LSU

LSU INFORMATION TECHNOLOGY SERVICES

Basic Shell Scripting

Le Yan/Xiaoxu Guan

HPC User Services @ LSU

LSU CENTER FOR COMPUTATION & TECHNOLOGY

Why Shell Scripting

- Imagine your research requires you to run simulations with a set of different parameters, one job for each possible combination of values
- For each job, the (laborious and repetitive) tasks include
  - Create a directory
  - Copy input files there
  - Create a job script with proper parameters and commands then submit it
  - …

Why Shell Scripting

```
$ ./13-job-script-autogen.bash -t 270 -nt 5 -dt 10 -v
Tue Sep 20 13:54:32 CDT 2016

A Total of 5 simulations will be run under /home/lyan1/traininglab/bash_scripting_fall_2016:
Temps = 270 280 290 300 310

Using allocation myallocation

Script: /home/lyan1/traininglab/bash_scripting_fall_2016/Sim_T270/job.T270.pbs
Job: myprogram.T270

Creating the job script in /home/lyan1/traininglab/bash_scripting_fall_2016/Sim_T270

……
```

- A script can be your friend
  - Process command line arguments
  - Set up working directories
  - Generate job scripts
  - Submit jobs

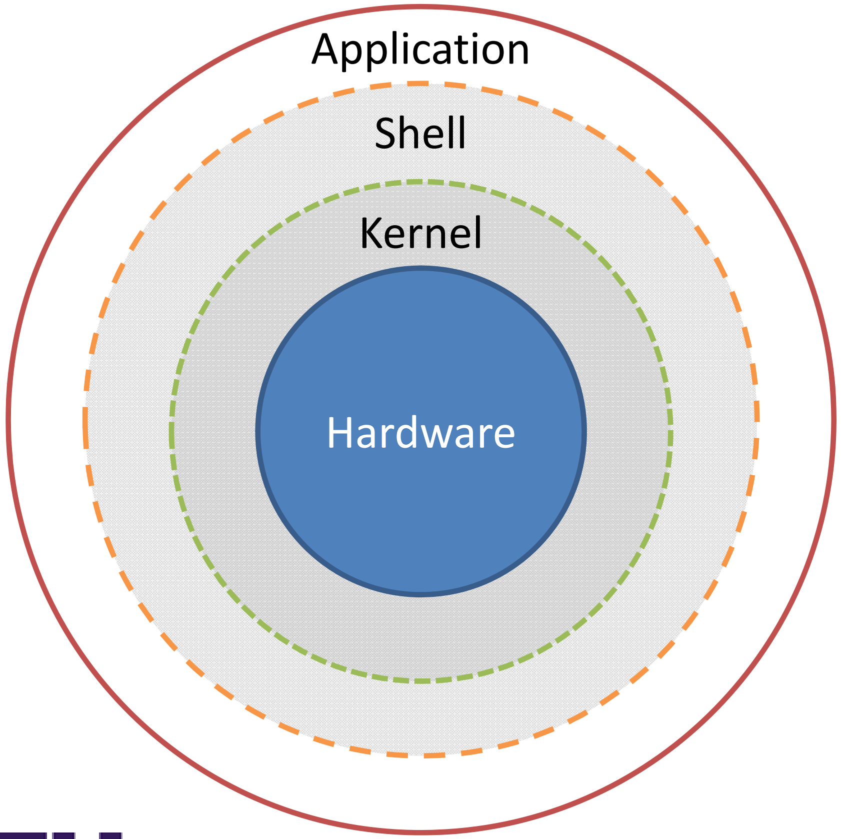
Why Shell Scripting

- Shell scripts are great for
  - Productivity and efficiency enhancement
  - System administration
  - Repetitive tasks
- Not good for
  - Heavy-duty floating point operations
  - Extensive file operations (line-by-line operations)
  - Visualization

Outline

- Recap of Linux Basics
- Shell Scripting
  - "Hello World!"
  - Special characters
  - Arithmetic Operations
  - Testing conditions
  - Flow Control
  - Command Line Arguments
  - Arrays
  - Functions
  - Pattern matching (regular expression)
  - Beyond the basics

Operating Systems



▨ : Operating System

- Operating systems work as a bridge between hardware and applications
  - Kernel: hardware drivers etc.
  - Shell: user interface to kernel
  - Some applications (system utilities)

Kernel

- Kernel
  - The kernel is the core component of most operating systems
  - Kernel’s responsibilities include managing the system’s resources
  - It provides the lowest level abstraction layer for the resources (especially processors and I/O devices) that application software must control to perform its functions
  - It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls

Shell

- Shell
  - Shell is a fundamental interface for users or applications to interact with OS kernels;
  - Shell is a special program that accepts commands from users' keyboard and executes it to get the tasks done;
  - Shell is an interpreter for command languages that reads instructions and commands;
  - Shell is a high-level programming language (compared to C/C++, Fortran, . . .);

Type of Shell

- sh (Bourne Shell)
  - Developed by Stephen Bourne at AT\&T Bell Labs
- csh (C Shell)
  - Developed by Bill Joy at University of California, Berkeley
- ksh (Korn Shell)
  - Developed by David Korn at AT&T Bell Labs
  - Backward-compatible with the Bourne shell and includes many features of the C shell
- **bash (Bourne Again Shell)**
  - **Developed by Brian Fox for the GNU Project as a free software replacement for the Bourne shell**
  - **Default Shell on Linux and Mac OSX**
  - **The name is also descriptive of what it did, bashing together the features of sh, csh and ksh**
- tcsh (TENEX C Shell)
  - Developed by Ken Greer at Carnegie Mellon University
  - It is essentially the C shell with programmable command line completion, command-line editing, and a few other features.

Shell Comparison

Software	sh	csh	ksh	bash	tcsh
Programming language	y	y	y	y	y
Shell variables	y	y	y	y	y
Command alias	n	y	y	y	y
Command history	n	y	y	y	y
Filename auto-completion	n	y*	y*	y	y
Command line editing	n	n	y*	y	y
Job control	n	y	y	y	y

*: not by default

http://www.cis.rit.edu/class/simg211/unixintro/Shell.html

Variables

- Linux allows the use of variables
  - Similar to programming languages
- A variable is a named object that contains data
  - Number, character or string
- There are two types of variables: ENVIRONMENT and user defined
- Environment variables provide a simple way to share configuration settings between multiple applications and processes in Linux
  - Environment variables are often named using all uppercase letters
  - Example: `PATH, LD_LIBRARY_PATH, DISPLAY` etc.
- To reference a variable, prepend $ to the name of the variable, e.g. `$PATH, $LD_LIBRARY_PATH`
  - Example: `$PATH, $LD_LIBRARY_PATH, $DISPLAY` etc.

Variables Names

- Rules for variable names
  - Must start with a letter or underscore
  - Number can be used anywhere else
  - Must not use special characters such as @,#,%,$
  - Case sensitive
  - Example
    - Allowed: `VARIABLE, VAR1234able, var_name, _VAR`
    - Not allowed: `1var, %name, $myvar, var@NAME`

Editing Variables (1)

- How to assign values to variables depends on the shell

Type	bash
Shell	`name=value`
Environment	`export name=value`

- Shell variables is only valid within the current shell, while environment variables are valid for all subsequently opened shells.

Editing Variables (2)

- Example: to add a directory to the PATH variable

```
export PATH=/path/to/executable:${PATH}
```

  - no spaces except between export and PATH
  - Use colon to separate different paths
  - The order matters: if you have a customized version of a software say `perl` in your home directory, if you append the `perl` path to PATH at the end, your program will use the system wide `perl` not your locally installed version

Basic Commands

- Command is a directive to a computer program acting as an interpreter of some kind, in order to perform a specific task
- Command prompt is a sequence of characters used in a command line interface to indicate readiness to accept commands
  - Its intent is literally to prompt the user to take action
  - A prompt usually ends with one of the characters $,%#,:,> and often includes information such as user name and the current working directory
- Each command consists of three parts: name, options and arguments

List of Basic Commands

Name	Function
`ls`	Lists files and directories
`cd`	Changes the working directory
`mkdir`	Creates new directories
`rm`	Deletes files and directories
`cp`	Copies files and directories
`mv`	Moves or renames files and directories
`pwd`	prints the current working directory
`echo`	prints arguments to standard output
`cat`	Prints file content to standard output

File Permission (1)

- Since *NIX OS's are designed for multi user environment, it is necessary to restrict access of files to other users on the system.
- In *NIX OS's, you have three types of file permissions
  - Read (r)
  - Write (w)
  - Execute (x)
- for three types of users
  - User (u) (owner of the file)
  - Group (g) (group owner of the file)
  - World (o) (everyone else who is on the system)

File Permission (2)

```
[lyan1@mike2 ~]$ ls -al
total 4056
drwxr-xr-x  45 lyan1 Admins     4096 Sep  2 13:30 .
drwxr-xr-x 509 root  root      16384 Aug 29 13:31 ..
drwxr-xr-x   3 lyan1 root       4096 Apr  7 13:07 adminscript
drwxr-xr-x   3 lyan1 Admins     4096 Jun  4  2013 allinea
-rw-r--r--   1 lyan1 Admins       12 Aug 12 13:53 a.m
drwxr-xr-x   5 lyan1 Admins     4096 May 28 10:13 .ansys
-rwxr-xr-x   1 lyan1 Admins   627911 Aug 28 10:13 a.out
```

- The first column indicates the type of the file
  - `d` for directory
  - `l` for symbolic link
  - `-` for normal file

File Permission (2)

```
[lyan1@mike2 ~]$ ls -al
total 4056
drwxr-xr-x  45 lyan1 Admins    4096 Sep  2 13:30 .
drwxr-xr-x 509 root  root     16384 Aug 29 13:31 ..
drwxr-xr-x   3 lyan1 root      4096 Apr  7 13:07 adminscript
drwxr-xr-x   3 lyan1 Admins    4096 Jun  4  2013 allinea
-rw-r--r--   1 lyan1 Admins      12 Aug 12 13:53 a.m
drwxr-xr-x   5 lyan1 Admins    4096 May 28 10:13 .ansys
-rwxr-xr-x   1 lyan1 Admins  627911 Aug 28 10:13 a.out
```

- The next nine columns can be grouped into three triads, which indicates what the owner, the group member and everyone else can do

File Permission (2)

```
[lyan1@mike2 ~]$ ls -al
total 4056
drwxr-xr-x  45 lyan1 Admins    4096 Sep  2 13:30 .
drwxr-xr-x 509 root  root     16384 Aug 29 13:31 ..
drwxr-xr-x   3 lyan1 root      4096 Apr  7 13:07 adminscript
drwxr-xr-x   3 lyan1 Admins    4096 Jun  4  2013 allinea
-rw-r--r--   1 lyan1 Admins      12 Aug 12 13:53 a.m
drwxr-xr-x   5 lyan1 Admins    4096 May 28 10:13 .ansys
-rwxr-xr-x   1 lyan1 Admins  627911 Aug 28 10:13 a.out
```

- We can also use weights to indicate file permission
  - r=4, w=2, x=1
  - Example: rwx = 4+2+1 = 7, r-x = 4+1 = 5, r-- = 4
  - This allows us to use three numbers to represent the permission
  - Example: rwxr-xr-w = 755

Input & Output Commands (1)

- The basis I/O statement are echo for displaying to screen and read for reading input from screen/keyboard/prompt
- **`echo`**
  - The `echo <arguments>` command will print arguments to screen or standard output, where `arguments` can be a single or multiple variables, string or numbers
- **`read`**
  - The `read` statement takes all characters typed until the Enter key is pressed
  - Usage: `read <variable name>`
  - Example: `read name`

Input & Output Commands (2)

- Examples

```
$ echo $SHELL
/bin/bash
$ echo Welcome to HPC      training
Welcome to HPC training
$ echo "Welcome to HPC     training"
Welcome to HPC     training
```

- By default, `echo` eliminates redundant whitespaces (multiple spaces and tabs) and replaces it with a single whitespace between arguments.
  - To include redundant whitespace, enclose the arguments within double quotes

I/O Redirection

- There are three file descriptors for I/O streams (remember everything is a file in Linux)
  - STDIN: Standard input
  - STDOUT: standard output
  - STDERR: standard error
- 1 represents STDOUT and 2 represents STDERR
- I/O redirection allows users to connect applications
  - <: connects a file to STDIN of an application
  - >: connects STDOUT of an application to a file
  - >>: connects STDOUT of an application by appending to a file
  - |: connects the STDOUT of an application to STDIN of another application.

Outline

- Recap of Linux Basics
- Shell Scripting
  - “Hello World!”
  - Special characters
  - Arithmetic Operations
  - Testing conditions
  - Flow Control
  - Command Line Arguments
  - Arrays
  - Functions
  - Pattern matching (regular expression)
  - Beyond the basics

Writing and Executing a Script

```
$ cat 01-hello-world.bash
#!/bin/bash
## Print Hello world ...
echo
echo "Hello World!"
echo "Greetings from" $USER "in" `pwd` "on" $HOSTNAME
echo "Today is" `date +"%A %B-%d-%Y"`
echo
$ chmod a+x 01-hello-world.bash
$ ./01-hello-world.bash

Hello World!
Greetings from lyan1 in /home/lyan1/traininglab/bash_scripting_fall_2016 on
mike5
Today is Tuesday September-20-2016
```

- Three steps
  - Create and edit a text file
  - Set the appropriate permission
  - Execute the script

Shell Script Components

```
#!/bin/bash
## Print Hello world ...
echo
echo "Hello World!"
echo "Greetings from" $USER "in" `pwd` "on" $HOSTNAME
echo "Today is" `date +"%A %B-%d-%Y"`
echo
```

- **Shebang line**: the first line is called the "Shebang" line. It tells the OS which interpreter to use.
- **Comments**: the second line is a comment. All comments begin with "#".
- **Commands**: the third line and below are commands.

Be A Good "Scriptor"

- Add comments and annotations so that people, yourself included, can understand what the script does
- Print helpful, human-readable information to screen
  - Don't print garbage though!

Special Characters and Operators (1)

`#`	Starts a comment line.
`$`	Indicates the name of a variable.
`\`	Escape character to display next character literally
`{}`	Used to enclose name of variable
`;`	Command separator, which allows one to put two or more commands on the same line.
`;;`	Terminator in a `case` option
`.`	"dot" command. Equivalent to `source` (for bash only)

Special Characters and Operators (2)

$?	Exit status of the last executed command.
$$	Process ID the current process.
[]	Test enclosed expression.
[[]]	Test enclosed expression. Has more functionalities than []
$[], $(())	Integer expansion
&&	Logical AND
||	Logical OR
!	Logical NOT

Quotes

- Double quotes “”
  - Enclosed string is expanded
- Single quotation ‘ ’
  - Enclosed string is read literally
- Back quotation ` `
  - Enclose string is executed as a command

Special Characters - Examples

```
$ HI=Hello
$ echo HI
HI
$ echo $HI
Hello
$ echo \$HI
$HI
$ echo "$HI"
Hello
$ echo '$HI'
$HI
$ echo "$HILe"

$ echo "${HI}Le"
HelloLe
$ echo `pwd`
/home/lyan1/traininglab/bash_scripting_fall_2016
```

Arithmetic Operations (1)

- You can carry out a number of numeric integer operations

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	`** (bash only)`
Modulo	%

Arithmetic Operations (2)

- `$((…))` or `$[…]` commands
  - Addition: `$((1+2))`
  - Multiplication: `$[$a*$b]`
  - The $ sign can be dropped: `$[a*b]`
- Or use the `let` command: `let c=$a-$b`
- Or use the `expr` command: `c=‘expr $a - $b‘`
- You can also use C-style increment operators:

  `let c+=1` or `let c--`
- Space required around operator in the `expr` command

Arithmetic Operations (3)

- For floating numbers
  - You would need an external calculator like the GNU `bc`
    - Add two numbers

```
echo "3.8 + 4.2" | bc
```

    - Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

    - Call `bc` directly:

```
bc <<< “scale=5; 2/5”
```

    - Use `bc -l` to see result in floating point at max scale:

```
bc -l <<< "2/5"
```

Declare command

- Use the `declare` command to set variable and functions attributes
- Create a constant variable, i.e. read-only
  - `declare -r var`
  - `declare -r varName=value`
- Create an integer variable
  - `declare -i var`
  - `declare -i varName=value`
- You can carry out arithmetic operations on variables declared as integers

```
$ j=10/5; echo $j
10/5
$ declare -i j; j=10/5; echo $j
2
```

Test Conditions

- Command: `test`
  - Evaluate a conditional expression
- Use the exit status variable ($?) to check results

```
$ x=1
$ test $x == "1"
$ echo $?
0
$ test $x -eq "0"; echo $?
1
```

File Tests

Operation	bash
File exists	`-e .bashrc`
File is a regular file	`-f .bashrc`
File is a directory	`-d /home`
File is not zero size	`-s .bashrc`
File has read permission	`-r .bashrc`
File has write permission	`-w .bashrc`
File has execute permission	`-x .bashrc`

Integer Comparisons

Operation	bash
Equal to	`1 -eq 2`
Not equal to	`$a -ne $b`
Greater than	`$a -gt $b`
Greater than or equal to	`1 -ge $b`
Less than	`$a -lt 2`
Less than or equal to	`$a -le $b`

String Comparisons

Operation	bash
Equal to	`$a == $b`
Not equal to	`$a != $b`
Zero length or null	`-z $a`
Non zero length	`-n $a`

Logical Operators

Operation	bash
! (NOT)	`! -e .bashrc`
-a (AND)	`-f .bashrc –a -s .bashrc`
-o (OR)	`-f .bashrc -o -f .bash_profile`

Flow Control

- Shell scripting languages execute commands in sequence similar to other programming languages
  - Control constructs can change the order of command execution
- Control constructs in `bash` are
  - Conditionals: `if`
  - Loops: `for, while, until`
  - Switches: `case, switch`

if…else…fi Constructs

- Tests whether the exit status of a list of commands is 0, and if so, execute one or more commands

```
if [ condition 1 ]; then
  some commands
elif [ condition 2 ]; then
  some commands
else
  some commands
fi
```

- Note the space between condition and the brackets

```
# Create a directory if it does not exist
if ! [ -e /path/to/directory ];
then
  mkdir /path/to/directory
fi

if [ $x –ge 10 ] –a [ $x –gt 100 ] ; then
  echo “something”
else
  echo “something else”
fi

# Equivalent to the construct above
if [[ $x –ge 10 && $x –gt 100 ]] ; then
  echo “something”
else
  echo “something else”
fi
```

Loop Constructs

- A loop is a block of code that iterates a list of commands
- When to stop depends on the loop control condition
- Loop constructs
  - bash: `for, while` and `until`

For Loop

- The `for` loop is the basic looping construct in **bash**

```
for arg in list
do
  some commands
done
```

- The `for` and `do` lines can be written on the same line:

```
for arg in list; do
```

- `for` loops can also use C style syntax

For Loop

```
for i in `seq 1 10`
do
  echo $i
done
```

```
for i in ((i=1;i<=10;i++))
do
  echo $i
done
```

While Loop

- The `while` construct tests for a condition at the top of a loop and keeps going as long as that condition is true.
- In contrast to a `for` loop, a `while` loop finds use in situations where the number of loop repetitions is not known beforehand.

```
while [ condition ]
do
  some commands
done
```

While Loop - Example

```
$ cat factorial.sh

#!/bin/bash

read counter
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial

$ ./factorial.sh
4
24
```

Until Loop

- The `until` construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (the opposite of `while` loop)

```
until[ condition ]
do
  some commands
done
```

Switching Constructs - bash

- The `case` and `select` constructs are technically not loops since they do not iterate the execution of a code block
- Like loops, however, they direct program flow according to conditions at the top or bottom of the block

```
# case construct
case variable in
  “condition1”)
    some commands
    ;;
  “condition2”)
    some commands
    ;;
esac
```

```
# select construct
select variable [ list ]
do
  some commands
  break
done
```

```
$ ./dooper.sh
Print two numbers
4 8
What operation do you want to do?
1) add            3) multiply       5) exponentiate  7) all
2) subtract       4) divide         6) modulo        8) quit
#? 3
4 * 8 = 32
#? 7
4 + 8 = 12
4 - 8 = -4
4 * 8 = 32
4 ** 8 = 65536
4 / 8 = 0
4 % 8 = 4
#? 8
```

Command Line Arguments

- bash can take command line arguments
  - Execute `./myscript arg1 arg2 arg3`
  - Within the script, the positional parameters `$0, $1, $2, $3` correspond to `./myscript, arg1, arg2,` and `arg3`, respectively.
  - `$#`: number of command line arguments
  - `$*`: all of the positional parameters, seen as a single word
  - `$@`: same as `$*` but each parameter is a quoted string.
  - `shift N`: shift positional parameters from N+1 to $# are renamed to variable names from $1 to `$# - N + 1`

```
$ cat shift.sh
#!/bin/bash

USAGE="USAGE: $0 <at least 1 argument>"
if [[ "$#" -lt 1 ]]; then
   echo $USAGE
   exit
fi
echo "Number of Arguments: " $#
echo "List of Arguments: " $@
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 $*
while [ "$#"  -gt 0 ]; do
  echo "Argument List is: " $@
  echo "Number of Arguments: " $#
  shift
done

$ ./shift.sh `seq 4 6`
Number of Arguments:  3
List of Arguments:  4 5 6
Name of script that you are running:  ./shift.sh
Command You Entered: ./shift.sh 4 5 6
Argument List is:  4 5 6
Number of Arguments:  3
Argument List is:  5 6
Number of Arguments:  2
Argument List is:  6
Number of Arguments:  1
```

Arrays (1)

- bash supports one-dimensional arrays
- Array elements may be initialized with the `variable[i]` notation
  - Example: `xarray[4]=1`
- Or initialize an array during declaration
  - Example: `name=($firstname ’last name’)`
- Reference an element `i` of an array `name`: `${name[i]}`
- Print the whole array
  - Example: `${name[@]}`
- Print length of array
  - Example: `${#name[@]}`

```
$ cat name.sh
#!/bin/bash

echo "Print your first and last name"
read firstname lastname
name=($firstname $lastname)
echo "Hello " ${name[@]}
echo "Enter your salutation"
read title
echo "Enter your suffix"
read suffix
name=($title "${name[@]}" $suffix)
echo "Hello " ${name[@]}
unset name[2]
echo "Hello " ${name[@]}

$ ./name.sh
Print your first and last name
Le Yan
Hello  Le Yan
Enter your salutation
Mr.
Enter your suffix

Hello  Mr. Le Yan
Hello  Mr. Le
```

Arrays (2)

- Print length of element `i` of array `name`: `${#name[i]}`
  - `${#name}` prints the length of the first element of the array
- Add an element to an existing array
  - Example: `name=($title ${name[@]})`
- Copy the values of an array to another
  - Example: `user=(${name[@]})`

Arrays (3)

- Concatenate two arrays
  - **Example:** `nameuser=(${name[@]} ${user[@]})`
- Delete an entire array: `unset name`
- Remove an element i from an array
  - `unset name[i]`
- Note: bash array index starts from 0

Functions (1)

- A function is a code block that implements a set of operations, a “black box” that performs a specified task.
- Consider using a function
  - When there is repetitive code
  - When a task repeats with only slight variations in procedure
- Two ways to declare a function:

```
function function_name {
   some commands
}
```

```
function_name () {
   some commands
}
```

```
$ cat shift10.sh
#!/bin/bash

usage () {
        echo "USAGE: $0 [atleast 11 arguments]"
  exit
}

[[ "$#" -lt 11 ]] && usage

echo "Number of Arguments: " $#
echo "List of Arguments: " $@
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 $*
echo "First Argument" $1
echo "Tenth and Eleventh argument" $10 $11 ${10}
${11}

echo "Argument List is: " $@
echo "Number of Arguments: " $#
shift 9
echo "Argument List is: " $@
echo "Number of Arguments: " $#

$ ./shift10.sh
USAGE: ./shift.sh <at least 1 argument>
```

Functions (2)

- You can also pass arguments to a function as if it were a script
  - Exampe: `func arg1 arg2 arg3`
- All function parameters can be accessed via `$1, $2, $3`…
- `$0` always point to the shell script name
- `$*` or $@ holds all parameters passed to a function
- $# holds the number of positional parameters passed to the function

Functions (3)

- Array variable called `FUNCNAME` contains the names of all shell functions currently in the execution call stack.
- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- You can create a local variables using the `local` command

```
local var=value
local varName
```

```
$ cat factorial3.sh
#!/bin/bash

usage () {
  echo "USAGE: $0 <integer>"
  exit
}

factorial() {
  local i=$1
  local f

  declare -i i
  declare -i f

  if [[ "$i" -le 2 && "$i" -ne 0 ]]; then
    echo $i
  elif [[ "$i" -eq 0 ]]; then
    echo 1
  else
    f=$(( $i - 1 ))
    f=$( factorial $f )
    f=$(( $f * $i ))
    echo $f
  fi
}
```

```
if [[ "$#" -eq 0 ]]; then
  usage
else
  for i in $@ ; do
    x=$( factorial $i )
    echo "Factorial of $i is $x"
  done
fi
$ ./factorial3.sh
USAGE: ./factorial3.sh <integers>
$ ./factorial3.sh 1 2 4 6 7
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 4 is 24
Factorial of 6 is 720
Factorial of 7 is 5040
```

A function may recursively call itself even without use of local variables.

Regular Expressions

- A regular expression (regex) is a extremely powerful method of using a pattern to describe a set of strings.
- Regular expressions enable strings that match a particular pattern within textual data records to be located and modified.
- Regular expressions are constructed analogously to arithmetic expressions by using various operators to combine smaller expressions
- Regular expressions are often used within utility programs that manipulate textual data.
  - Command line tools: `grep`, `egrep`, `sed`
  - Editors: `vi`, `emacs`
  - Languages: `awk, perl, python, php, ruby, tcl, java, javascript, .NET`

Bracket Expression

- `[ ]`
  - Matches a single character that is contained within the brackets. For example, `[abc]` matches "a", "b", or "c". `[a-z]` specifies a range which matches any lowercase letter from "a" to "z".
  - These forms can be mixed: `[abcx-z]` matches "a", "b", "c", "x", "y", or "z", as does `[a-cx-z]`.
- `[^ ]`
  - Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "a", "b", or "c". `[^a-z]` matches any single character that is not a lowercase letter from "a" to "z".

Metacharacters

`.`	Any character except \n		
`^`	Start of a line		
`$`	End of a line		
`\s`	Any whitespace	`\S`	Any non-whitespace
`\d`	Any digit	`\D`	Any non-digit
`\w`	Any word	`\W`	Any non-word
`\b`	Any word boundary	`\B`	Anything except a word boundary

Special characters such as `.`, `^`, `$` need to be escaped by using "\" if the literals are to be matched

Regex Examples

Pattern	Matches	Does not match

`line`	`Feline animals`
	`Two linebackers`
		`Three LINEs`
		`FeLine animals`

`^line`	`line 1 has eight characters`
	`linebackers`
		`feline animals`
		`two lines of text`

`\bline\b`	`There are one line of text`	`Three lines of text`
		`The file has only one line.`

`l.ne`	`line`
	`lane`
	`l$ne`	

`\$\d\.\d\d`	`$3.88`	

Quantifiers

- Allow users to specify the number of occurrence

*	Zero or more
+	One or more
?	Zero or one
`{n}`	Exactly n times
`{n,}`	At least n times
`{n,m}`	At least n, but at most m times

Regex Examples

Pattern	Matches
`ca?t`	ct cat
`ca*t`	ct cat caat
`ca+t`	cat caat caaat
`ca{3}t`	caaat
`\d{3}-?\d{3}-?\d{4}`	2255787844 000-000-0000 291-1938183 573384-2333

Alternation

- Use "|" to match either one thing or another
  - Example: `pork|chicken` will match `pork` or `chicken`

Grouping And Capturing Matches

- Use parentheses ( ) to
  - Group atoms into larger unit
  - Capture matches for later use

Pattern	Matches

`(ca?t){2,3}`	`ctct`
	`ctctct`
	`catcat`
	`catcatcat`

`(\d*\.)?\d+`	Integers or decimals

A Great Learning Source

- http://www.regexr.com/

grep

- `grep` is a Unix utility that searches through either information piped to it or files.
- `egrep` is extended `grep` (extended regular expressions), same as `grep -E`
- Use `zgrep` for compressed files.
- Usage: `grep <options> <search pattern> <files>`
- Commonly used options

`-i`	ignore case during search
`-r,-R`	search recursively
`-v`	invert match i.e. match everything except *`pattern`*
`-l`	list files that match *`pattern`*
`-L`	list files that do not match *`pattern`*
`-n`	prefix each line of output with the line number within its input file.
`-A num`	print `num` lines of trailing context after matching lines.
`-B num`	print `num` lines of leading context before matching lines.

grep Examples (1)

- Search files that contain the word node in the examples directory

```
$ egrep node *

checknodes.pbs:#PBS -o nodetest.out
checknodes.pbs:#PBS -e nodetest.err
checknodes.pbs:for nodes in "${NODES[@]}"; do
checknodes.pbs: ssh -n $nodes ’echo $HOSTNAME ’$i’ ’ &
checknodes.pbs:echo "Get Hostnames for all unique nodes"
```

- Repeat above search using a case insensitive pattern match and print line number that matches the search pattern

```
$ egrep -in node *

checknodes.pbs:20:NODES=(‘cat "$PBS_NODEFILE"‘ )
checknodes.pbs:21:UNODES=(‘uniq "$PBS_NODEFILE"‘ )
checknodes.pbs:23:echo "Nodes Available: " ${NODES[@]}
checknodes.pbs:24:echo "Unique Nodes Available: " ${UNODES[@]}
checknodes.pbs:28:for nodes in "${NODES[@]}"; do
checknodes.pbs:29: ssh -n $nodes ’echo $HOSTNAME ’$i’ ’ &
checknodes.pbs:34:echo "Get Hostnames for all unique nodes"
checknodes.pbs:39: ssh -n ${UNODES[$i]} ’echo $HOSTNAME ’$i’ ’
```

grep Examples (2)

- Print files that contain the word "counter”

```
$ egrep -l counter *

factorial2.sh
factorial.csh
factorial.sh
```

- List all files that contain a comment line i.e. lines that begin with "#“

```
$ egrep -l “^#” *

backups.sh
checknodes.pbs
dooper1.sh
dooper.csh
dooper.sh
factorial2.sh
factorial3.sh
factorial.csh
factorial.sh
hello.sh
name.csh
name.sh
```

grep Examples (3)

- List all files that are bash or csh scripts i.e. contain a line that end in bash or csh

```
$ egrep -l “bash$|csh$” *

backups.sh
checknodes.pbs
dooper1.sh
dooper.csh
dooper.sh
factorial2.sh
factorial3.sh
factorial.csh
factorial.sh
hello.sh
name.csh
name.sh
nestedloops.csh
nestedloops.sh
quotes.csh
quotes.sh
shift10.sh
```

Outline

- Recap of Linux Basics
- Shell Scripting
  - "Hello World!"
  - Special characters
  - Arithmetic Operations
  - Testing conditions
  - Flow Control
  - Command Line Arguments
  - Arrays
  - Functions
  - Pattern matching (regular expression)
  - Beyond the basics

sed

- sed ("stream editor") is Unix utility for parsing and transforming text files.
  - Also works for either information piped to it or files
- sed is line-oriented - it operates one line at a time and allows regular expression matching and substitution.
- sed has several commands, the most commonly used command and sometime the only one learned is the substitution command, s

```
echo day | sed ’s/day/night/’

night
```

List of sed commands and flags

Flags	Operation	Command	Operation
`-e`	combine multiple commands	`s`	substitution
`-f`	read commands from file	`g`	global replacement
`-h`	print help info	`p`	print
`-n`	disable print	`i`	ignore case
`-V`	print version info	`d`	delete
`-r`	use extended regex	`G`	add newline
		`w`	write to file
		`x`	exchange pattern with hold buffer
		`h`	copy pattern to hold buffer
		`;`	separate commands

sed Examples (1)

- Double space a file

```
$ sed G hello.sh

#!/bin/bash


# My First Script


echo "Hello World!"
```

- Triple space a file sed ’G;G’

sed Examples (2)

- Add the -e to carry out multiple matches.

```
cat hello.sh | sed -e ’s/bash/tcsh/g’ -e ’s/First/First tcsh/g’

#!/bin/tcsh
# My First tcsh Script
echo "Hello World!"
```

- Alternate form

```
sed ’s/bash/tcsh/g; s/First/First tcsh/g’ hello.sh

#!/bin/tcsh
# My First tcsh Script
echo "Hello World!"
```

- The default delimiter is slash (/), but you can change it to whatever you want which is useful when you want to replace path names

```
sed ’s:/bin/bash:/bin/tcsh:g’ hello.sh

#!/bin/tcsh
# My First Script
echo "Hello World!"
```

sed Examples (3)

- If you do not use an alternate delimiter, use backslash (\) to escape the slash character in your pattern

```
sed ’s/\/bin\/bash/\/bin\/tcsh/g’ hello.sh

#!/bin/tcsh
# My First Script
echo "Hello World!"
```

- If you enter all your `sed` commands in a file, say `sedscript`, you can use the `-f` flag to `sed` to read those commands

```
cat sedscript

s/bash/tcsh/g

sed -f sedscript hello.sh

#!/bin/tcsh
# My First Script
echo "Hello World!"
```

sed Examples (4)

- `sed` can also delete blank lines from a file

```
sed '/^$/d' hello.sh

#!/bin/bash
# My First Script
echo "Hello World!"
```

- Delete line `n` through `m` in a file

```
sed '2,4d' hello.sh

#!/bin/bash
echo "Hello World!"
```

- Insert a blank line above every line which matches *`pattern`*

```
sed '/First/{x;p;x}' hello.sh

#!/bin/bash


# My First Script

echo "Hello World!"
```

sed Examples (5)

- Insert a blank line below every line which matches *pattern*

```
sed '/First/G' hello.sh

#!/bin/bash

# My First Script


echo "Hello World!"
```

- Insert a blank line above and below every line which matches *pattern*

```
sed '/First/{x;p;x;G}' hello.sh

#!/bin/bash


# My First Script


echo "Hello World!"
```

sed Examples (6)

- Print only lines which match *`pattern`* (emulates `grep`)

```
sed -n '/echo/p' hello.sh

echo "Hello World!"
```

- Print only lines which do NOT match *`pattern`* (emulates `grep -v`)

```
sed -n '/echo/!p' hello.sh

#!/bin/bash
# My First Script
```

- Print current line number to standard output

```
sed -n '/echo/ =' quotes.sh

5
6
7
8
9
10
11
12
13
```

sed example (7)

- If you want to make substitution in place, i.e. in the file, then use the `-i` command. If you append a suffix to `-i`, then the original file will be backed up as *`filename.suffix`*.

```
cat hello1.sh

#!/bin/bash
# My First Script
echo "Hello World!"

sed -i.bak -e ’s/bash/tcsh/g’ -e ’s/First/First tcsh/g’ hello1.sh

cat hello1.sh

#!/bin/tcsh
# My First tcsh Script
echo "Hello World!"

cat hello1.sh.bak

#!/bin/bash
# My First Script
echo "Hello World!"
```

sed Examples (8)

- Print section of file between *`pattern1`* and *`pattern2`*

```
cat nh3-drc.out | sed -n '/START OF DRC CALCULATION/,/END OF ONEELECTRON INTEGRALS/p'

START OF DRC CALCULATION
************************
---------------------------------------------------------------------------
TIME MODE Q P KINETIC POTENTIAL TOTAL
FS BOHR*SQRT(AMU) BOHR*SQRT(AMU)/FS E ENERGY ENERGY
0.0000 L 1 1.007997 0.052824 0.00159 -56.52247 -56.52087
L 2 0.000000 0.000000
L 3 -0.000004 0.000000
L 4 0.000000 0.000000
L 5 0.000005 0.000001
L 6 -0.138966 -0.014065
---------------------------------------------------------------------------
CARTESIAN COORDINATES (BOHR) VELOCITY (BOHR/FS)
---------------------------------------------------------------------------
7.0 0.00000 0.00000 0.00000 0.00000 0.00000 -0.00616
1.0 -0.92275 1.59824 0.00000 0.00000 0.00000 0.02851
1.0 -0.92275 -1.59824 0.00000 0.00000 0.00000 0.02851
1.0 1.84549 0.00000 0.00000 0.00000 0.00000 0.02851
---------------------------------------------------------------------------
----------------------
GRADIENT OF THE ENERGY
----------------------
UNITS ARE HARTREE/BOHR E'X E'Y E'Z
1 NITROGEN 0.000042455 0.000000188 0.000000000
2 HYDROGEN 0.012826176 -0.022240529 0.000000000
3 HYDROGEN 0.012826249 0.022240446 0.000000000
4 HYDROGEN -0.025694880 -0.000000105 0.000000000
...... END OF ONE-ELECTRON INTEGRALS ......
```

sed Examples (9)

- Print section of file from *`pattern`* to end of file

```
cat h2o-opt-freq.nwo | sed -n '/CITATION/,$p'

CITATION
--------
Please use the following citation when publishing results
obtained with NWChem:
E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma,
M. Valiev, D. Wang, E. Apra, T. L. Windus, J. Hammond, P. Nichols,
S. Hirata, M. T. Hackler, Y. Zhao, P.-D. Fan, R. J. Harrison,
M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan,
Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen,
E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao,
R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell,
D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan,
K. Dyall, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffe,
B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield,
X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing,
G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong,
and Z. Zhang,
"NWChem, A Computational Chemistry Package for Parallel Computers,
Version 5.1" (2007),
Pacific Northwest National Laboratory,
Richland, Washington 99352-0999, USA.
Total times cpu: 3.4s wall: 18.5s
```

sed One-liners

- `sed` one-liners: http://sed.sourceforge.net/sed1line.txt

awk

- The awk text-processing language is useful for such tasks as:
  - Tallying information from text files and creating reports from the results.
  - Adding additional functions to text editors like "vi".
  - Translating files from one format to another.
  - Creating small databases.
  - Performing mathematical operations on files of numeric data.
- awk has two faces:
  - It is a utility for performing simple text-processing tasks, and
  - It is a programming language for performing complex text-processing tasks.
- awk comes in three variations
  - awk : Original AWK by A. Aho, B. W. Kernighnan and P. Weinberger from AT&T
  - nawk : New AWK, also from AT&T
  - gawk : GNU AWK, all Linux distributions come with gawk. In some distros, awk is a symbolic link to gawk.

awk Syntax

- Simplest form of using `awk`
  - `awk pattern {action}`
    - `pattern` decides when `action` is performed
  - Most common action: `print`
  - Print file dosum.sh: `awk ’{print $0}’ dosum.sh`
  - Print line matching bash in all `.sh` files in current directory: `awk ’/bash/{print $0}’ *.sh`

awk Patterns

`BEGIN`	special pattern which is not tested against input. Action will be performed before reading input.
`END`	special pattern which is not tested against input. Action will be performed after reading all input.
`/regular expression/`	the associated regular expression is matched to each input line that is read
`relational expression`	used with the if, while relational operators
`&&`	logical AND operator used as pattern1 && pattern2. Execute action if pattern1 and pattern2 are true
`||`	logical OR operator used as pattern1 || pattern2. Execute action if either pattern1 or pattern2 is true
`!`	logical NOT operator used as !pattern. Execute action if pattern is not matched
`?:`	Used as pattern1 ? pattern2 : pattern3. If pattern1 is true use pattern2 for testing else use pattern3
`pattern1, pattern2`	Range pattern, match all records starting with record that matches pattern1 continuing until a record has been reached that matches pattern2

Awk Examples

- Print list of files that are csh script files

```
awk '/^#\!\/bin\/tcsh/{print FILENAME}' *

dooper.csh
factorial.csh
hello1.sh
name.csh
nestedloops.csh
quotes.csh
shift.csh
```

- Print contents of hello.sh that lie between two patterns

```
awk '/^#\!\/bin\/bash/,/echo/{print $0}' hello.sh

#!/bin/bash
# My First Script
echo "Hello World!"
```

How awk Works

- `awk` reads the file being processed line by line.
- The entire content of each line is split into columns with space or tab as the delimiter. The delimiter can be changed as will be seen in the next few slides.
- To print the entire line, use `$0`.
- The intrinsic variable NR contains the number of records (lines) read.
- The intrinsic variable NF contains the number of fields or columns in the current line.

Changing Field Delimiter

- By default the field delimiter is space or tab. To change the field delimiter use the `-F<delimiter>` command.

```
uptime

11:18am up 14 days 0:40, 5 users, load average: 0.15, 0.11, 0.17

uptime | awk '{print $1,NF}'

11:19am 0.17

uptime | awk -F: '{print $1,NF}'

11 0.12, 0.10, 0.16

for i in $(seq 1 10); do touch file${i}.dat ; done
ls file*

file10.dat file2.dat file4.dat file6.dat file8.dat
file1.dat file3.dat file5.dat file7.dat file9.dat

for i in file* ; do
> prefix=$(echo $i | awk -F. '{print $1}')
> suffix=$(echo $i | awk -F. '{print NF}')
> echo $prefix $suffix $i
> done

file10 dat file10.dat
file1 dat file1.dat
file2 dat file2.dat
file3 dat file3.dat
file4 dat file4.dat
file5 dat file5.dat
file6 dat file6.dat
file7 dat file7.dat
file8 dat file8.dat
file9 dat file9.dat
```

Formatting Output (1)

- Printing an expression is the most common action in the `awk` statement. If formatted output is required, use the `printf` format.
- The `print` command puts an explicit newline character at the end while the `printf` command does not.

```
echo hello 0.2485 5 | awk ’{printf “%s %f %d %05d\n”,$1,$2,$3,$3}’

hello 0.248500 5 00005
```

Formatting Output (2)

- Format specifiers are similar to the C-programming language

`%d,%i`	decimal number
`%e,%E`	floating point number of the form [-]d.dddddd.e[]dd. The %E format uses E instead of e.
`%f`	floating point number of the form [-]ddd.dddddd
`%g,%G`	Use %e or %f conversion with nonsignificant zeros truncated. The %G format uses %E instead of %e
`%s`	character string

Formatting Output (3)

- Format specifiers have additional parameter which may lie between the % and the control Letter

`0`	A leading 0 (zero) acts as a flag, that indicates output should be padded with zeroes instead of spaces.
`width`	The field should be padded to this width. The field is normally padded with spaces. If the 0 flag has been used, it is padded with zeroes.
`.prec`	A number that specifies the precision to use when printing.

- String constants supported by `awk`

`\\`	Literal backslash
`\n`	newline
`\r`	carriage-return
`\t`	horizontal tab
`\v`	vertical tab

awk Variables

- Like all progamming languages, awk supports the use of variables. Like shell, variable types do not have to be defined.
- awk variables can be user defined or could be one of the columns of the file being processed.
- Unlike Shell, awk variables are referenced as is i.e. no $ prepended to variable name (except for the special variables such as $1, $2 etc).

```
awk '{print $1}' hello.sh

#!/bin/bash
#
echo

awk '{col=$1;print col,$2}' hello.sh

#!/bin/bash
# My
echo "Hello
```

Conditionals and Loops (1)

- awk supports
  - `if ... else if .. else` conditionals.
  - `while` and `for` loops
- They work similar to that in C-programming
- Supported operators: ==, !=, >, >=, <, <=, ~ (string matches), !~ (string does not match)

```
awk '{if (NR > 0 ){print NR,":", $0}}' hello.sh

1 : #!/bin/bash
2 :
3 : # My First Script
4 :
5 : echo "Hello World!"
```

Conditionals and Loops (2)

- The `for` command can be used for processing the various columns of each line

```
cat << EOF | awk ’{for (i=1;i<=NF;i++){if (i==1){a=$i}else if (i==NF){print a}else{a+=$i}}}’

1 2 3 4 5 6
7 8 9 10
EOF
15
24

echo $(seq 1 10) | awk ’BEGIN{a=6}{for (i=1;i<=NF;i++){a+=$i}}END {print a}’

61
```

awk One-liners

- awk one-liners: http://www.pement.org/awk/awk1line.txt

The awk Programming Language

- awk can also be used as a programming language.
- The first line in awk scripts is the shebang line (#!) which indicates the location of the awk binary.
- To support scripting, awk has several built-in variables, which can also be used in one line commands
  - ARGC : number of command line arguments
  - ARGV : array of command line arguments
  - FILENAME : name of current input file
  - FS : field separator
  - OFS : output field separator
  - ORS : output record separator, default is newline
- awk permits the use of arrays
- awk has built-in functions to aid writing of scripts
- GNU awk also supports user defined function

Further Reading

- Bash Programming: http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
- Advanced Bash-Scripting Guide: http://tldp.org/LDP/abs/html/
- Regular Expressions: http://www.grymoire.com/Unix/Regular.html
- AWK Programming: http://www.grymoire.com/Unix/Awk.html
- awk one-liners: http://www.pement.org/awk/awk1line.txt
- sed: http://www.grymoire.com/Unix/Sed.html
- sed one-liners: http://sed.sourceforge.net/sed1line.txt
- Wiki Books: http://en.wikibooks.org/wiki/Subject:Computing

Getting Help

- User Guides
  - LSU HPC: http://www.hpc.lsu.edu/docs/guides.php#hpc
  - LONI:http://www.hpc.lsu.edu/docs/guides.php#loni
- Documentation: http://www.hpc.lsu.edu/docs
- Online courses: http://moodle.hpc.lsu.edu
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900
  - Instant Messenger (AIM, Yahoo Messenger, Google Talk)
    - Add “lsuhpchelp”

Questions?