Linux Admin

Class 3 - Shell Scripting

By Ian Robert Blair, M.Sc.

Agenda

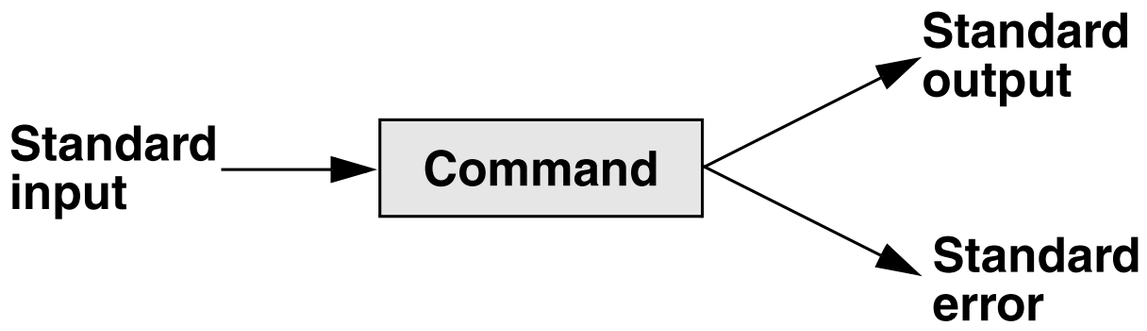
- Using the Command Line
- Shell Scripting
- Processes

The Command Line 1

- **Standard input** is a place a program gets information from (default the keyboard)

- **Standard output** is a place to which a program can send information (default screen, but could be redirected to a printer or an ordinary file)

- **Standard error** (default the screen)



The Command Line 2

- A **file descriptor** is the place a program sends its output to and gets its input from
- When you execute a program, Linux opens three file descriptors for the program: **0 (standard input), 1 (standard output), and 2 (standard error)**
- The redirect output symbol > is shorthand for **1>**, which tells the shell to redirect standard output
- Similarly < is short for **0<**, which redirects standard input
- The symbol **2>** redirects standard error
- The **&>** token redirects standard output and standard error to a single file
- **1>&2** to redirect standard output of a command to standard error and **2>&1** redirects standard error to output

The Command Line 3

- The redirect **output symbol (>)** instructs the shell to redirect the output of a command to the specified file

  command [arguments] > filename

  date > whoson1 ; who > whoson2

- The redirect **input symbol (<)** redirects a command's input to come from the specified file

- command [arguments] < filename

  cat < whoson1 ; cat < whoson2

- The **append output symbol (>>)** causes the shell to add new information to the end of a file

- date > whoson; who >> whoson

The Command Line 4

- A **pipe** connects standard output of one command to standard input of another command

  command_a [arguments] | *command_b [arguments]*

- A **filter** is a command that processes an input stream of data to produce an output stream of data, command_b is a filter

  command_a [arguments] | *command_b [arguments]* | *command_c [arguments]*

- The **tee** utility copies its standard input both to a file and to standard output

  command_a [arguments] | *tee filename*

  cat /etc/passwd | *grep /bin/bash* | *tee users.txt*

- Output you do not want to keep or see can be redirected to **/dev/null,** and the output will disappear

Filters and Pipes

- The **sort** utility takes its input from the file specified on the command line or, when a file is not specified, from standard input; it sends its output to standard output.

- The **grep** utility displays the line containing the string you specify
    - cat /etc/passwd | grep "/bin/bash" | cut -d ":" -f 1 | sort

- **sed** performs text editing on a stream of text, either a set of specified files or standard input
    - cat Logfile.txt | sed 's/system1.c.net/host1.lab.net/'

Foreground/Background Jobs 1

- A command, program, or script running in the **foreground** must finish before displaying another prompt
- However, when run in the **background** it allows use of the shell while running
- You can have only **one** foreground job but you can have many background jobs
- To run a command in the background, type an **ampersand (&)** just before the RETURN that ends the command line
- You can also suspend a foreground job (stop it from running) by pressing **CONTROL-Z**
- The job can be run in the background by restarting it using the **bg** command

Foreground/Background Jobs 2

- Type **fg** or a percent sign (%), followed by the number of the job to bring it into the foreground

- To stop a job use **kill** on the command line with either the PID number of the process you want to abort or a percent sign (%) followed by the job number

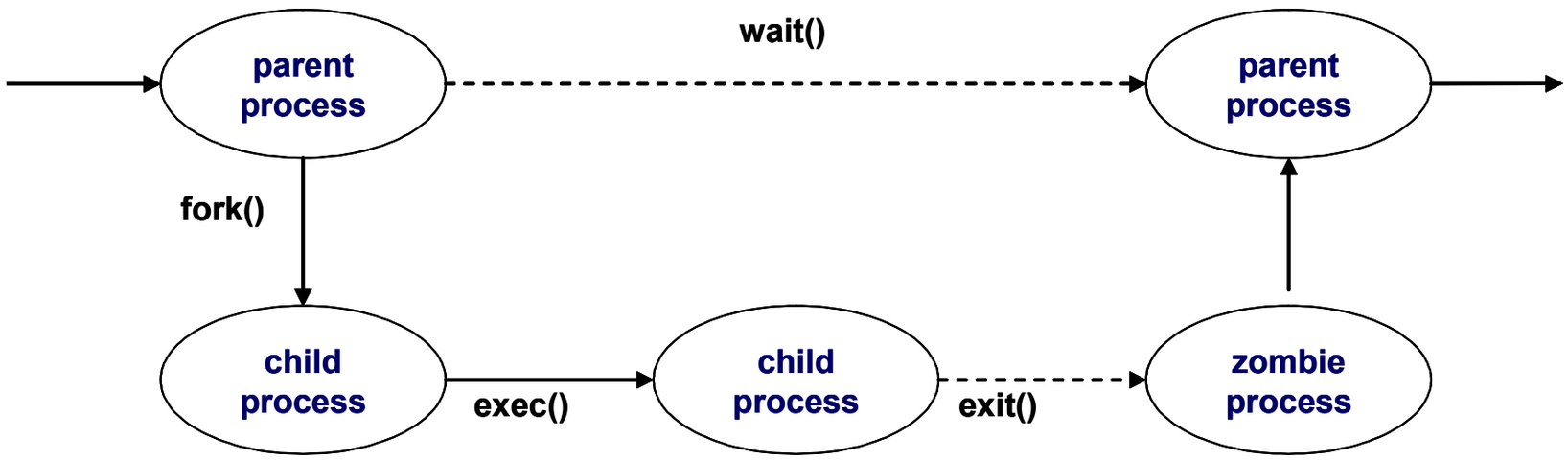
- The **ps (process status)** utility is used to display process IDs

- The **jobs** command to display a list of job numbers

- To enable the process will continue after the shell is closed use **nohup**

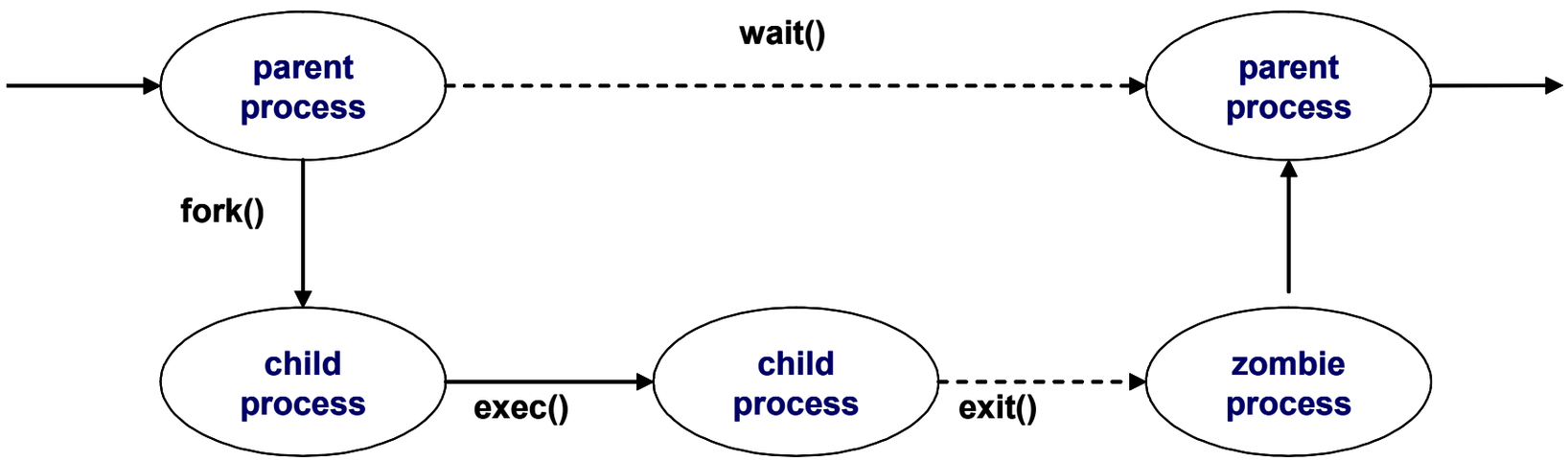
Process 1

- A parent process **forks (or spawns)** a child process, which in turn can fork other processes
- Any **program** run on the command line forks a new process
- Linux assigns a unique PID (process identification) number at the inception of each process (pstree -p)



Process 2

- While the child process is executing the command, the parent process **sleeps**
- When the child process finishes executing the command, it tells its parent of its success or failure via its **exit status** and then dies
- **Background** processes run as child process without the parent going to sleep



Additional Utilities 1

- You can also use the **pgrep** utility to list all processes owned by a user
  - pgrep -u sam -l calc
- The **pkill** utility, which has a syntax similar to pgrep, kills a process based on the command line that initiated the process
  - pkill -u sam bash
- The **kill** builtin sends a signal to a process
  - kill or kill –TERM or kill –15 (used first)
  - kill –KILL or kill – 9 (if 15 fails)
- The **killall** utility is similar to kill (previous) but uses a command name instead of a PID number
  - killall gnome-calculator vi

Additional Utilities, pt. 2

- To Kill all processes own by a user:

- su zach -c 'kill -TERM -1'

Bash Scripts

- After login, the shell first executes the commands in **/etc/profile** or the *.sh files in **/etc/profile.d**
- Then, the shell looks for **~/.bash_profile, ~/.bash_login, or ~/.profile,** in that order, executing the commands in the first of these files it finds
- On log out, bash executes commands in the **~/.bash_logout** file
- An **interactive nonlogin** shell executes commands in the **~/.bashrc** file
- The default **~/.bashrc** file calls **/etc/bashrc**

Shell Script

```
#!/bin/bash
#Author: Ian Robert Blair (ianrobertblair@icloud.com)
#Date: 2013.2.14
#Name: Script 1
#Description: Demonstrate Bash Scripting
#Version: 1.1

echo ------   Find Large Files  Script  ----------
echo
echo The following files are found to be over the 500MB limit
echo Please contact users \
and move them to secondary storage - \
for more info email: admin@company.net.
echo
find /home -type f -size +500000k -exec ls -lh {} \; | awk '{ print $9 ": " $5 }'
```

Variables 1

- To assign a value to a variable in the Bourne Again Shell, use the following syntax(no whitespaces):

  VARIABLE=value

- The shell substitutes the value of a variable only when you precede the name of the variable with a dollar sign ($)

  echo $var1

- To prevent the substitution use **single** quotation marks or a backslash **(\)**

  echo ‘$var1’ or echo \$var1

- To assign a value that contains SPACEs or TABs to a variable, use double quotation marks around the value

  users=”nancy and ann”; echo $users

Variables 2

- Braces insulate the variable name from adjacent characters

  echo ${admin}
- To make a variable read-only

  readonly user
- You can remove a variable using the **unset** builtin

  unset user
- To make a variable available to another shell

  export variable
- **Declare** and **typeset** assign attributes to variables
- Attributes: -r (read only), -x (export), -a (array), -i (integer), -f (function name)

  declare -rx user=ian; declare -ix group=454

Alias

- An **alias** is a (usually short) name that the shell translates into another (usually longer) name or (complex) command

- They are typically placed in the **~/.bashrc** startup files so that they are available to interactive subshells

- The syntax of the alias builtin is:
  - alias [name[=value]]
  - $ alias ls='ls -lh'

Functions

- A **shell function** is similar to a shell script in that it stores a series of commands for execution at a later time

- Stored in RAM (fast)

- Declare a shell function in the **~/.bash_profile** startup file, in the script that uses it, or directly from the command line

- You can remove functions with the **unset** builtin, and shell does not retain functions after you log out

Command-Line Expansion

- The shell performs **arithmetic expansion** by evaluating an arithmetic expression and replacing it with the result, $((expression))

  echo age?; read age;

  echo "Wow, in $((60-age)) years, you'll be 60!"

- The **let** builtin evaluates arithmetic expressions just as the $(( )) syntax does

  $ let a=5+3 b=7+2

Operators

Parameter	Definition
var--, var++	post increment
--var, ++var	pre increment
-var, +var	unary minus, plus
!	Boolean negation
**	Exponent
*, /, %	multiplication, division, remainder (modulus)
+, -	addition, subtraction
<=, >=, <, >	less than or equal, greater than or equal, less than, greater
==, !=	Equal, not equl
&&, ||	Boolean and, boolean or
exp1?exp2:exp3	Ternary operator
=, *=, /=, %=, +=, -=	Assignment

Built-in Variables

Variable	Contents
DISPLAY	Then name of your display if you are running a graphical env
EDITOR	Program used for text editing
SHELL	Shell
HOME	Path of your home directory
LANG	Character set and collation
OLD_PWD	Previous working directory
PAGER	Program for paping (i.e. less)
PATH	Colon-separated list of directories that are searched when enter the name of an executable program
PS1	Setting for shell prompt
PWD	Current working directory
TERM	Terminal type
USER	Username

Command Substitution

- **Command substitution** replaces a command with the output of that command, **$(command)** or the older **`command`**

- The shell executes command within a sub-shell and replaces command with the standard output of command

- $ ls -l $(find . -name README -print)

- $ ls -l `find . -name README -print`

Test 1

- **Test** is a builtin—part of the shell
- Syntax 1: test *equation*
- Syntax 2: [*equation*]
    - *Note there must be a space or tab between the equation and the bracket
- Returns **0 if true** and **not 0 if false**

Testing Files

Option	Usage
-d	Exists as a directory
-e	Exists
-f	Exists as an ordinary file
-r	Exists and is readable
-s	Exists and is greater than 0
-w	Exists and is writable
-x	Exists and is executable

Testing Integers

Option	Usage
-eq	Equal
-ne	Not Equal
-le	Less than or equal
-lt	Less than
-ge	Greater than or equal
-gt	Greater than

Testing Strings

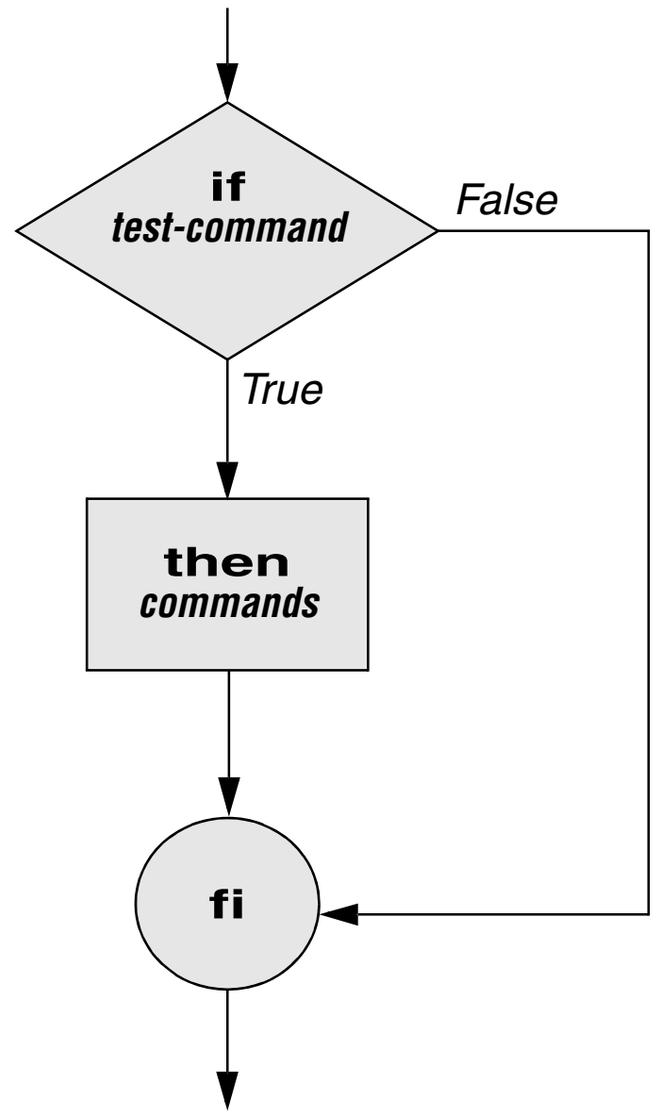
Option	Usage
-n	Length greater than zero
-z	Length is zero
s == s	Equal
!=	Not Equal

Control Structure: if

if *test-command*
then
commands
fi

Example:

```
if test $# -eq 0
    then
        echo " You must supply at least one arg."
        exit 1
fi
```

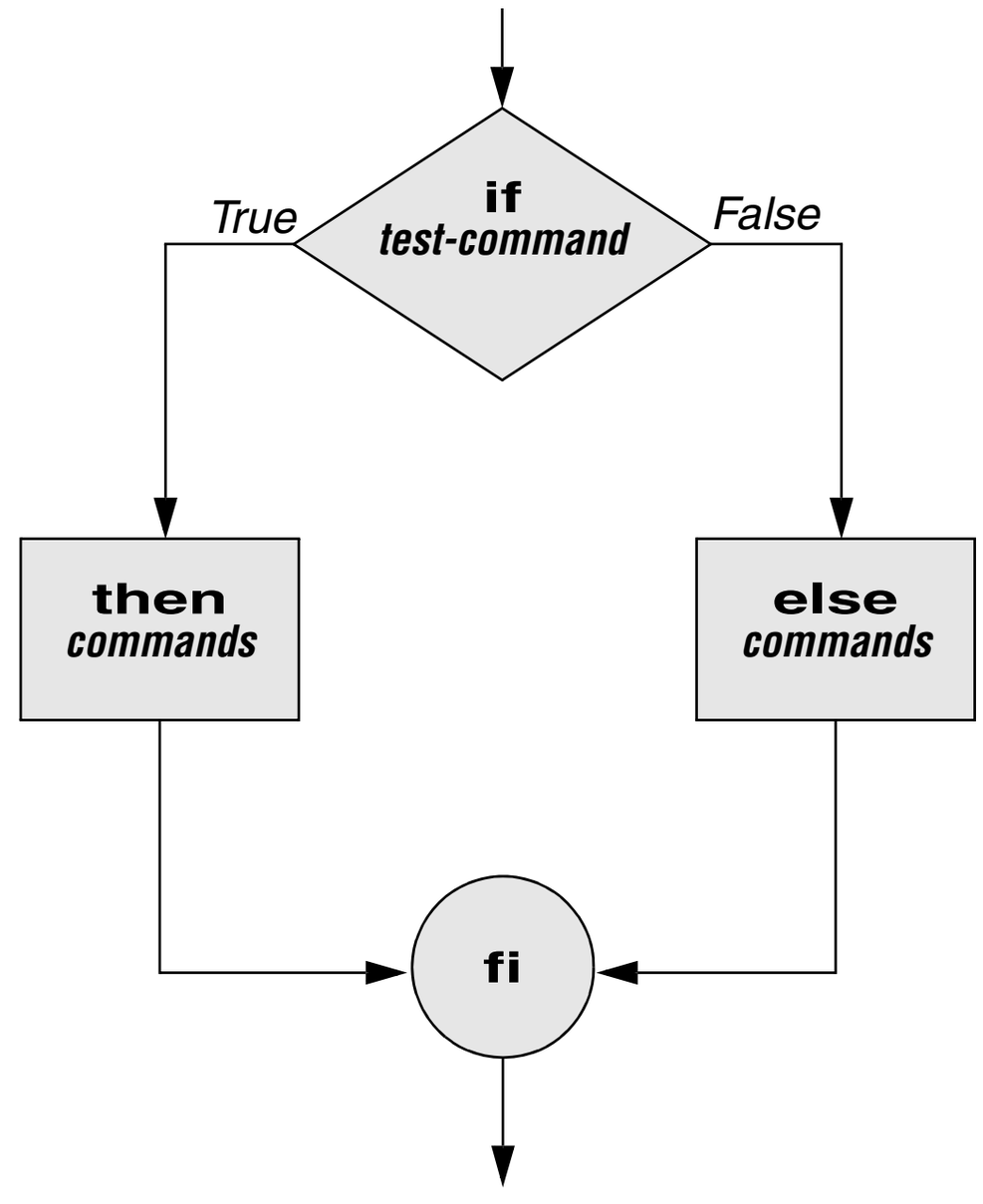


Control Structure: if-then-elif

If *test-command*
then
commands
else
commands
fi

Example:

```
if [ "$1" = "-v" ]
    then
shift
       less -- "$@"
    else
        cat -- “$@”
fi
```

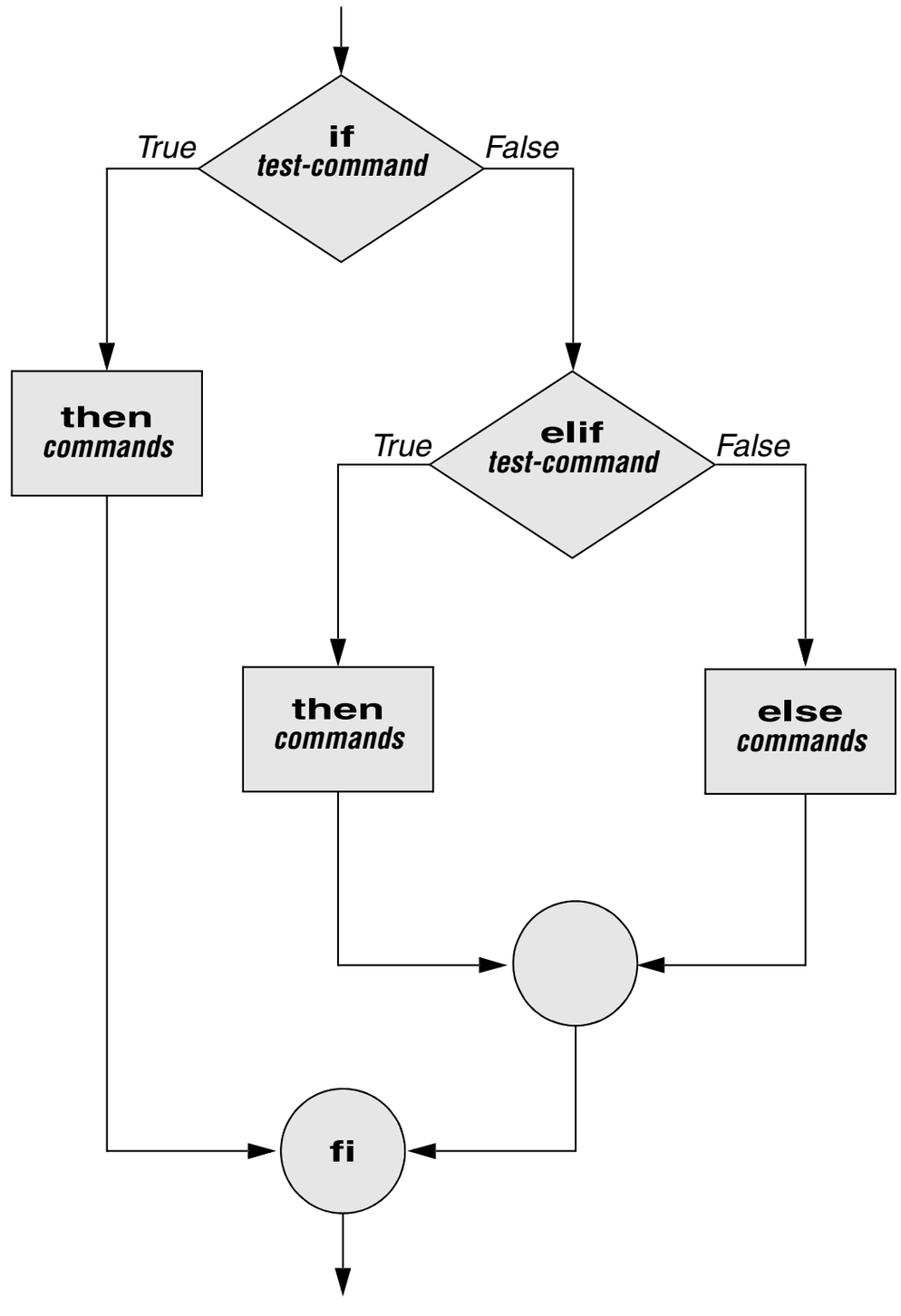


Control Structure: if-then-elif

```
if test-command
        then
                commands
        elif test-command
                then
                        commands
        else
                commands
fi
```

Example:

```
if [ "$word1" = "$word2" -a "$word2" = "$word3" ]
        then
           echo "Match: words 1, 2, & 3"
        elif [ "$word1" = "$word2" ]
        then
           echo "Match: words 1 & 2"
        elif [ "$word1" = "$word3" ]
        then
           echo "Match: words 1 & 3"
        elif [ "$word2" = "$word3" ]
        then
           echo "Match: words 2 & 3"
        else
           echo "No match"
fi
```

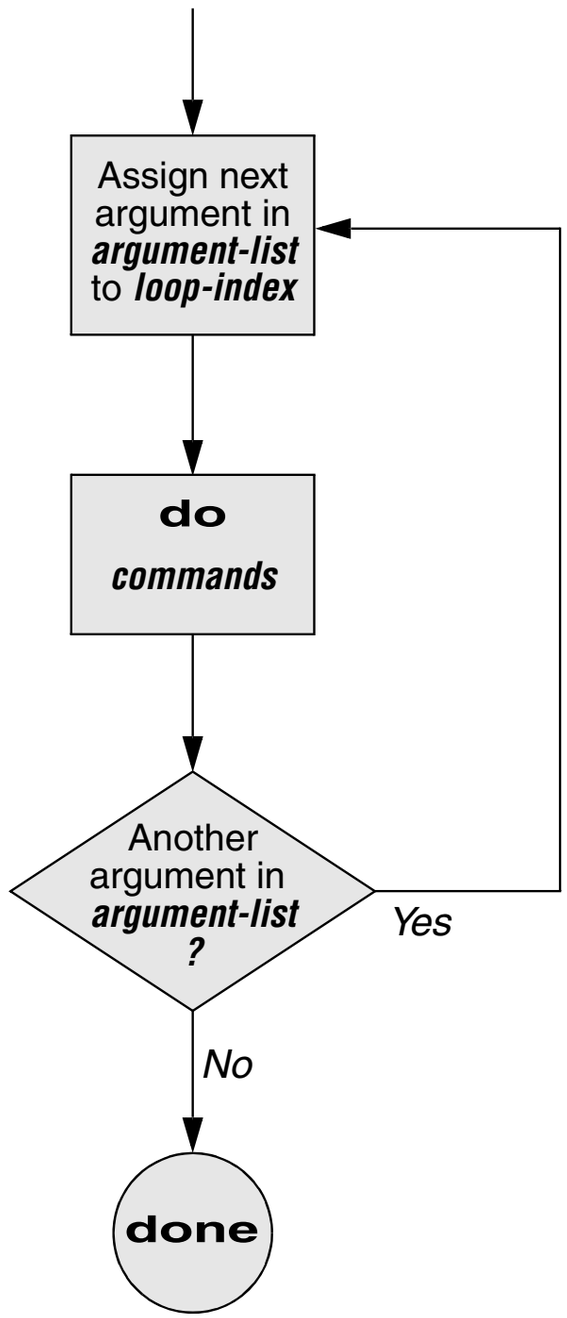


Control Structure: for-in

for *loop-index* in *argument-list*
do
    commands
done

Example:

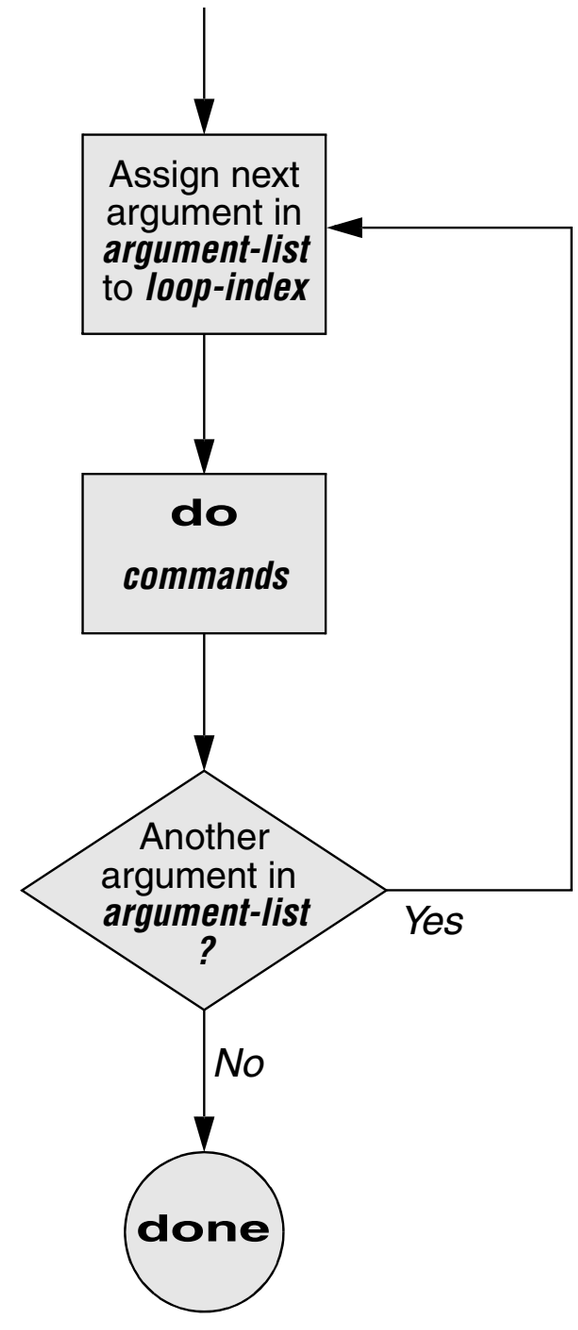
```
for i in *
do
    if [ -d "$i" ]
      then
        echo "$i"
     fi
done
```



Control Structure: for

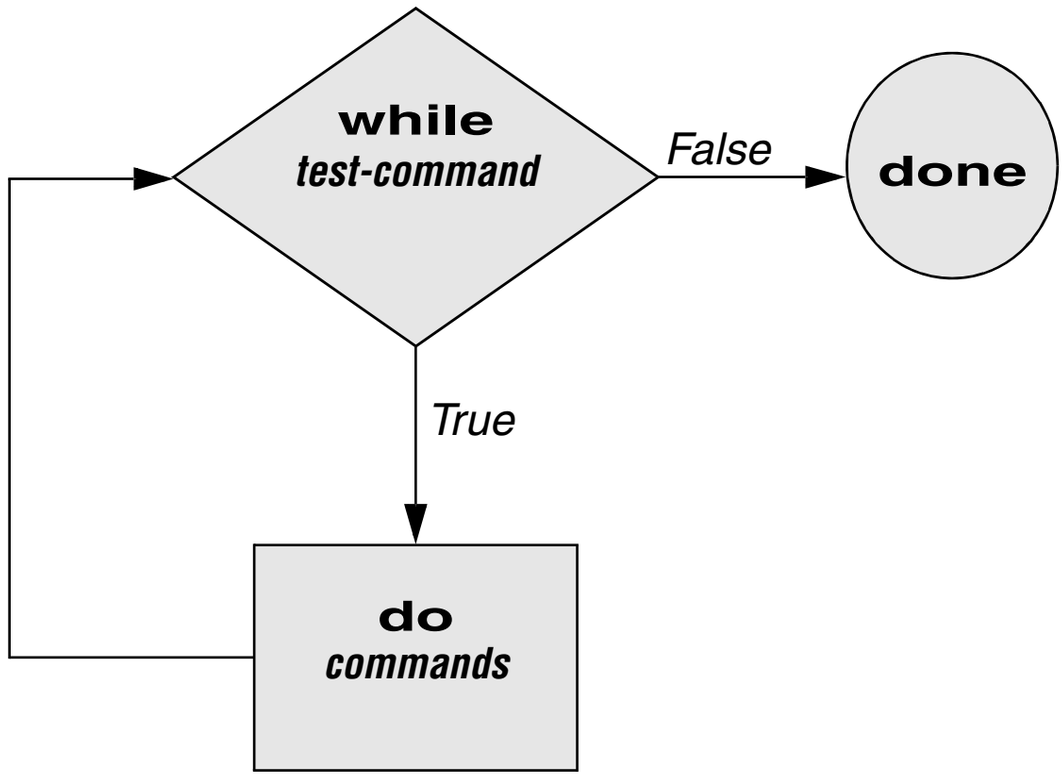
for *loop-index*
do
	commands
done

Example:
for arg
do
	echo "$arg"
done



Control Structure: while

while *test-command*
do
	commands
done

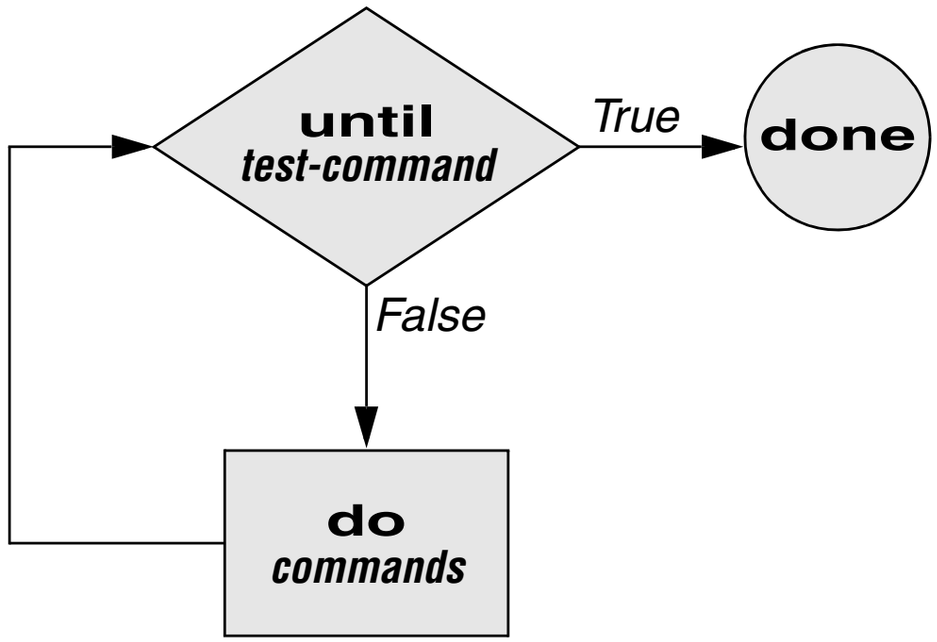


Example:

```
number=0
while [ "$number" -lt 10 ]
do
	echo -n "$number"
	((number +=1))
done
echo
```

Control Structure: until

until *test-command*
do
commands
done



Example:

```
secretname=zach
name=noname
echo "Try to guess the secret name!"
until [ "$name" = "$secretname" ]
do
    echo -n "Your guess: "
    read name
 done
echo "Very good."
```

Break and Continue

- You can interrupt a for, while, or until loop by using a **break** or **continue** statement

- The break statement transfers control to the statement after the done statement, thereby **terminating** execution of the loop

- The continue command transfers control to the done statement, **continuing** execution of the loop

```
for index in 1 2 3 4 5 6 7 8 9 10
        do
            if [ $index -le 3 ] ; then
                echo "continue"
                continue
            fi
#
        echo $index
    #
        if [ $index -ge 8 ] ; then
                    echo "break"
                    break
         fi
        done
```

Source or “.” Command in Script

- Using . command or source command will run a script in the current shell without forking a sub-shell

- For example:

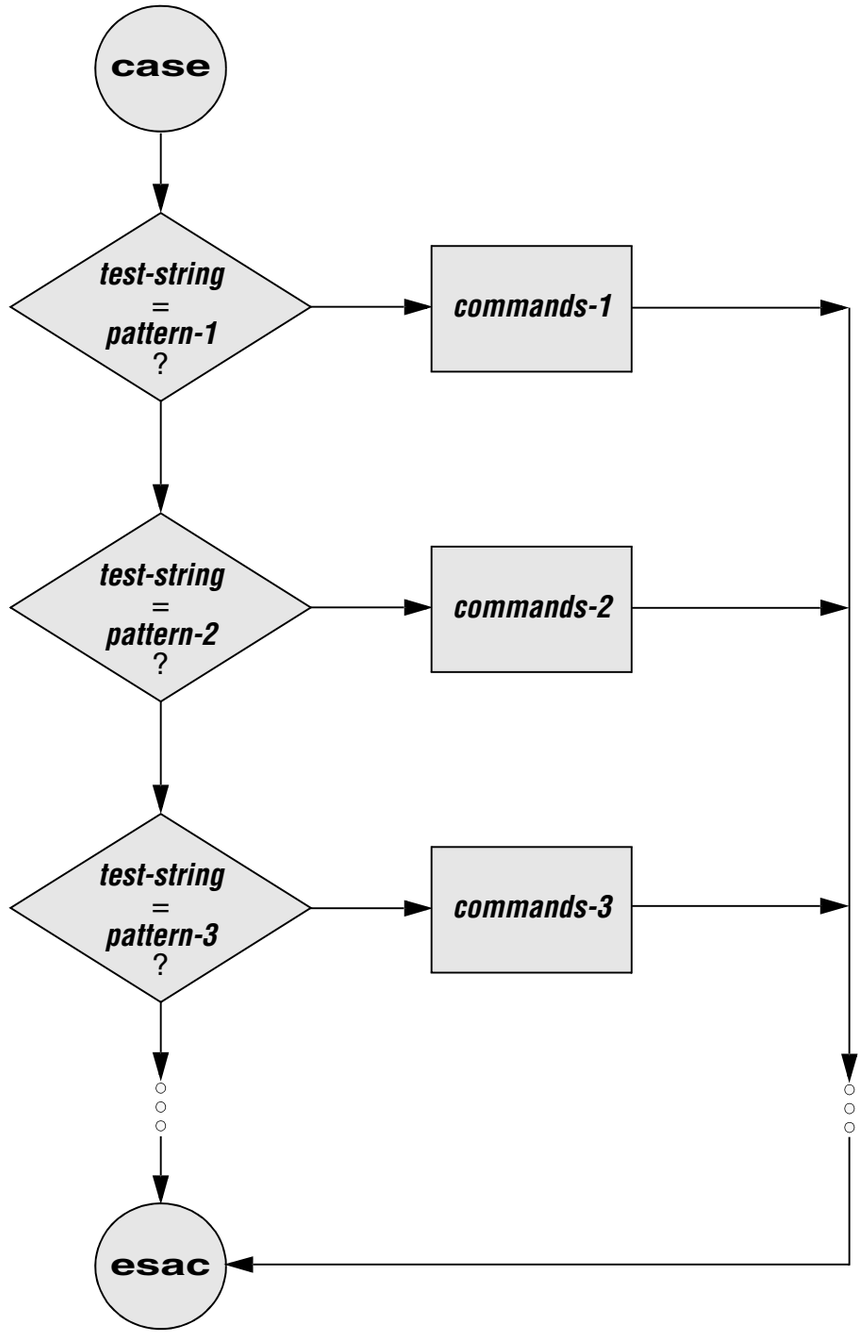
. script.sh

source script.sh

Control Structure: case

```
case test-string in
        pattern-1)
                commands-1
                ;;
        pattern-2)
                commands-2
                ;;
        pattern-3)
                commands-3
                ;; ...
esac
```

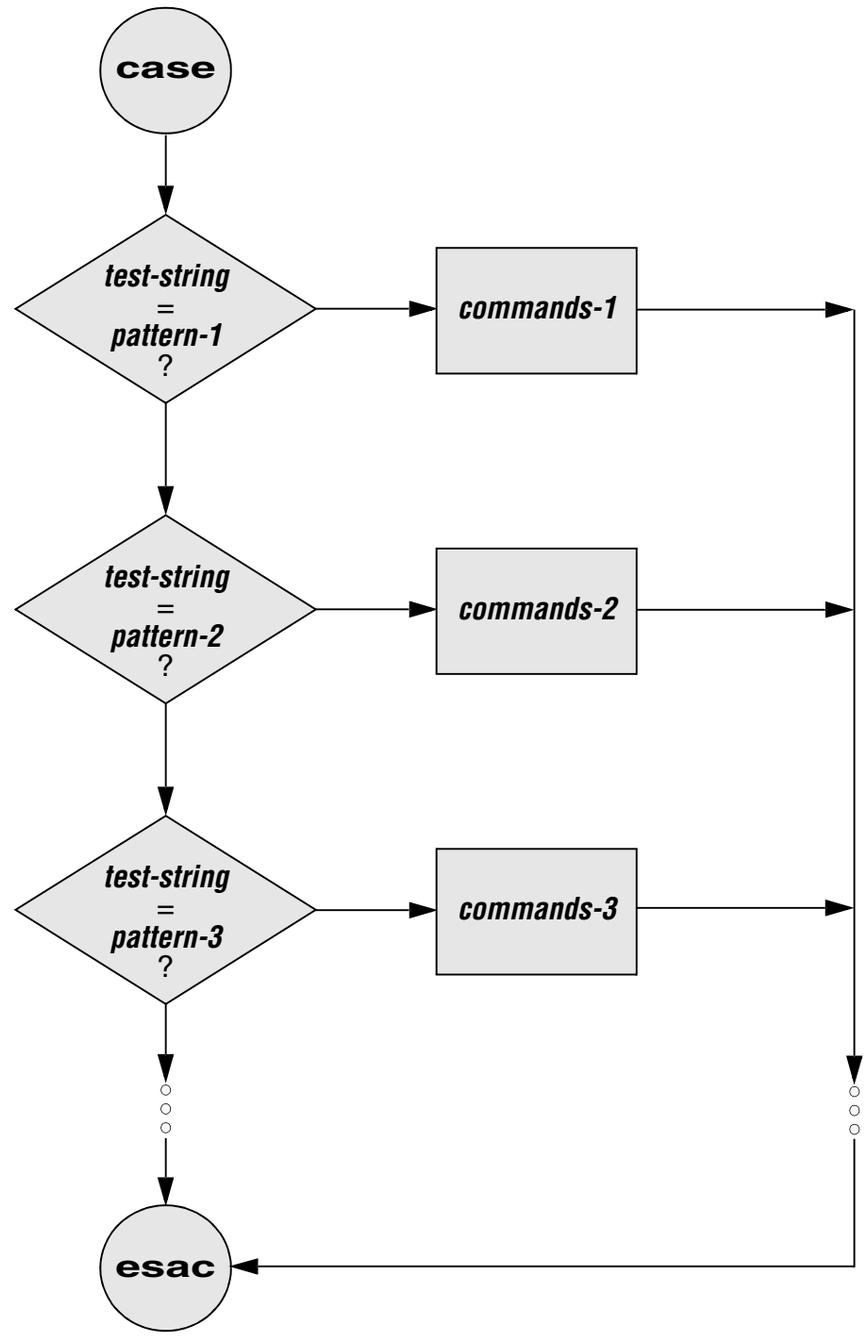
* patterns also be:
* - any string
? - any single character
[...], [a-z] - any characters or range in brackets
a|A - matches alternate choices



Control Structure: case, cont.

Example:

```
echo -n "Enter A, B, or C: "
read letter
case "$letter" in
      a|A)
         echo "You entered A"
         ;;
      b|B)
         echo "You entered B"
         ;;
      c|C)
         echo "You entered C"
        ;;
       *)
         echo "You did not enter A, B, or C"
         ;;
esac
```



Array Variables

- Bash supports one dimensional **Array Variables**
  - NAMES=(max helen sam zach)
  - echo ${NAMES[2]} #will return element 2

Special Parameters

Parameter	Definition
$$	PID Number
$?	Exit Status
$#	Number of Arguments
$0	Name of Calling Program
$1 - $n	Arguments by number
shift	Shifts arguments to left
$* and $@	Represents all command-line arguments
${name:–default}	Uses a default value for a null variable

More Built-in Commands

Parameter	Definition
read	Accepts input from the user and store that input in variables; -p prompt, -a array

String Pattern Matching

- The Bourne Again Shell provides string pattern-matching operators that can manipulate pathnames and other strings
- These operators can delete from strings prefixes or suffixes that match patterns
- # removes minimal prefix, ## removes maximal prefix, % removes minimal suffixes, and %% removes maximal suffixes

```
$ SOURCEFILE=/usr/local/src/prog.c

$ echo ${SOURCEFILE#/*/}
local/src/prog.c

$ echo ${SOURCEFILE##/*/}
prog.c

$ echo ${SOURCEFILE%/*}
/usr/local/src
```

Regular Expressions 1

Expression	Matches	Examples
/ring/	Any string	ring, spring, ringing, stinging
/.ing/	“.” Matches any single character	sing, ping, inglenook
/[bB]ill/	Matches any single character	Bill, bill
/#[6-9]/	“-” matches any range of characters	#60, #8, #9
/[^a-zA-Z]/	“^” any character that is not a within the brackets	1, 7, @
/t*c/	“*” matches any number of preceding characters	tttc, tc, ttc
/t.*ing/	“.*” matches any number of any character	thing, ting

Regular Expressions 2

Expression	Matches	Examples
/^T/	A regular expression that begins with a caret (^) can match a string only at the beginning of a line	This line..., That Time...,
/:$/	In a similar manner, a dollar sign ($) at the end of a regular expression matches the end of a line	...below:
/end \ ./	You can quote any special character by preceding it with a backslash	The end., send., pretend.mail

AWK

awk ‘ {instructions} ‘ file(s)
awk ‘/pattern/ {procedure}’ file
awk -f script_file file(s)

*tokenizes fields into $1-n

awk -F: '{print "User: "$1 "\tDescription: " $5}' /etc/passwd

Other Scripting Languages

- Perl
- Python

Homework

- Read chapter 11, 13, 15, and 16
- Do page 251, Exercises 1 and 4
- Do page 356, Exercises 2, 4, and 6
- Do page 1052, Exercises 2 and 4
- Find a bash (online or within linux) of 10+ lines. Write comments to explain what the script does on each line.
- Find a short shell script and write flow chart for the script.

Contact

Ian Robert Blair, MSc.

ian.robertblair@icloud.com

QQ: 2302412574

Notes

- Add chsh [-l] to change shell