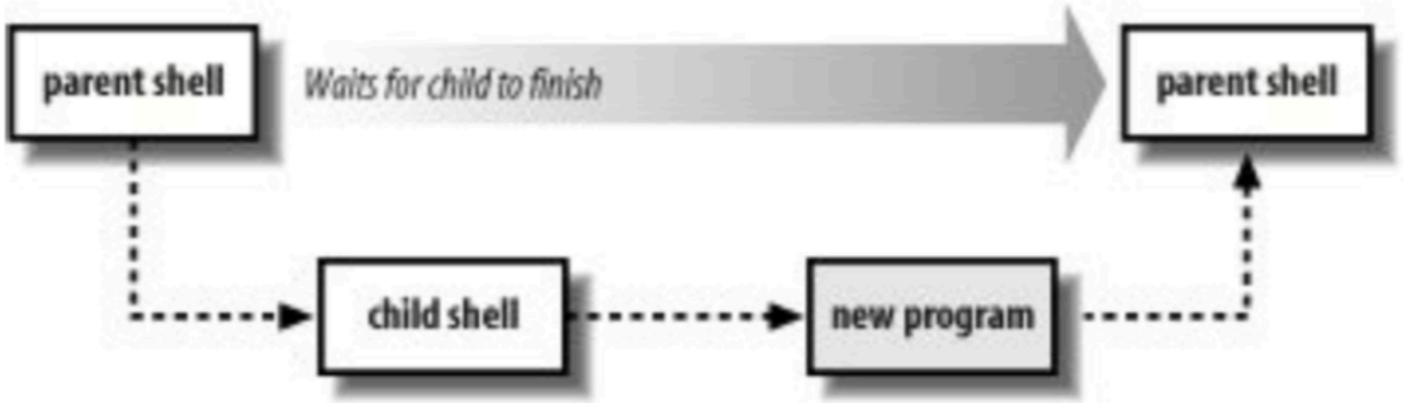
```
#! /bin/sh
# finduser --- see if user named by john is logged in
who | grep john
```

**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)
```

# 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

- 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

- 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

# 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

# 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

# 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)
```

# Exit: Return value

| Value | Meaning |
|---|---|
| 0 | Command exited successfully. |
| > 0 | Failure during redirection or word expansion (tilde, 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. |

# 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. |
| - (hyphen) | Options given to shell on invocation. |
| ? | Exit status of previous command. |
| $ | 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 full pathname for a file to be read and executed at startup. This is an XSI requirement. |
| HOME | Home (login) directory. |
| IFS | Internal field separator; i.e., the list of characters that act as word separators. Normally set to space, tab, and newline. |
| LANG | Default name of current locale; overridden by the other LC_* variables. |
| LC_ALL | Name of current locale; overrides LANG and the other LC_* variables. |
| LC_COLLATE | Name of current locale for character collation (sorting) purposes. |
| LC_CTYPE | Name of current locale for character class determination during pattern matching. |
| LC_MESSAGES | Name of current language for output messages. |
| LINENO | Line number in script or function of the line that just ran. |
| NLSPATH | 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. |

# 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

# 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 opera tor s | Right to left |

# 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`
```

# Structured command

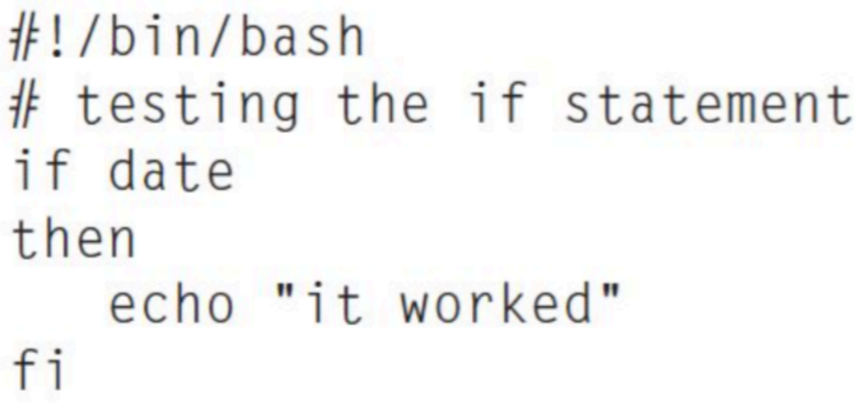
- Alter the flow of operations based conditions
- **If** statement
- **For** statement
- **While** loops
- **Case** statement
- **Break** statement
- **Continue** statement

# 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
```

# IF-THEN Statement

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

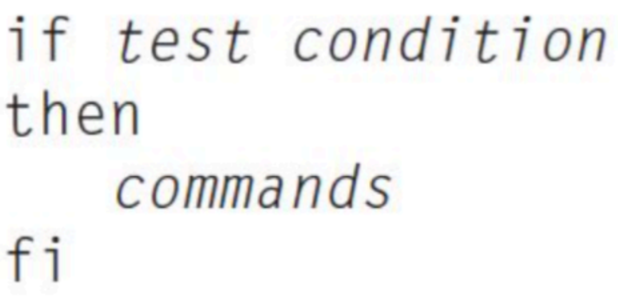


```
#!/bin/bash
# testing the if statement
if date
then
   echo "it worked"
fi
```

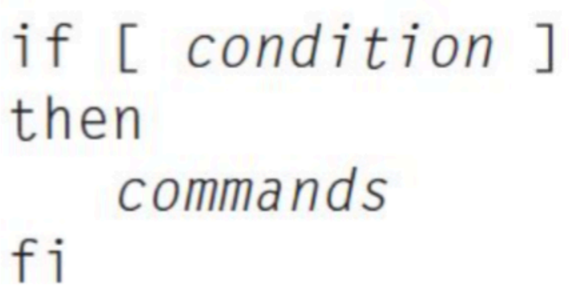
```
if grep pattern myfile > /dev/null
then
  ... Pattern is there
else
  ... Pattern is not there
fi
```

# Test command

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



```
if test condition
then
    commands
fi
```



```
if [ condition ]
then
    commands
fi
```

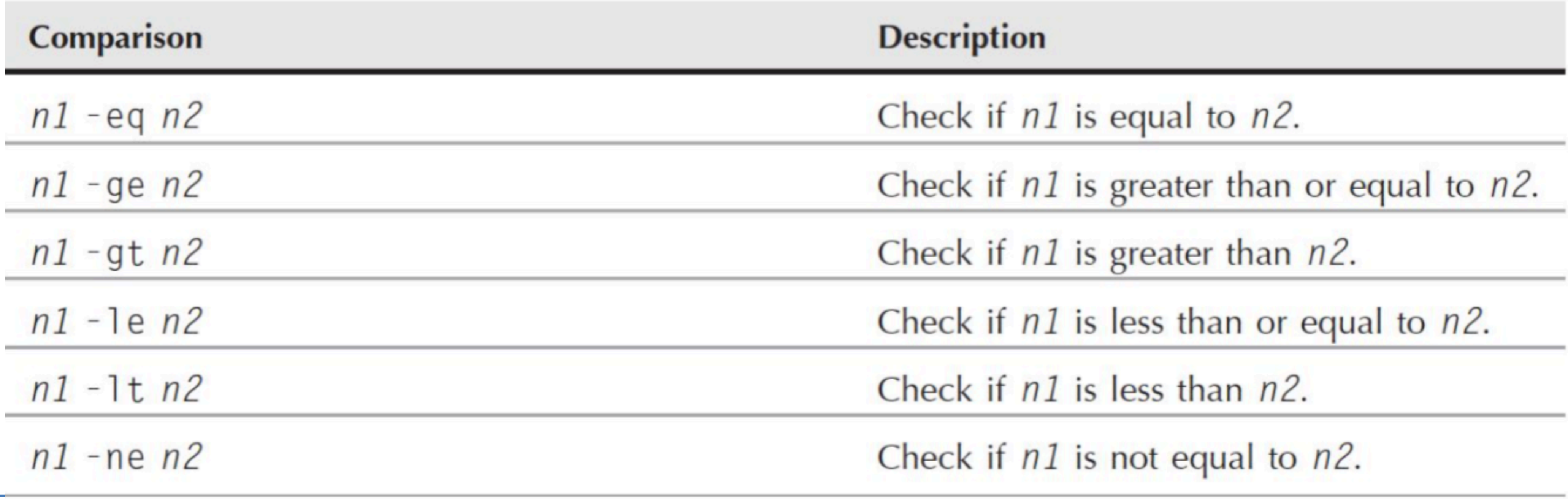
- If the condition listed in the test command is true, the test command exits with 0

# Test command

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

# Test command: Numeric comparisons

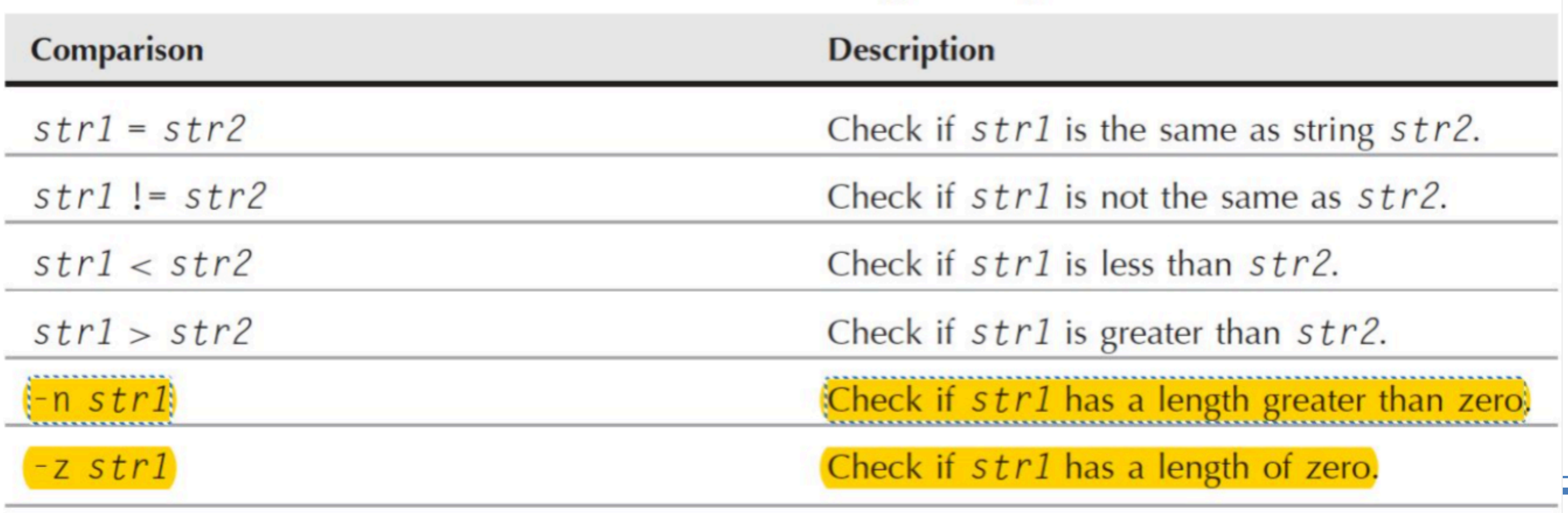
- Evaluate both numbers and variables
- - Ex: $var -eq 1; $var1 -ge $var2



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

# Test command: String comparisons

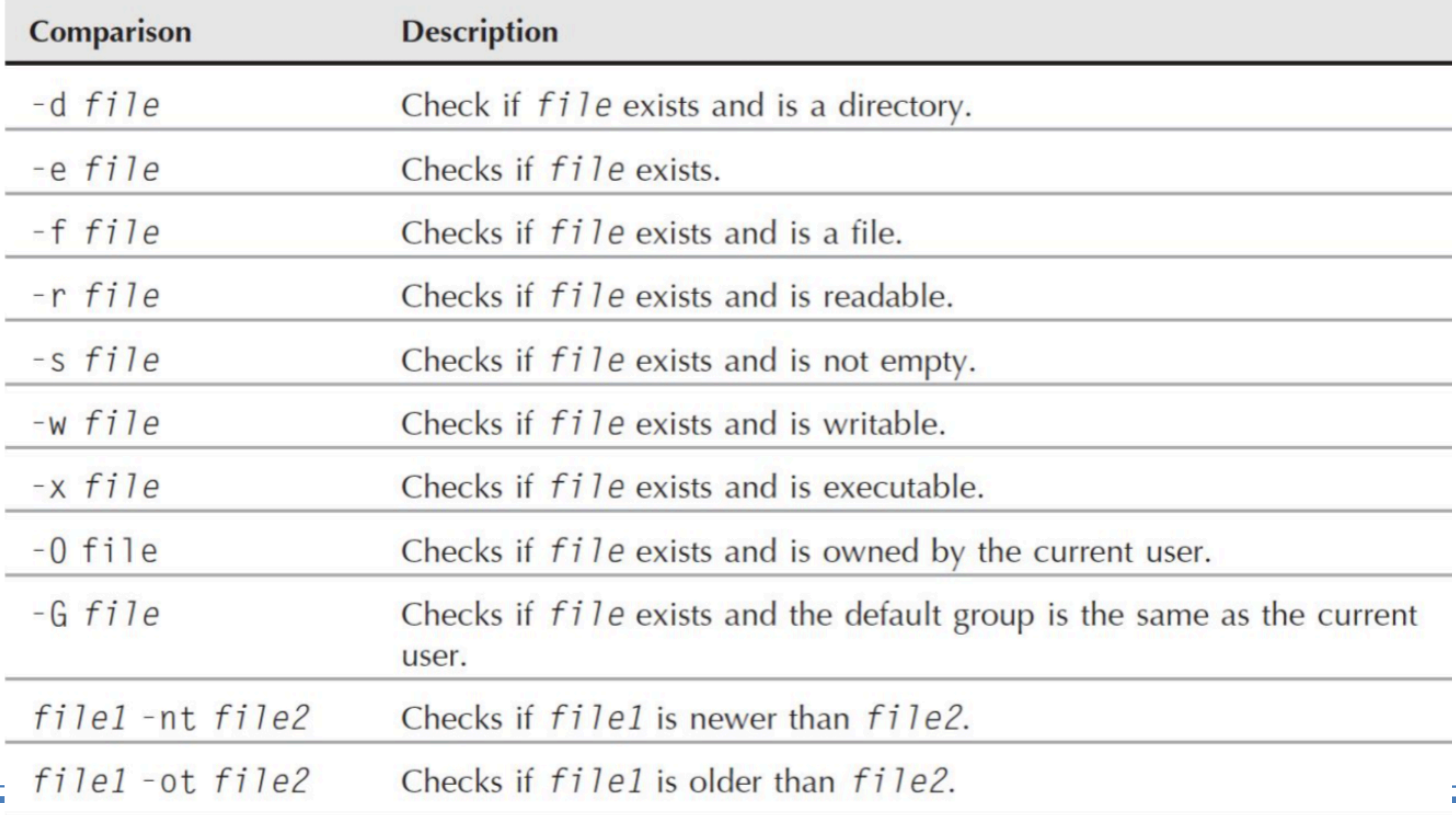
- - The greater-than and less-than symbols must be escaped (otherwise will be interpreted as redirection)
- - The greater-than and less-than order is not the same as sort (ASCII vs. locale language)
- - Ex: $USER = $testuser



| Comparison | Description |
|---|---|
| *str1* = *str2* | Check if *str1* is the same as string *str2*. |
| *str1* != *str2* | Check if *str1* is not the same as *str2*. |
| *str1* < *str2* | Check if *str1* is less than *str2*. |
| *str1* > *str2* | Check if *str1* is greater than *str2*. |
| -n *str1* | Check if *str1* has a length greater than zero. |
| -z *str1* | Check if *str1* has a length of zero. |

# Test command: File comparisons

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



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

# 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

# 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
```

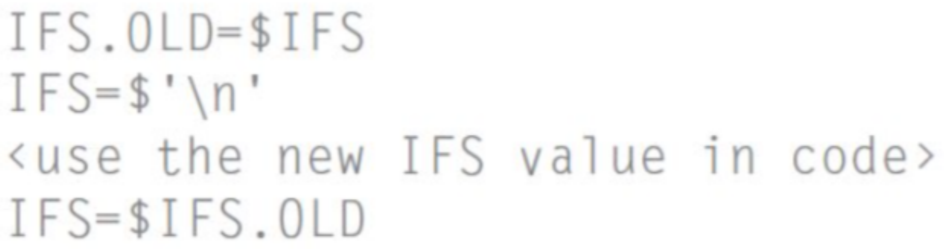
# for Loops

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

# 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
```

# while and until loops

```
while condition
do
    statements
done

until condition
do
    statements
done
```

# 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

# 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.

# 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
}
```

**/dev/null** is a special filesystem object that throws away everything written into it. Redirecting a stream into it means hiding an output.

**${parameter:-word}** If parameter is unset or null, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

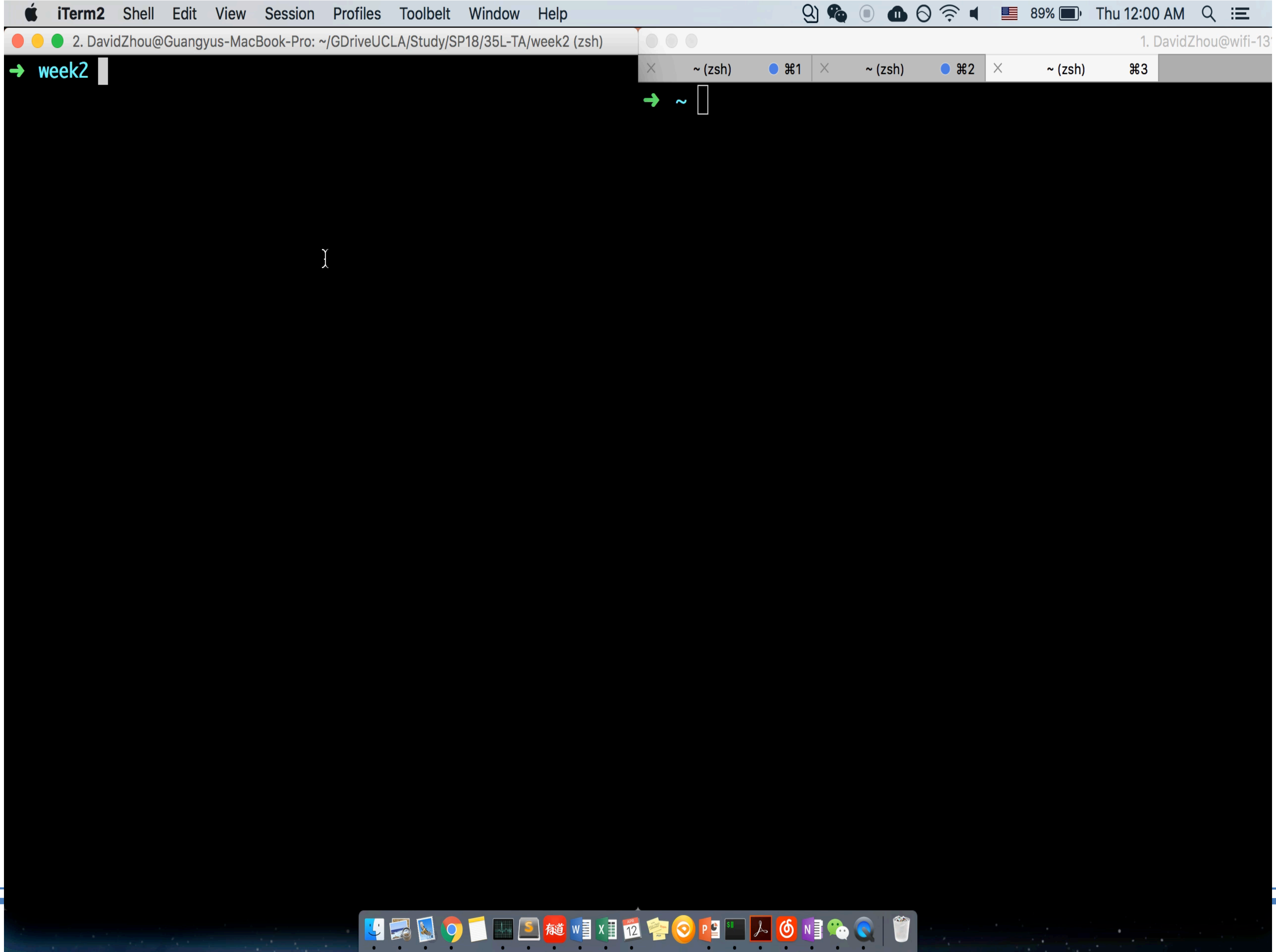
Functions are invoked the same way a command is

```
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
}
```

# Demo


iTerm2
Shell
Edit
View
Session
Profiles
Toolbelt
Window
Help
89%
Thu 12:00 AM
2. DavidZhou@Guangyus-MacBook-Pro: ~/GDriveUCLA/Study/SP18/35L-TA/week2 (zsh)
1. DavidZhou@wifi-13
~ (zsh) ⌘1
~ (zsh) ⌘2
~ (zsh) ⌘3

```
➜ week2
```

```
➜ ~
```

# For more information

- Supplement Materials: shell scripting tutorial

  http://tldp.org/LDP/Bash-Beginners-Guide/html/

- Classic Shell Scripting (only available via an UCLA IP address or UCLA VPN)

  http://proquest.safaribooksonline.com/0596005954

# Notice about Assignment 2

- Please refer to the instructions on Piazza
- Format and grading policy: inquire the TA for this week
- If there are any conflict between these hints and instructions on Piazza, follow the latter
- One of the hardest assignment. START EARLY!!!

# Laboratory -- Spell-checking Hawaiian

- More Hints:
- # replace a html tag </abcd> with new line character
  - 's/<\/abcd>/\n/g’
- #reject words without characters in a given dictionary(e.g. abcde)
  - '/[^a^b^c^d^e]/d’
- # replace + with ?
  - 's/\+/\?/g’
- # delete any html tags(surronded by '<>')
  - 's/<[^>]*>//g'

# Homework: find duplicate files

- Input argument: the path of directory
- Usage: ./sameln [directory name]
- Output: a list of regular files immediately under the given directory which have duplicates
- First line: **#!/bin/bash**
- Test with files that contain special characters
  - Spaces, *, leading “-”

# Homework: find duplicate files

How to check Hard Links

- Inode: data structure that stores information about files
  - File type
  - Permission
  - Owner
  - File Size, etc.
- Each inode is identified by a unique *inode number* within the file system
- Check a file’s inode number: ls –i filename
- How do you check if two files are hard-linked?
  - Same inode number

# Homework: find duplicate files

- Some tips
  - Only consider files immediately under given directory

    (hints: find -maxdepth 1 -type f)
  - For duplicates, keep the one whose name is lexicographically first, replace other files with hard links to the first one

    (hints: use ln command)
  - Don’t forget hidden files that begin with . !

    (hints: ls -a [directory] | grep '^\.')
  - Ignore non-regular and not readable files
  - File names may contain special characters (e.g. space, *, -)

# Week 2 Check List

- Unix wildcards, basic regular expressions
- More advanced commands (e.g., grep, find)
- Text editing tools (tr, sed)
- Pipelines and redirection
- Simple shell scripting