University
of Basel
Department of Mathematics and Computer Science

Distributed Programming and Internet (“DPI”)

Ali Ajorian (ali.ajorian@unibas.ch)

Shell Scripting

20. March 2025:

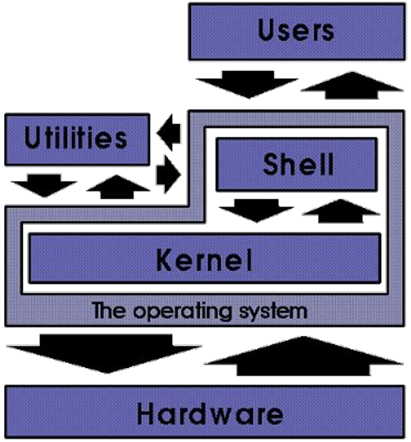
What is Shell

What is Shell and Shell scripting language

A shell scripting language is a scripting language for automating OS tasks.

- Similar to all other scripting languages it needs an interpreter

Shell is the outermost layer around OS and the interpreter of a shell language



http://ibgwww.colorado.edu/~lessem/psyc5112/usail/concepts/anatomy-of-unix/anatomy.html

How many interpreters for a language?

There are many interpreters for shell language. All are listed in a file located on /etc/shells:

- To see the list: `cat /etc/shells`

How many interpreters for a language?

There are many interpreters for shell language. All are listed in a file located on /etc/shells:

- To see the list: `cat /etc/shells`

Where each one is located:

- Use the command: `where` *`interpreter_name`*

How many interpreters for a language?

There are many interpreters for shell language. All are listed in a file located on /etc/shells:

- To see the list: `cat /etc/shells`

Where each one is located:

- Use the command: `where` *`interpreter_name`*

Some protocols provide remote access to these interpreters:

- Secure Shell Protocol (SSH)
- PowerShell Remote (on Windows)

Working modes of interpreters

Similar to Python interpreter, shell interpreters also have two working mode:

1. Interactive: interacts directly with the user

2. Non-interactive: executes commands from a script file without user interaction

- Create an empty file and write your scripts:
- Interpret the file: *interpreter_path script_path (Ex: /bin/bash ~/Desktop/a.sh)*

Working modes of interpreters

Similar to Python interpreter, shell interpreters also have two working mode:

1. Interactive: interacts directly with the user

2. Non-interactive: executes commands from a script file without user interaction

- Create an empty file and write your scripts:
- Interpret the file: *interpreter_path script_path (Ex: /bin/bash ~/Desktop/a.sh)*

You can call the default interpreter automatically:

- Change the permissions: `chmod +x hello.sh`
- Call the default interpreter: `./hello.sh`

Working modes of interpreters

Similar to Python interpreter, shell interpreters also have two working mode:

1. Interactive: interacts directly with the user

2. Non-interactive: executes commands from a script file without user interaction

- Create an empty file and write your scripts:
- Interpret the file: `interpreter_path script_path` (*Ex:* `/bin/bash ~/Desktop/a.sh`)

You can call the default interpreter automatically:

- Change the permissions: `chmod +x hello.sh`
- Call the default interpreter: `./hello.sh`

You can determine which interpreter interprets the file using shebang:

- *#! location_of_interpreter*

Instantiation of Shells

When a user logs in to the OS, it automatically opens a shells and reads several config files based on the OS and installed Shells

- Example: ~/.zprofile, ~/.zshrc, ~/.profile, ~/.login, ~/.bash_profile, ~/.bashrc

Multiple shells can be instantiated

- When you open a terminal you instantiate a new shell (ex Zsh, bash, etc.)
- Each instance has its own independent environment and life cycle
- Each instance reads these config files

Structure of commands

All command have the following structure: cmd arg_1 arg_2 … arg_n

- Example: find / -name “.profile” -print (The command is *find* with 4 arguments)

Structure of commands

All command have the following structure: cmd arg_1 arg_2 … arg_n

- Example: `find / -name “.profile” -print` (The command is *find* with 4 arguments)

What about *hi how are you*?

- The interpreter interprets it as the command *hi* with 3 arguments
- First it searches for *hi* in built-in command
- Then it searches in default paths

Structure of commands

All command have the following structure: cmd arg_1 arg_2 … arg_n

- Example: find / -name “.profile” -print (The command is *find* with 4 arguments)

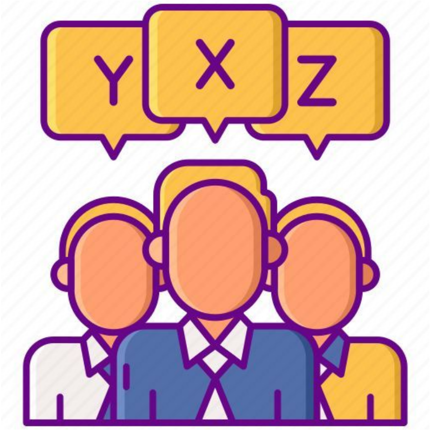
What about *hi how are you*?

- The interpreter interprets it as the command *hi* with 3 arguments
- First it searches for *hi* in built-in command
- Then it searches in default paths

Each command sets the exit status (0 as True and 1 as False)

- Something similar to *register flags* in assembly language
- To see the exit status use the command *echo $?*

Variables



Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
```

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
echo $ttt
```

If the variable is not declared the output is an empty string

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
y=$x+1
echo $y
```

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
y=$x+1
echo $y
```

In shell, everything is treated as a string by default, so the result of *echo $y* is 4+1 not 5.

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
y=$(expr $x + 1)
echo $y
```

1st solution: use *expr* command

- Arguments must be separated with space
- To replace the output of a command, use the command substitution syntax *$(...)*

Basic Assignment

Legacy command substitution syntax is: Backtick Strings

`y=`expr $x+1``

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
y=`expr $x + 1`
echo $y
```

1st solution: use *expr* command

- Arguments must be separated with space
- To replace the output of a command, use the command substitution syntax *$(...)*

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
((y = x+1))
echo $y
```

2nd solution: use *((*

- No need for space between arguments

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
y=$((x +1))
echo $y
```

3rd solution: use *((*

- No need for space between arguments
- Use the command substitution syntax *$(...),* you can omit the third parenthesis

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
let y=x+1
echo $y
```

4th solution: use *let*

- You can use it for arithmetic evaluation such as *let "a > 5"*

Basic Assignment

To declare: *variable_name=value*

- No space is allowed as any space acts a delimiter (invalid: x =4, x= 4 or x = 4)

To access: *$variable_name*

```
x=4
echo $x
declare -i y=x+1
echo $y
```

5th solution: declare y as integer

Declare keyword

declare allows you to create variables and set their attributes:

- Syntax: *declare options var_name*

option	meaning
-a	An index array
-i	An integer variable
-l	Lowercase string
-u	Uppercase string
-r	Read only variable
-x	Making variable available to child processes
-g	Global variable

Arrays

Syntax: *name[index]=value* or *name=(v1 v2 … vn)*

```
x[0]=1
x[1]=2
x[2]=3
echo ${x[*]}
```

```
x=(1 2 3)
x+=(4)
echo ${x[*]}
```

Arrays

Syntax: *name[index]=value* or *name=(v1 v2 … vn)*

```
x[0]=1
x[1]=2
x[2]=3
echo ${x[*]}
```

```
x=(1 2 3)
x+=(4)
echo ${x[*]}
```

- Accessing an element: *${array[index]}*
- Accessing all elements: *${array[*]}* (1 word : “arr[*]”)
- Accessing all elements: *${array[@]}* (list)
- Length of the array: *${#array[@]}*
- Appending an element: *array+=(element)*

Arrays (Initialization)

Initializing an array of integer numbers using brace expansion:

- Brace expansion is a feature that generates a series of strings
- Syntax: *{start..end}*
- Example: *echo {22..33}*

It can be used to initialize an array: *({start..end})*

```
x=({a..z})
y=(file{1..4}.txt)
echo ${x[@]}
echo ${y[@]}
```

Arrays (Initialization)

Initializing an array of integer numbers using *seq* command:

- seq generates a sequence of numbers
- Syntax: *seq [options] [first [step] last]*
- Example: *seq 1 9*

It can be used to initialize an array: *($(seq start step end))*

```
x=($(seq 100 -5 20))
echo ${x[2]}
```

List of Declared Variables

To show the list of all variables: *declare -p* or *typeset -p or set*

- *typeset* is an alias for the *declare* command

To delete a variable: *unset x*

PATH Variable

It is an environment variable holding a colon-separated list of directories of directories where the system searches for executable files when you run a command in the terminal.

- Initialized during the startup of a shell session.

PATH Variable

It is an environment variable holding a colon-separated list of directories of directories where the system searches for executable files when you run a command in the terminal.

- Initialized during the startup of a shell session.

You can edit configuration files that are automatically executed during boot time to add a directory to the PATH.

For example in file ~/.zshrc :

```
export PATH="/opt/homebrew/bin:$PATH"
```

PATH Variable

It is an environment variable holding a colon-separated list of directories of directories where the system searches for executable files when you run a command in the terminal.

- Initialized during the startup of a shell session.

You can edit configuration files that are automatically executed during boot time to add a directory to the PATH.

~/ is home directory of current user

For example in file ~/.zshrc :

```
export PATH="/opt/homebrew/bin:$PATH"
```

Strings

Different ways of representation:

1. *Unquoted strings*: spaces are delimiters, variables are expanded and special characters are interpreted.
2. *Single-Quoted strings*: are literal strings and variables and special characters are not expanded. *‘Hello $USER pwd’*
3. *Double-Quoted strings*: variables are expanded and special characters are interpreted while we can escape special characters using a backslash (\).

Input/Output

Output: *echo* Command

echo command: displays a line of text on standard output

- Syntax: *echo [options] [string …]*

```
echo -e "Hello\t $USER\nWaiting ..."
```

option	meaning
-n	Suppresses the trailing newline
-e	Enables interpretation of backslash

Output: *printf* Command

printf command: provides more control over formatting

- Syntax: *printf "format_string" [arguments...]*

```
printf "The user %s is %d years old\n" $USER 36
```

specifier	meaning
%s	String
%d	Decimal integer
%f	Floating-point number
%x	Hexadecimal number
%c	Character

Output: Redirection to a File

Output Redirection allows you to control where the output of a command goes.

Standard output redirection:

- `>`: overwriting the file if it already exists.
- `>>`: appending the output to the end of the file if it exists.a

Standard Error Redirection (2> and 2>>)

```
echo "Hello, World" > myfile.txt

ls abc 2> /dev/null
```

Output: Redirection to a File

Output Redirection allows you to control where the output of a command goes.

Standard output redirection:

- `>`: overwriting the file if it already exists.
- `>>`: appending the output to the end of the file if it exists.a

Standard Error Redirection (2> and 2>>)

```
echo "Hello, World" > myfile.txt

ls abc 2> /dev/null

date > a.txt 2>&1
```

To redirect **both** std output and std error use 2>&1 at the **end of the command**

Output: Redirecting to a Command with Pipes

A *pipe* redirects stdout of a command into the stdin of another command

- Arguments of the right command are specified separately, not from the pipe
- Syntax: Command$_1$ | Command$_2$| … | Command$_n$

```
cd ~/
ls | grep "Desk*"
```

Grep (Global Regular Expression Print)

grep is a powerful text-search tool for files or input from standard input

Syntax: *grep [options] pattern [file...]*

- Without a file argument it reads from stdin

```
cd ~/
ls | grep "Desk*"
```

option	meaning
-i	Ignore case sensitive
-v	Show lines that don’t match
-r	Recursive search in directories
-l	List only the name of files
-n	Show line numbers in files
-c	Count the number of matchings
-q	Quiet mode (just sets exit status)

Input: Read Command

read command stores the input into variables

- Syntax: *read [options] variable_names*

```
#!/bin/bash

read -p "Enter your name: " name
read -p "Enter your age: " age
read -a hobbies -p "Enter your hobbies (space-separated): "
echo "Name: $name"
echo "Age: $age"
echo "Hobbies: ${hobbies[@]}"
read -s -p "Enter your password: " password
echo -e "\nPassword entered."
```

option	meaning
-p	Displays a prompt message
-a	Reads an array
-n	Reads only n number of chars
-s	Silent mode (for reading passwords)
-t *timeout*	Stops after *timeout* seconds
-d *delimiter*	Specifies delimiter other than space

Input: Redirection from a File

Input redirection allows a command to read from a file instead of standard input

- Syntax: *command < filename*

```
echo -e "Hello, how are you?\nHow is your project going?"  > a.txt

grep -c -i "how" < a.txt
```

Input: Command-line Arguments

Command-line arguments are values provided to a script or command when it is executed

```
#! /bin/bash
name=$0
cnt=$#
arr=($@)
printf "$name has $cnt arguments\nThe array is ( ${arr[*]} )\n"
```

specifier	meaning
$0	Name of the script
$1, $2, ...	ith argument passed to the script
$#	The number of arguments
$@	All arguments as separate words
$*	All arguments as single word

Input: Here String

In shell, many commands read data from files. *Here String* allows you to pass a string **directly** to the standard input (stdin) of a command without needing to create a temporary file.

- Syntax: *command* <<< *string*

```
cat <<< "This is a text"
result=$(wc -w <<< "This is a sample sentence.")
echo "Word count: $result"
```

Input: Here String and Here Document

In shell, many commands read data from files. *Here Document* allows you to pass a block of text **directly** to the standard input (stdin) of a command without needing to create a temporary file.

- Syntax: *command << delimiter block of text delimiter*

```
name="Alice"
cat << EOF
Hello, $name!
Welcome to the here document example.
EOF
```

Conditional Statements

test Command

test evaluates a conditional expression and sets exit status

- Syntax: *test expr* or *[expr]* (paces are important)

test Command

test evaluates a conditional expression and sets exit status

- Syntax: *test expr* or *[expr]* (paces are important)

```
test "abcd" = "abcd"
echo $?
```

test Command

test evaluates a conditional expression and sets exit status

- Syntax: *test expr* or *[expr]* (paces are important)

```
test "abcd" = "abcd"
echo $?
```

Possible expressions		
-d file	is a directory	[-d /tmp]
-f file	is a regular file	[-f ./a.txt]
-e file	exists	[-e ./a.txt]
-x file	executable	[-x ./cmd.sh]
-z string	empty string	[-z “$var_name”]
-n string	non-empty string	[-n “$var_name”]
string1 = != < > string2	Comparing strings	["$s1" > "$s2"]
int1 -eq -ne -lt -gt -le -ge int2	Comparing integers	[5 -ge 3]
! expr	Boolean not	[! -f ./a.txt]
-a expr	Boolean and	[-z $var -a -f a.txt]
-o expr	Boolean or	[-z $var -o -f a.txt]

test Command

test evaluates a conditional expression and sets exit status

- Syntax: *test expr* or *[expr]* (paces are important)

```
[ "abcd" = "abcd" ] && [ -d /tmp ]
echo $?
```

&& and || can be used to combine two tests

Possible expressions		
-d file	is a directory	[-d /tmp]
-f file	is a regular file	[-f ./a.txt]
-e file	exists	[-e ./a.txt]
-x file	executable	[-x ./cmd.sh]
-z string	empty string	[-z “$var_name”]
-n string	non-empty string	[-n “$var_name”]
string1 = != < > string2	Comparing strings	["$s1" > "$s2"]
int1 -eq -ne -lt -gt -le -ge int2	Comparing integers	[5 -ge 3]
! expr	Boolean not	[! -f ./a.txt]
-a expr	Boolean and	[-z $var -a -f a.txt]
-o expr	Boolean or	[-z $var -o -f a.txt]

test Command

test evaluates a conditional expression and sets exit status

- Syntax: *test expr* or *[expr]* (paces are important)

```
[[ "abcd" = "abcd" && -d /tmp ]]
echo $?
```

[[syntax is more powerful and supports && and || and regular expressions

Possible expressions		
-d file	is a directory	[-d /tmp]
-f file	is a regular file	[-f ./a.txt]
-e file	exists	[-e ./a.txt]
-x file	executable	[-x ./cmd.sh]
-z string	empty string	[-z “$var_name”]
-n string	non-empty string	[-n “$var_name”]
string1 = != < > string2	Comparing strings	["$s1" > "$s2"]
int1 -eq -ne -lt -gt -le -ge int2	Comparing integers	[5 -ge 3]
! expr	Boolean not	[! -f ./a.txt]
-a expr	Boolean and	[-z $var -a -f a.txt]
-o expr	Boolean or	[-z $var -o -f a.txt]

if Command

if is a control structure that allows you to execute code conditionally

Syntax:

```
if condition ; then
  # commands
elif
 # commands
else
 # commands
fi
```

if Command

if is a control structure that allows you to execute code conditionally

Syntax:

```
if condition ; then
  # commands
elif
 # commands
else
 # commands
fi
```

condition is usually a test command (e.g., [ ] or [[ ]]) or any command with an exit status.

if Command

if is a control structure that allows you to execute code conditionally

Syntax:

```
if condition ; then
  # commands
elif
  # commands
else
  # commands
fi
```

condition is usually a test command (e.g., [ ] or [[ ]]) or any command with an exit status.

```
if [ -e a.txt ]; then
  echo "The file exists"
fi
```

if Command

if is a control structure that allows you to execute code conditionally

Syntax:

```
if condition ; then
  # commands
elif
 # commands
else
 # commands
fi
```

condition is usually a test command (e.g., [ ] or [[ ]]) or any command with an exit status.

```
if [ 5 -lt 3 ]; then
  echo "True"
else
  echo "False"
fi
```

if Command

if is a control structure that allows you to execute code conditionally

Syntax:

```
if condition ; then
  # commands
elif
 # commands
else
 # commands
fi
```

condition is usually a test command (e.g., [ ] or [[ ]]) or any command with an exit status.

```
if (( 2 + 3 == 5 )); then
  echo "True"
fi
```

if Command

if is a control structure that allows you to execute code conditionally

Syntax:

```
if condition ; then
  # commands
elif
 # commands
else
 # commands
fi
```

condition is usually a test command (e.g., [ ] or [[ ]]) or any command with an exit status.

```
if grep -q -i "error" file.txt; then
  echo "Failed"
fi
```

Loop Statements

Loops

Loop statements are control structures that execute a block of code repeatedly based on a condition or a set of items

Bash supports three main types of loops:

- for loop
- while loop
- until loop

for Loops

Iterates over a list of items (e.g., words, numbers, files)

Syntax:

```
for var in list ; do
        # commands
done
```

It assigns *var* to each item in the *list* one by one and runs the code block.

```
for name in alice bob charlie; do
  echo "Hello, $name"
done
```

for Loops

Iterates over a list of items (e.g., words, numbers, files)

Syntax:

```
for var in list ; do
        # commands
done
```

It assigns *var* to each item in the *list* one by one and runs the code block.

- You can use various syntaxes or commands to generate a list

```
n=10
for i in $(seq 1 $n); do
  echo $i
done
```

```
n=10
for i in {1..$n}; do
  echo $i
done
```

for Loops

Iterates over a list of items (e.g., words, numbers, files)

Syntax:

```
for var in list ; do
        # commands
done
```

It assigns *var* to each item in the *list* one by one and runs the code block.

- You can use various syntaxes or commands to generate a list

```
for ((i=1; i<=3; i++)); do
  echo "Count: $i"
done
```

C-style loops

while Loops

Repeats as long as a condition is true (exit status 0).

Syntax:

```
while condition ; do
        # commands
done
```

Tests the condition before each iteration and stops when it’s false

```
i=3
while [ $i -gt 0 ]; do
  echo $i
  ((i=i-1))
done
```

```
while read line; do
  echo "Line: $line"
done < file.txt
```

```
while grep -q "error" log.txt; do
  echo "Error still present"
done
```

until Loops

Repeats as long as a condition is false (exit status 0).

Syntax:

```
until condition ; do
        # commands
done
```

Tests the condition before each iteration and stops when it’s true

```
i=0
until [ $i -eq 3 ]; do
  echo "Count: $i"
  i=$((i+1))
done
```

break and *continue*

break: Exits the loop early

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then
     break
  fi
  echo "Number: $i"
done
```

⇨

```
for i in {1..5}; do
  [ $i -eq 3 ] && break
  echo "Number: $i"
done
```

break and *continue*

break: Exits the loop early

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then
     break
  fi
  echo "Number: $i"
done
```

⇨

```
for i in {1..5}; do
  [ $i -eq 3 ] && break
  echo "Number: $i"
done
```

continue: Skips the rest of the current iteration

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then
     continue
  fi
  echo "Number: $i"
done
```

⇨

```
for i in {1..5}; do
  [ $i -eq 3 ] && continue
  echo "Number: $i"
done
```

Functions

Declaration and Invocation

There are two ways to declare a function, with and without keyword *function*

- No need to declare arguments
- No return values,they use return for exit status

```
name() {
  # Commands
}
```

```
function name() {
  # Commands
}
```

Declaration and Invocation

There are two ways to declare a function, with and without keyword *function*

- No need to declare arguments
- No return values,they use return for exit status

```
name() {
  # Commands
}
```

```
function name() {
  # Commands
}
```

To access arguments: $1, $2, …

- $#: number of arguments
- $@ or $*: all arguments as list

To call a function: *name arg1 arg2 … argn*

Declaration and Invocation

There are two ways to declare a function, with and without keyword *function*

- No need to declare arguments
- No return values,they use return for exit status

```
name() {
  # Commands
}
```

```
function name() {
  # Commands
}
```

To access arguments: $1, $2, …

- $#: number of arguments
- $@ or $*: all arguments as list

To call a function: *name arg1 arg2 … argn*

```
function greeting(){
  name=$1
  echo Hello $name
}

greeting "Erick"
```

Declaration and Invocation

There are two ways to declare a function, with and without keyword *function*

- No need to declare arguments
- No return values,they use return for exit status

```
name() {
  # Commands
}
```

```
function name() {
  # Commands
}
```

To access arguments: $1, $2, …

- $#: number of arguments
- $@ or $*: all arguments as list

To call a function: *name arg1 arg2 … argn*

```
factorial() {
  if [ "$1" -le 1 ]; then
    echo 1
  else
    local n=$1
    echo $(( n * $(factorial $((n-1))) ))
  fi
}
factorial 4
```

Use *local* keyword for declaring local variables