Shell Scripting

Basic Commands & Environment

Alexander B. Pacheco
LTS Research Computing
March 19 & 26, 2015

Outline Day I

1. Overview of Introduction to Linux
2. Shell Scripting Basics
3. Beyond Basic Shell Scripting
4. Wrap Up
5. Hands-On Exercises: Day 1

Outline Day II

6 Regular Expressions

7 File Manipulation

8 grep

9 sed

10 awk

11 Wrap Up

Tutorial Outline

Day 1: Basic Shell Scripting

On the first day, we will cover simple topics such as creating and executing simple shell scripts, arithmetic operations, loops and conditionals, command line arguments and functions. 6

Day 2: Advanced Shell Scripting

On the second day, we will cover advanced topics such as creating shell scripts for data analysis which make use of tools such as regular expressions, grep, sed and the awk programming language.

Overview of Introduction to Linux

Introduction

What is a SHELL

- The command line interface is the primary interface to Linux/Unix operating systems.
- Shells are how command-line interfaces are implemented in Linux/Unix.
- Each shell has varying capabilities and features and the user should choose the shell that best suits their needs.
- The shell is simply an application running on top of the kernel and provides a powerful interface to the system.

Types 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 (sh).
- 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	✓	✓	✓	✓	✓
Shell Variables	✓	✓	✓	✓	✓
Command alias	✗	✓	✓	✓	✓
Command history	✗	✓	✓	✓	✓
Filename completion	✗	✯	✯	✓	✓
Command line editing	✗	✗	✯	✓	✓
Job control	✗	✓	✓	✓	✓

✓ : Yes

✗ : No

✯ : Yes, not set by default

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

Variables I

- A variable is a named object that contains data used by one or more applications.
- There are two types of variables, Environment and User Defined and can contain a number, character or a string of characters.
- Environment Variables provides a simple way to share configuration settings between multiple applications and processes in Linux.
- As in programming languages like C, C++ and Fortran, defining your own variables makes the program or script extensible by you or a third party
- Rules for Variable Names
  1. Variable names must start with a letter or underscore
  2. Number can be used anywhere else
  3. DO NOT USE special characters such as `@, #, %, $`
  4. Case sensitive
  5. Examples
     - Allowed: VARIABLE, VAR1234able, var_name, _VAR
     - Not Allowed: 1VARIABLE, %NAME, $myvar, VAR@NAME
- To reference a variable, environment or user defined, you need to prepend the variable name with "$" as in $VARIABLE, $PATH, etc.

Variables II

- Its a good practice to protect your variable name within {. . . } such as ${PATH} when referencing it. (We'll see an example in a few slides)
- Assigning value to a variable

Type	sh,ksh,bash	csh,tcsh
Shell	name=value	set name = value
Environment	export name=value	setenv name value

- **sh,ksh,bash** THERE IS NO SPACE ON EITHER SIDE OF =
- **csh,tcsh** space on either side of = is allowed for the `set` command
- **csh,tcsh** There is no = in the `setenv` command

File Permissions I

- In *NIX OS's, you have three types of file permissions
  1. read (r)
  2. write (w)
  3. execute (x)
- for three types of users
  1. user
  2. group
  3. world i.e. everyone else who has access to the system

```
drwxr-xr-x.   2   user   user   4096   Jan   28   08:27   Public
-rw-rw-r--.   1   user   user   3047   Jan   28   09:34   README
```

- The first character signifies the type of the file

  `d` for directory

  `l` for symbolic link

  `-` for normal file
- The next three characters of first triad signifies what the owner can do
- The second triad signifies what group member can do

File Permissions II

- The third triad signifies what everyone else can do

$$d\,\underbrace{rwx}_{u}\,\overbrace{r-x}^{g}\,\underbrace{r-x}_{o}$$

- Read carries a weight of 4
- Write carries a weight of 2
- Execute carries a weight of 1
- The weights are added to give a value of 7 (rwx), 6(rw), 5(rx) or 3(wx) permissions.
- `chmod` is a *NIX command to change permissions on a file
- To give user rwx, group rx and world x permission, the command is

```
chmod 751 filename
```

- Instead of using numerical permissions you can also use symbolic mode

u/g/o or a	user/group/world or all i.e. ugo
+/-	Add/remove permission
r/w/x	read/write/execute

File Permissions III

- Give everyone execute permission:

  ```
  chmod a+x hello.sh
  ```

  ```
  chmod ugo+x hello.sh
  ```

- Remove group and world read & write permission:

  ```
  chmod go-rw hello.sh
  ```

- Use the `-R` flag to change permissions recursively, all files and directories and their contents.

  ```
  chmod -R 755 ${HOME}/*
  ```

  What is the permission on ${HOME}?

Input/Output I

- The command `echo` is used for displaying output to screen
- For reading input from screen/keyboard/prompt

  bash `read`

  tcsh `$<`
- The `read` statement takes all characters typed until the Enter key is pressed and stores them into a variable.

  Syntax `read <variable name>`

  Example `read name` Enter

  Alex Pacheco
- `$<` can accept only one argument. If you have multiple arguments, enclose the `$<` within quotes e.g. `"$<"`

  Syntax: `set <variable> = $<`

  Example: `set name = "$<"` Enter

  Alex Pacheco
- In the above examples, the name that you enter in stored in the variable `name`.
- Use the `echo` command to print the variable `name` to the screen

Input/Output II

`echo $name` Enter

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

`echo Welcome to HPC    Training`↵ (more than one space between HPC and Training)

```
echo "Welcome to HPC    Training"↵
```

`read name`↵ or `set name = "$<"`↵

Alex   Pacheco↵

```
echo $name↵
```

```
echo "$name"↵
```

Input/Output III

- You can also use the **printf** command to display output

  Syntax: `printf <format> <arguments>`

  Example: `printf "$name"`↵

  `printf "%s\n" "$name"`↵

- Format Descriptors

%s	print argument as a string
%d	print argument as an integer
%f	print argument as a floating point number
\n	print new line
	you can add a width for the argument between the % and {s,d,f} fields %4s, %5d, %7.4f

- The **printf** command is used in **awk** to print formatted data (more on this later)

I/O Redirection

- There are three file descriptors for I/O streams
    1. STDIN: Standard Input
    2. STDOUT: Standard Output
    3. STDERR: Standard Error
- 1 represents STDOUT and 2 represents STDOUT
- 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.

- Examples:
    1. write STDOUT to file: `ls -l > ls-l.out`
    2. write STDERR to file: `ls -l 2> ls-l.err`
    3. write STDOUT to STDERR: `ls -l 1>&2`
    4. write STDERR to STDOUT: `ls -l 2>&1`
    5. send STDOUT as STDIN: `ls -l | wc -l`

Shell Scripting Basics

Start Up Scripts

- When you login to a *NIX computer, shell scripts are automatically loaded depending on your default `shell`
- **sh,ksh**
  1. `/etc/profile`
  2. `$HOME/.profile`
- **bash**
  1. `/etc/profile, login terminal only`
  2. `/etc/bashrc or /etc/bash/bashrc`
  3. `$HOME/.bash_profile, login terminal only`
  4. `$HOME/.bashrc`
- **csh,tcsh**
  1. `/etc/csh.cshrc`
  2. `$HOME/.tcshrc`
  3. `$HOME/.cshrc if .tcshrc is not present`
- The `.bashrc, .tcshrc, .cshrc, .bash_profile` are script files where users can define their own aliases, environment variables, modify paths etc.
- e.g. the `alias rm="rm -i"` command will modify all `rm` commands that you type as `rm -i`

Examples I

.bashrc

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
alias c="clear"
alias rm="/bin/rm -i"
alias psu="ps -u apacheco"
alias em="emacs -nw"
alias ll="ls -lF"
alias la="ls -al"
export PATH=/home/apacheco/bin:${PATH}
export g09root=/home/apacheco/Software/Gaussian09
export GAUSS_SCRDIR=/home/apacheco/Software/scratch
source $g09root/g09/bsd/g09.profile

export TEXINPUTS=.:/usr/share/texmf//:/home/apacheco/LaTeX//:${TEXINPUTS}
export BIBINPUTS=.:/home/apacheco/TeX//:${BIBINPUTS}
```

Examples II

.tcshrc

```
# .tcshrc

# User specific aliases and functions
alias c clear
alias rm "/bin/rm -i"
alias psu "ps -u apacheco"
alias em "emacs -nw"
alias ll "ls -lF"
alias la "ls -al"
setenv PATH "/home/apacheco/bin:${PATH}"
setenv g09root "/home/apacheco/Software/Gaussian09"
setenv GAUSS_SCRDIR "/home/apacheco/Software/scratch"
source $g09root/g09/bsd/g09.login

setenv TEXINPUTS ".:/usr/share/texmf//:/home/apacheco/LaTeX//:${TEXINPUTS}"
setenv BIBINPUTS ".:/home/apacheco/TeX//:${BIBINPUTS}"
```

What is a scripting Language?

- A **scripting language** or **script language** is a *programming language* that supports the writing of **scripts**.
- **Scripting Languages** provide a higher level of abstraction than standard programming languages.
- Compared to programming languages, scripting languages do not distinguish between data types: integers, real values, strings, etc.
- Scripting Languages tend to be good for automating the execution of other programs.
  - analyzing data
  - running daily backups
- They are also good for writing a program that is going to be used only once and then discarded.
- A **script** is a program written for a software environment that automate the execution of tasks which could alternatively be executed one-by-one by a human operator.
- The majority of script programs are "quick and dirty", where the main goal is to get the program written quickly.

Writing your first script

Three things to do to write and execute a script

1. Write a script
   - A shell script is a file that contains ASCII text.
   - Create a file, `hello.sh` with the following lines

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

2. Set permissions

```
~/Tutorials/BASH/scripts> chmod 755 hello.sh
```

OR

```
~/Tutorials/BASH/scripts> chmod a+x hello.sh
```

3. Execute the script

```
~/Tutorials/BASH/scripts>./hello.sh
Hello World!
```

4. If you do not set execute permission for the script, then

```
~/Tutorials/BASH/scripts> sh hello.sh
Hello World!
```

Description of the script

- My First Script

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

- The first line is called the "ShaBang" line. It tells the OS which interpreter to use. In the current example, bash
- Other options are:
  - sh : #!/bin/sh
  - ksh : #!/bin/ksh
  - csh : #!/bin/csh
  - tcsh: #!/bin/tcsh
- The second line is a comment. All comments begin with "#".
- The third line tells the OS to print "Hello World!" to the screen.

Special Characters

#: starts a comment.

$: indicates the name of a variable.

\: escape character to display next character literally.

{ }: used to enclose name of variable.

; Command separator [semicolon]. Permits putting two or more commands on the same line.

;; Terminator in a case option [double semicolon].

. "dot" command [period]. Equivalent to source. This is a bash builtin.

$? exit status variable.

$$ process ID variable.

[ ] test expression

[[ ]] test expression, more flexible than [ ]

$[ ], (( )) integer expansion

||, &&, ! Logical OR, AND and NOT

Quotation

- Double Quotation `"  "`
  - Enclosed string is expanded ("$", "/" and "‘")
  - Example: `echo "$myvar"` prints the value of `myvar`
- Single Quotation `’  ’`
  - Enclosed string is read literally
  - Example: `echo ’$myvar’` prints `$myvar`
- Back Quotation `‘  ‘`
  - Used for command substitution
  - Enclosed string is executed as a command
  - Example: `echo ‘pwd‘` prints the output of the `pwd` command i.e. print working directory
  - In **bash**, you can also use `$(···)` instead of `‘···‘`
    e.g. `$(pwd)` and `‘pwd‘` are the same

Example

```
#!/bin/bash

HI=Hello

echo HI           # displays HI
echo $HI          # displays Hello
echo \$HI         # displays $HI
echo "$HI"        # displays Hello
echo ’$HI’        # displays $HI
echo "$HIAlex"    # displays nothing
echo "${HI}Alex"  # displays HelloAlex
echo ‘pwd‘        # displays working directory
echo $(pwd)       # displays working directory
```

```
~/Tutorials/BASH/scripts/day1/examples>./quotes.sh
HI
Hello
$HI
Hello
$HI

HelloAlex
/home/apacheco/Tutorials/BASH/scripts/day1/examples
/home/apacheco/Tutorials/BASH/scripts/day1/examples
~/Tutorials/BASH/scripts/day1/examples>
```

Beyond Basic Shell Scripting

Arithmetic Operations I

- You can carry out numeric operations on integer variables

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

- Arithmetic operations in **bash** can be done within the `$((⋯))` or `$[⋯]` commands
  - ★ Add two numbers: `$((1+2))`
  - ★ Multiply two numbers: `$[$a*$b]`
  - ★ You can also use the `let` command: `let c=$a-$b`
  - ★ or use the `expr` command: `` c=`expr $a - $b` ``

Arithmetic Operations II

- In **tcsh**,
  - Add two numbers: `@ x = 1 + 2`
  - Divide two numbers: `@ x = $a / $b`
  - You can also use the `expr` command: `set c = ‘expr $a % $b‘`
- Note the use of space

bash space required around operator in the `expr` command

tcsh space required between `@` and variable, around = and numeric operators.

- You can also use C-style increment operators

bash `let c+=1` or `let c--`

tcsh `@ x -= 1` or `@ x++`

/=, ∗= and %= are also allowed.

bash

- The above examples only work for integers.
- What about floating point number?

Arithmetic Operations III

- Using floating point in **bash** or **tcsh** scripts requires an external calculator like 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"`
- You can also use **awk** for floating point arithmetic.

Arrays I

- **bash** and **tcsh** supports one-dimensional arrays.
- Array elements may be initialized with the `variable[xx]` notation

  `variable[xx]=1`
- Initialize an array during declaration

bash `name=(firstname ’last name’)`

tcsh `set name = (firstname ’last name’)`

- reference an element `i` of an array `name`

  `${name[i]}`
- print the whole array

bash `${name[@]}`

tcsh `${name}`

- print length of array

bash `${#name[@]}`

tcsh `${#name}`

Arrays II

- print length of element `i` of array `name`

  `${#name[i]}`

  Note: In **bash** `${#name}` prints the length of the first element of the array
- Add an element to an existing array

bash `name=(title ${name[@]})`

tcsh `set name = ( title "${name}")`

- In **tcsh** everything within "..." is one variable.
- In the above **tcsh** example, `title` is first element of new array while the second element is the old array `name`
- copy an array `name` to an array `user`

bash `user=(${name[@]})`

tcsh `set user = ( ${name} )`

Arrays III

- concatenate two arrays

bash `nameuser=(${name[@]} ${user[@]})`

tcsh `set nameuser=( ${name} ${user} )`

- delete an entire array

```
unset name
```

- remove an element i from an array

bash `unset name[i]`

tcsh

```
@ j = $i - 1
@ k =$i + 1
set name = ( ${name[1-$j]} ${name[$k-]})
```

bash the first array index is zero (0)

tcsh the first array index is one (1)

Arrays IV

name.sh

```bash
#!/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.csh

```
#!/bin/tcsh

echo "Print your first name"
set firstname = $<
echo "Print your last name"
set lastname = $<

set name = ( $firstname $lastname)
echo "Hello " ${name}

echo "Enter your salutation"
set title = $<

echo "Enter your suffix"
set suffix = "$<"

set name = ($title $name $suffix )
echo "Hello " ${name}

@ i = $#name
set name = ( $name[1-2] $name[4-$i] )
echo "Hello " ${name}
```

```
~/Tutorials/BASH/scripts/day1/examples>./name.sh
Print your first and last name
Alex Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
the first
Hello  Dr. Alex Pacheco the first
Hello  Dr. Alex the first
```

```
~/Tutorials/BASH/scripts/day1/examples>./name.csh
Print your first name
Alex
Print your last name
Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
the first
Hello  Dr. Alex Pacheco the first
Hello  Dr. Alex the first
```

Flow Control

- Shell Scripting Languages execute commands in sequence similar to programming languages such as C, Fortran, etc.
- Control constructs can change the sequential order of commands.
- Control constructs available in **bash** and **tcsh** are
  1. Conditionals: `if`
  2. Loops: `for`, `while`, `until`
  3. Switches: `case`, `switch`

if statement

- An `if/then` construct tests whether the exit status of a list of commands is 0, and if so, executes one or more commands.

bash

```
if [ condition1 ]; then
  some commands
elif [ condition2 ]; then
  some commands
else
  some commands
fi
```

tcsh

```
if ( condition1 ) then
  some commands
else if ( condition2 ) then
  some commands
else
  some commands
endif
```

- Note the space between *condition* and "["   "]"
- **bash** is very strict about spaces.
- **tcsh** commands are not so strict about spaces.
- **tcsh** uses the `if-then-else if-else-endif` similar to Fortran.

Integer & String Comparison Operators

Integer Comparison

Operation	**bash**	**tcsh**
equal to	`if [ 1 -eq 2 ]`	`if (1 == 2)`
not equal to	`if [ $a -ne $b ]`	`if ($a != $b)`
greater than	`if [ $a -gt $b ]`	`if ($a > $b)`
greater than or equal to	`if [ 1 -ge $b ]`	`if (1 >= $b)`
less than	`if [ $a -lt 2 ]`	`if ($a < 2)`
less than or equal to	`if [[ $a -le $b ]]`	`if ($a <= $b)`

String Comparison

Operation	**bash**	**tcsh**
equal to	`if [ $a == $b ]`	`if ($a == $b)`
not equal to	`if [ $a != $b ]`	`if ($a != $b)`
zero length or null	`if [ -z $a ]`	`if ($%a == 0)`
non zero length	`if [ -n $a ]`	`if ($%a > 0)`

File Test & Logical Operators

File Test Operators

Operation	**bash**	**tcsh**
file exists	if [-e .bashrc]	if (-e .tcshrc)
file is a regular file	if [-f .bashrc]	
file is a directory	if [-d /home]	if (-d /home)
file is not zero size	if [-s .bashrc]	if (! -z .tcshrc)
file has read permission	if [-r .bashrc]	if (-r .tcshrc)
file has write permission	if [-w .bashrc]	if (-w .tcshrc)
file has execute permission	if [-x .bashrc]	if (-x .tcshrc)

Logical Operators

Operation	**bash**	**tcsh**
NOT	if [! -e .bashrc]	if (! -z .tcshrc)
AND	if [$a -eq 2] && [$x -gt $y]	if ($a == 2 && $x <= $y)
OR	if [[$a -eq 2 || $x -gt $y]]	if ($a == 2 || $x <= $y)

Examples

- Condition tests using the `if/then` may be nested

```
read a
if [ "$a" -gt 0 ]; then
  if [ "$a" -lt 5 ]; then
    echo "The value of \"a\" lies somewhere between 0
           and 5"
  fi
fi
```

```
set a = $<
if ( $a > 0 ) then
  if ( $a < 5 ) then
    echo "The value of $a lies somewhere between
           0 and 5"
  endif
endif
```

- This is same as

```
read a
if [[ "$a" -gt 0 &&  "$a" -lt 5 ]]; then
  echo "The value of $a lies somewhere between 0 and
         5"
fi
OR
if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]; then
  echo "The value of $a lies somewhere between 0 and
         5"
fi
```

```
set a = $<
if ( "$a" > 0  &&  "$a" < 5 ) then
  echo "The value of $a lies somewhere between 0
         and 5"
endif
```

Loop Constructs

- A *loop* is a block of code that iterates a list of commands as long as the *loop control condition* is true.
- Loop constructs available in

bash: `for, while` and `until`

tcsh: `foreach` and `while`

bash: for loops

- 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 (( EXP1; EXP2; EXP3 )); do
  some commands
done
```

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

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

```
for ((i=1;i<=10;i++))
do
  touch file${i}.dat
done
```

tcsh: foreach loop

- The `foreach` loop is the basic looping construct in **tcsh**

```
foreach arg (list)
  some commands
end
```

```
foreach i (‘seq 1 10‘)
  touch file$i.dat
end
```

while Construct

- The while construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status).
- In contrast to a for loop, a while loop finds use in situations where the number of loop repetitions is not known beforehand.

bash

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

tcsh

```
while ( condition )
  some commands
end
```

factorial.sh

```
#!/bin/bash

echo -n "Enter a number less than 10: "
read counter
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial
```

factorial.csh

```
#!/bin/tcsh

echo -n "Enter a number less than 10: "
set counter = $<
set factorial = 1
while ( $counter > 0 )
   @ factorial = $factorial * $counter
   @ counter -= 1
end
echo $factorial
```

until Contruct (bash only)

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

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

factorial2.sh

```
#!/bin/bash

echo -n "Enter a number less than 10: "
read counter
factorial=1
until [ $counter -le 1 ]; do
  factorial=$[ $factorial * $counter ]
  if [ $counter -eq 2 ]; then
    break
  else
    let counter-=2
  fi
done
echo $factorial
```

Nested Loops

- for, while & until loops can nested. To exit from the loop use the break command

nestedloops.sh

```bash
#!/bin/bash

## Example of Nested loops

echo "Nested for loops"
for a in $(seq 1 5) ; do
  echo "Value of a in outer loop:" $a
  for b in `seq 1 2 5` ; do
    c=$(($a*$b))
    if [ $c -lt 10 ]; then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 10"
      break
    fi
  done
done
echo "========================"
echo
echo "Nested for and while loops"
for ((a=1;a<=5;a++)); do
  echo "Value of a in outer loop:" $a
  b=1
  while [ $b -le 5 ]; do
    c=$(($a*$b))
    if [ $c -lt 5 ]; then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 5"
      break
    fi
    let b+=2
  done
done
echo "========================"
```

nestedloops.csh

```csh
#!/bin/tcsh

## Example of Nested loops

echo "Nested for loops"
foreach a (`seq 1 5`)
  echo "Value of a in outer loop:" $a
  foreach b (`seq 1 2 5`)
    @ c = $a * $b
    if ( $c <  10 ) then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 10"
      break
    endif
  end
end
echo "========================"
echo
echo "Nested for and while loops"
foreach a (`seq 1 5`)
  echo "Value of a in outer loop:" $a
  set b = 1
  while ( $b <= 5 )
    @ c = $a * $b
    if ( $c < 5 ) then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 5"
      break
    endif
    @ b = $b + 2
  end
end
echo "========================"
```

Switching or Branching Constructs I

- 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 command
    ;;
  "condition2")
    some other command
    ;;
esac
```

select construct

```
select variable [ list ]
do
  command
  break
done
```

Switching or Branching Constructs II

- `tcsh` has the `switch` construct

switch construct

```
switch (arg list)
  case "variable"
    some command
    breaksw
endsw
```

dooper.sh

```
#!/bin/bash

echo "Print two numbers"
read num1 num2
echo "What operation do you want to do?"

operations='add subtract multiply divide exponentiate
      modulo all quit'
select oper in $operations ; do
  case $oper in
    "add")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      ;;
    "subtract")
      echo "$num1 - $num2 =" $[$num1 - $num2]
      ;;
    "multiply")
      echo "$num1 * $num2 =" $[$num1 * $num2]
      ;;
    "exponentiate")
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      ;;
    "divide")
      echo "$num1 / $num2 =" $[$num1 / $num2]
      ;;
    "modulo")
      echo "$num1 % $num2 =" $[$num1 % $num2]
      ;;
    "all")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      echo "$num1 - $num2 =" $[$num1 - $num2]
      echo "$num1 * $num2 =" $[$num1 * $num2]
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      echo "$num1 / $num2 =" $[$num1 / $num2]
      echo "$num1 % $num2 =" $[$num1 % $num2]
    ;;
    *)
      exit
      ;;
  esac
done
```

dooper.csh

```
#!/bin/tcsh

echo "Print two numbers one at a time"
set num1 = $<
set num2 = $<
echo "What operation do you want to do?"
echo "Enter +, -, x, /, % or all"
set oper = $<

switch ( $oper )
  case "x"
     @ prod = $num1 * $num2
     echo "$num1 * $num2 = $prod"
     breaksw
  case "all"
     @ sum = $num1 + $num2
     echo "$num1 + $num2 = $sum"
     @ diff = $num1 - $num2
     echo "$num1 - $num2 = $diff"
     @ prod = $num1 * $num2
     echo "$num1 * $num2 = $prod"
     @ ratio = $num1 / $num2
     echo "$num1 / $num2 = $ratio"
     @ remain = $num1 % $num2
     echo "$num1 % $num2 = $remain"
     breaksw
  case "*"
     @ result = $num1 $oper $num2
     echo "$num1 $oper $num2 = $result"
     breaksw
endsw
```

```
~/Tutorials/BASH/scripts>./day1/examples/dooper.sh
Print two numbers
1 4
What operation do you want to do?
1) add 3) multiply  5) exponentiate  7) all
2) subtract 4) divide  6) modulo   8) quit
#? 7
1 + 4 = 5
1 - 4 = -3
1 * 4 = 4
1 ** 4 = 1
1 / 4 = 0
1 % 4 = 1
#? 8
```

```
~/Tutorials/BASH/scripts>./day1/examples/dooper.csh
Print two numbers one at a time
1
5
What operation do you want to do?
Enter +, -, x, /, % or all
all
1 + 5 = 6
1 - 5 = -4
1 * 5 = 5
1 / 5 = 0
1 % 5 = 1
```

dooper1.sh

```bash
#!/bin/bash

echo "Print two numbers"
read num1 num2
echo "What operation do you want to do?"
echo "Options are add, subtract, multiply,
     exponentiate, divide, modulo and all"
read oper

  case $oper in
    "add")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      ;;
    "subtract")
      echo "$num1 - $num2 =" $[$num1 - $num2]
      ;;
    "multiply")
      echo "$num1 * $num2 =" $[$num1 * $num2]
      ;;
    "exponentiate")
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      ;;
    "divide")
      echo "$num1 / $num2 =" $[$num1 / $num2]
      ;;
    "modulo")
      echo "$num1 % $num2 =" $[$num1 % $num2]
      ;;
    "all")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      echo "$num1 - $num2 =" $[$num1 - $num2]
      echo "$num1 * $num2 =" $[$num1 * $num2]
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      echo "$num1 / $num2 =" $[$num1 / $num2]
      echo "$num1 % $num2 =" $[$num1 % $num2]
    ;;
    *)
      exit
      ;;
  esac
```

```
~/Tutorials/BASH/scripts>./day1/examples/dooper1.sh
Print two numbers
2 5
What operation do you want to do?
Options are add, subtract, multiply, exponentiate,
      divide, modulo and all
all
2 + 5 = 7
2 - 5 = -3
2 * 5 = 10
2 ** 5 = 32
2 / 5 = 0
2 % 5 = 2
```

Command Line Arguments

- Similar to programming languages, bash (and other shell scripting languages) can also take command line arguments
  - ./scriptname arg1 arg2 arg3 arg4 ...
  - $0,$1,$2,$3, etc: positional parameters corresponding to ./scriptname,arg1,arg2,arg3,arg4,... 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
- In csh,tcsh
  - an array argv contains the list of arguments with argv[0] set to name of script.
  - #argv is the number of arguments i.e. length of argv array.

shift.sh

```bash
#!/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
```

```
dyn100085:examples apacheco$./shift.sh $(seq 1 5)
Number of Arguments:  5
List of Arguments:  1 2 3 4 5
Name of script that you are running: ./shift.sh
Command You Entered:./shift.sh 1 2 3 4 5
Argument List is:  1 2 3 4 5
Number of Arguments:  5
Argument List is:  2 3 4 5
Number of Arguments:  4
Argument List is:  3 4 5
Number of Arguments:  3
Argument List is:  4 5
Number of Arguments:  2
Argument List is:  5
Number of Arguments:  1
```

shift.csh

```tcsh
#!/bin/tcsh

set USAGE="USAGE: $0 <at least 1 argument>"

if ( "$#argv" < 1 ) then
   echo $USAGE
   exit
endif

echo "Number of Arguments: " $#argv
echo "List of Arguments: " ${argv}
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 ${argv}

while ( "$#argv"  > 0 )
  echo "Argument List is: " $*
  echo "Number of Arguments: " $#argv
  shift
end
```

```
dyn100085:examples apacheco$./shift.csh $(seq 1 5)
Number of Arguments:  5
List of Arguments:  1 2 3 4 5
Name of script that you are running: ./shift.csh
Command You Entered:./shift.csh 1 2 3 4 5
Argument List is:  1 2 3 4 5
Number of Arguments:  5
Argument List is:  2 3 4 5
Number of Arguments:  4
Argument List is:  3 4 5
Number of Arguments:  3
Argument List is:  4 5
Number of Arguments:  2
Argument List is:  5
Number of Arguments:  1
```

Declare command

- Use the **declare** command to set variable and functions attributes.
- Create a constant variable i.e. read only variable

  Syntax:

  `declare -r var`

  `declare -r varName=value`
- Create an integer variable

  Syntax:

  `declare -i var`

  `declare -i varName=value`
- You can carry out arithmetic operations on variables declared as integers

```
~/Tutorials/BASH> j=10/5 ; echo $j
10/5
~/Tutorials/BASH> declare -i j; j=10/5 ; echo $j
2
```

Functions I

- Like ”real” programming languages, **bash** has functions.
- A function is a subroutine, a code block that implements a set of operations, a ”black box” that performs a specified task.
- Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

```
function function_name {
  command
}
OR
function_name () {
  command
}
```

Functions II

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: " $#
```

```
dyn100085:examples apacheco$./shift10.sh
USAGE:./shift10.sh [atleast 11 arguments]
dyn100085:examples apacheco$./shift10.sh $(seq 1 10)
USAGE:./shift10.sh [atleast 11 arguments]
dyn100085:examples apacheco$./shift10.sh ‘seq 1 2 22‘
Number of Arguments:  11
List of Arguments:  1 3 5 7 9 11 13 15 17 19 21
Name of script that you are running: ./shift10.sh
Command You Entered:./shift10.sh 1 3 5 7 9 11 13 15 17 19
      21
First Argument 1
Tenth and Eleventh argument 10 11 19 21
Argument List is:  1 3 5 7 9 11 13 15 17 19 21
Number of Arguments:  11
Argument List is:  19 21
Number of Arguments:  2
dyn100085:examples apacheco$./shift10.sh $(seq 21 2 44)
Number of Arguments:  12
List of Arguments:  21 23 25 27 29 31 33 35 37 39 41 43
Name of script that you are running: ./shift10.sh
Command You Entered:./shift10.sh 21 23 25 27 29 31 33 35
      37 39 41 43
First Argument 21
Tenth and Eleventh argument 210 211 39 41
Argument List is:  21 23 25 27 29 31 33 35 37 39 41 43
Number of Arguments:  12
Argument List is:  39 41 43
Number of Arguments:  3
```

Functions III

- You can also pass arguments to a function.
- All function parameters or arguments can be accessed via $1, $2, $3,..., $N.
- $0 always point to the shell script name.
- $* or $@ holds all parameters or arguments passed to the function.
- $# holds the number of positional parameters passed to the function.
- 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

  Syntax:

  ```
  local var=value
  ```

  ```
  local varName
  ```

Functions IV

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

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
```

```
dyn100085:examples apacheco$ ./factorial3.sh $(seq 1 2 11)
Factorial of 1 is 1
Factorial of 3 is 6
Factorial of 5 is 120
Factorial of 7 is 5040
Factorial of 9 is 362880
Factorial of 11 is 39916800
```

Scripting for Job Submission

Problem Description

- I have to run more than one serial job.
  - Solution: Create a script that will submit and run multiple serial jobs.
- I don't want to submit multiple jobs using the serial queue since
  - Cluster Admins give lower priority to jobs that are not parallelized
  - The number of jobs that I want to run exceed the maximum number of jobs that I can run simultaneously
- How do I submit *one* job which can run multiple serial jobs?

One Solution of many

- Write a script which will log into all unique nodes and run your serial jobs in background.
- Easy said than done
- What do you need to know?
  1. Shell Scripting
  2. How to run a job in background
  3. Know what the `wait` command does

```
[alp514@corona1 ~]$ cat checknodes.pbs
#!/bin/bash
#
#PBS -q normal
#PBS -l nodes=4:ppn=16
#PBS -l walltime=00:30:00
#PBS -V
#PBS -o nodetest.out
#PBS -e nodetest.err
#PBS -N testing
#PBS -M alp514@lehigh.edu
#PBS -m abe
#

export WORK_DIR=$PBS_O_WORKDIR
export NPROCS=‘wc -l $PBS_NODEFILE |gawk ’//{print $1}’‘

NODES=(‘cat ‘‘$PBS_NODEFILE’’‘ )
UNODES=(‘uniq ‘‘$PBS_NODEFILE’’‘ )

echo ‘‘Nodes Available: ‘‘ ${NODES[@]}
echo ‘‘Unique Nodes Available: ‘‘ ${UNODES[@]}

echo ‘‘Get Hostnames for all processes’’
i=0
for nodes in ‘‘${NODES[@]}’’; do
  ssh -n $nodes ’echo $HOSTNAME ’$i’ ’ &
  let i=i+1
done
wait

echo ‘‘Get Hostnames for all unique nodes’’
i=0
NPROCS=‘uniq $PBS_NODEFILE | wc -l |gawk ’//{print $1}’‘
let NPROCS-=1
while [ $i -le $NPROCS ] ; do
  ssh -n ${UNODES[$i]} ’echo $HOSTNAME ’$i’ ’
  let i=i+1
done

[alp514@corona1 ~]$ qsub checknodes.pbs
688825.corona1.cc.lehigh.edu
```

```
[alp514@corona1 ~]$ cat nodetest.out
Wed Mar 11 08:20:40 EDT 2015 : erasing contents of corona63:/scratch
Wed Mar 11 08:20:40 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:41 EDT 2015 : swap reset
Wed Mar 11 08:20:41 EDT 2015 : erasing contents of corona56:/scratch
Wed Mar 11 08:20:41 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:41 EDT 2015 : swap reset
Wed Mar 11 08:20:41 EDT 2015 : erasing contents of corona50:/scratch
Wed Mar 11 08:20:42 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:42 EDT 2015 : swap reset
Wed Mar 11 08:20:43 EDT 2015 : erasing contents of corona27:/scratch
Wed Mar 11 08:20:43 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:43 EDT 2015 : swap reset
Nodes Available:  corona63 corona63 corona63 corona63 corona63 corona63 corona63 corona63 corona63 corona63
       corona63 corona63 corona63 corona63 corona63 corona63 corona56 corona56 c
orona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56
      corona56 corona56 corona50 corona50 corona50 corona50 corona50 corona50 co
rona50 corona50 corona50 corona50 corona50 corona50 corona50 corona50 corona50 corona50 corona27 corona27
      corona27 corona27 corona27 corona27 corona27 corona27 corona27 corona27 cor
ona27 corona27 corona27 corona27 corona27 corona27
Unique Nodes Available:  corona63 corona56 corona50 corona27
Get Hostnames for all processes
corona27 52
corona27 59
corona27 60
corona27 57
corona27 51
corona27 62
corona27 54
corona27 48
corona27 63
corona27 58
corona27 53
corona50 43
```

```
corona50 40
corona50 38
corona50 33
corona50 34
corona50 47
corona56 31
corona63 13
corona63 6
corona56 22
corona63 9
corona63 14
corona56 16
corona56 25
corona56 23
corona56 17
corona63 5
corona63 10
corona63 8
corona63 15
corona63 3
corona50 32
corona50 44
corona56 18
corona50 36
corona50 46
corona56 27
corona50 42
corona63 1
corona63 12
corona50 45
corona50 41
corona50 35
corona56 29
corona63 7
corona56 28
corona63 11
corona56 26
corona56 21
corona63 4
```

```
corona56 24
corona63 0
corona56 20
Get Hostnames for all unique nodes
corona63 0
corona56 1
corona50 2
corona27 3
```

Wrap Up

References & Further Reading

- BASH Programming `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`
- CSH Programming `http://www.grymoire.com/Unix/Csh.html`
- csh Programming Considered Harmful `http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/`
- Wiki Books `http://en.wikibooks.org/wiki/Subject:Computing`

Hands-On Exercises: Day 1

Exercises

1. Create shell scripts to do the following
   - Write a simple hello world script
   - Modify the above script to use a variable
   - Modify the above script to prompt you for your name and then display your name with a greeting.
2. Write a script to add/subtract/multiply/divide two numbers.
3. Write a script to read your first and last name to an array.
   - Add your salutation and suffix to the array.
   - Drop either the salutation or suffix.
   - Print the array after each of the three steps above.
4. Write a script to calculate the factorial and double factorial of an integer or list of integers.

Solution 1

hellovariable.sh

```bash
#!/bin/bash

# Hello World script using a variable
STR="Hello World!"
echo $STR
```

```
~/Tutorials/BASH/scripts/day1/solution> ./
      hellovariable.sh
Hello World!
```

helloname.sh

```bash
#!/bin/bash

# My Second Script

echo Please Enter your name:
read name1 name2
Greet="Welcome to HPC Training"
echo "Hello $name1 $name2, $Greet"
```

```
~/Tutorials/BASH/scripts/day1/solution> ./
      helloname.sh
Please Enter your name:
Alex Pacheco
Hello Alex Pacheco, Welcome to HPC Training
```

Solution 2

dosum.sh

```
#!/bin/bash

echo "Enter two integers"
read num1 num2

echo "$num1 + $num2 = " $num1 + $num2
echo "$num1 + $num2 = " $(($num1 + $num2))

let SUM=$num1+$num2
echo "sum of $num1 & $num2 is " $SUM

echo "$num1/$num2 = " $(echo "scale=5;$num1/
      $num2" | bc)
echo "$num2/$num1 = " $(bc -l <<< $num2/$num1)

exit
```

```
~/Tutorials/BASH/scripts/day1/solution> ./dosum.
      sh
Enter two integers
5 7
5 + 7 =  5 + 7
5 + 7 =  12
sum of 5 & 7 is  12
5/7 =  .71428
7/5 =  1.40000000000000000000
```

doratio.csh

```
#!/bin/tcsh

echo "Enter first integer"
set num1 = $<
set num2 = $<

echo "$num1 / $num2 = " $num1 / $num2

@ RATIO = $num1 / $num2
echo "ratio of $num1 & $num2 is " $RATIO

set ratio=‘echo "scale=5 ; $num1/$num2" | bc‘
echo "ratio of $num1 & $num2 is " $ratio

exit
```

```
~/Tutorials/BASH/scripts/day1/solution> ./
      doratio.csh
Enter first integer
5
7
5 / 7 =  5 / 7
ratio of 5 & 7 is  0
ratio of 5 & 7 is  .71428
```

Alternate Solution 2

```
#!/bin/bash

echo "Print two numbers"
read num1 num2
echo "What operation do you want to do?"

operations=’add subtract multiply divide exponentiate
       modulo all quit’
select oper in $operations ; do
  case $oper in
    "add")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      ;;
    "subtract")
      echo "$num1 - $num2 =" $[$num1 - $num2]
      ;;
    "multiply")
      echo "$num1 * $num2 =" $[$num1 * $num2]
      ;;
    "exponentiate")
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      ;;
    "divide")
      echo "$num1 / $num2 =" $[$num1 / $num2]
      ;;
    "modulo")
      echo "$num1 % $num2 =" $[$num1 % $num2]
      ;;
    "all")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      echo "$num1 - $num2 =" $[$num1 - $num2]
      echo "$num1 * $num2 =" $[$num1 * $num2]
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      echo "$num1 / $num2 =" $[$num1 / $num2]
      echo "$num1 % $num2 =" $[$num1 % $num2]
    ;;
    *)
      exit
      ;;
  esac
done
```

```
#!/bin/tcsh

echo "Print two numbers one at a time"
set num1 = $<
set num2 = $<
echo "What operation do you want to do?"
echo "Enter +, -, x, /, % or all"
set oper = $<

switch ( $oper )
  case "x"
     @ prod = $num1 * $num2
     echo "$num1 * $num2 = $prod"
     breaksw
  case "all"
     @ sum = $num1 + $num2
     echo "$num1 + $num2 = $sum"
     @ diff = $num1 - $num2
     echo "$num1 - $num2 = $diff"
     @ prod = $num1 * $num2
     echo "$num1 * $num2 = $prod"
     @ ratio = $num1 / $num2
     echo "$num1 / $num2 = $ratio"
     @ remain = $num1 % $num2
     echo "$num1 % $num2 = $remain"
     breaksw
  case "*"
     @ result = $num1 $oper $num2
     echo "$num1 $oper $num2 = $result"
     breaksw
endsw
```

Solution 3

name.sh

```bash
#!/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.csh

```
#!/bin/tcsh

echo "Print your first name"
set firstname = $<
echo "Print your last name"
set lastname = $<

set name = ( $firstname $lastname)
echo "Hello " ${name}

echo "Enter your salutation"
set title = $<

echo "Enter your suffix"
set suffix = "$<"

set name = ($title $name $suffix )
echo "Hello " ${name}

@ i = $#name
set name = ( $name[1-2] $name[4-$i] )
echo "Hello " ${name}
```

```
~/Tutorials/BASH/scripts/day1/solution> ./name.
      sh
Print your first and last name
Alex Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
the first
Hello  Dr. Alex Pacheco the first
Hello  Dr. Alex the first
```

```
~/Tutorials/BASH/scripts/day1/solution> ./name.
      csh
Print your first name
Alex
Print your last name
Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
the first
Hello  Dr. Alex Pacheco the first
Hello  Dr. Alex the first
```

Solution 4

fac2.sh

```
#!/bin/bash

echo "Enter the integer whose factorial and
      double factorial you want to calculate"
read counter
factorial=1
i=$counter
while [ $i -gt 1 ]; do
  factorial=$[ $factorial * $i ]
  let i-=1
done

i=$counter
dfactorial=1
until [ $i -le 2 ]; do
  dfactorial=$[ $dfactorial * $i ]
  let i-=2
done

echo "$counter! = $factorial & $counter!! =
      $dfactorial"
```

fac2.csh

```
#!/bin/tcsh

echo "Enter the integer whose factorial and
      double factorial you want to calculate"
set counter = $<
@ factorial = 1
@ i = $counter
while ( $i > 1 )
  @ factorial = $factorial * $i
  @ i--
end

@ i = $counter
@ dfactorial = 1
while ( $i >= 1 )
  @ dfactorial = $dfactorial * $i
  @ i = $i - 2
end

echo "$counter\! = $factorial & $counter\!\! =
      $dfactorial"
```

fac3.sh: Alternate Solution 4 (bash only)

```bash
#!/bin/bash

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

factorial() {
  local i=$1
  local f
  local type=$2

  declare -i i
  declare -i f

  if [[ "$i" -le 2 && "$i" -ne 0 ]]; then
    echo $i
  elif [[ "$i" -eq 0 ]]; then
    echo 1
  else
    case $type in
      "single")
        f=$(( $i - 1 ))
        ;;
      "double")
        f=$(( $i - 2 ))
        ;;
    esac
    f=$( factorial $f $type)
    f=$(( $f * $i ))
    echo $f
  fi
}

if [[ "$#" -eq 0 ]]; then
  usage
else
  for i in $@ ; do
    x=$( factorial $i single )
    y=$( factorial $i double )
    echo "$i! = $x & $i!! = $y"
  done
fi
```

Regular Expressions

Regular Expressions

- A regular expression (regex) is a method of representing a string matching pattern.
- Regular expressions enable strings that match a particular pattern within textual data records to be located and modified and they are often used within utility programs and programming languages that manipulate textual data.
- Regular expressions are extremely powerful.
- Supporting Software and Tools
  1. Command Line Tools: grep, egrep, sed
  2. Editors: ed, vi, emacs
  3. Languages: awk, perl, python, php, ruby, tcl, java, javascript, .NET

Shell Regular Expressions

- The Unix shell recognises a limited form of regular expressions used with filename substitution

? : match any single character.

∗ : match zero or more characters.

[ ] : match list of characters in the list specified

[!] : match characters not in the list specified

- Examples:

```
1. ls *
2. cp [a-z]* lower/
3. cp [!a-z]* upper_digit/
```

POSIX Regular Expressions I

[ ] : A 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].

[ˆ] : Matches a single character that is not contained within the brackets.

For example, [ˆabc] matches any character other than ”a”, ”b”, or ”c”.

[ˆa-z] matches any single character that is not a lowercase letter from ”a” to ”z”.

( ) : Defines a marked subexpression.

The string matched within the parentheses can be recalled later.

A marked subexpression is also called a block or capturing group

| : The choice (or set union) operator: match either the expression before or the expression after the operator

For example, ”abc|def” matches ”abc” or ”def”.

POSIX Regular Expressions II

. : Matches any single character.

For example, a.c matches "abc", etc.

∗ : Matches the preceding element zero or more times.

For example, ab*c matches "ac", "abc", "abbbc", etc.

[xyz]* matches ", "x", "y", "z", "zx", "zyx", "xyzzy", and so on.

(ab)* matches "", "ab", "abab", "ababab", and so on.

{m,n} : Matches the preceding element at least m and not more than n times.

{m,} : Matches the preceding element at least m times.

{n} : Matches the preceding element exactly n times.

For example, a{3,5} matches only "aaa", "aaaa", and "aaaaa".

+ : Match the last "block" one or more times

For example, "ba+" matches "ba", "baa", "baaa" and so on

? : Match the last "block" zero or one times

For example, "ba?" matches "b" or "ba"

POSIX Regular Expressions III

^ : Matches the starting position within the string. In line-based tools, it matches the starting position of any line.

$: Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.

\s : Matches any whitespace.

\S : Matches any non-whitespace.

\d : Matches any digit.

\D : Matches any non-digit.

\w : Matches any word.

\W : Matches any non-word.

\b : Matches any word boundary.

\B : Matches any non-word boundary.

File Manipulation

Linux cut command

- Linux command cut is used for text processing to extract portion of text from a file by selecting columns.
- Usage: `cut <options> <filename>`
- Common Options:

  `-c list` : The list specifies character positions.
  `-b list` : The list specifies byte positions.
  `-f list` : select only these fields.
  `-d delim` : Use delim as the field delimiter character instead of the tab character.

- list is made up of one range, or many ranges separated by commas

  `N` : Nth byte, character or field. count begins from 1
  `N-` : Nth byte, character or field to end of line
  `N-M` : Nth to Mth (included) byte, character or field
  `-M` : from first to Mth (included) byte, character or field

```
~/Tutorials/BASH/scripts/day1/examples> uptime
 14:17pm  up 14 days  3:39,  5 users,  load average: 0.51, 0.22, 0.20
~/Tutorials/BASH/scripts/day1/examples> uptime | cut -c-8
 14:17pm
~/Tutorials/BASH/scripts/day1/examples> uptime | cut -c14-20
14 days
~/Tutorials/BASH/scripts/day1/examples> uptime | cut -d’’:’’ -f4
 0.41, 0.22, 0.20
```

paste

- The paste utility concatenates the corresponding lines of the given input files, replacing all but the last file's newline characters with a single tab character, and writes the resulting lines to standard output.

  If end-of-file is reached on an input file while other input files still contain data, the file is treated as if it were an endless source of empty lines.

- Usage: `paste <option> <files>`
- Common Options

  `-d delimiters` specifies a list of delimiters to be used instead of tabs for separating consecutive values on a single line. Each delimiter is used in turn; when the list has been exhausted, paste begins again at the first delimiter.

  `-s` causes paste to append the data in serial rather than in parallel; that is, in a horizontal rather than vertical fashion.

- Example

```
> cat names.txt
Mark Smith
Bobby Brown
Sue Miller
Jenny Igotit
```

```
> cat numbers.txt
555-1234
555-9876
555-6743
867-5309
```

```
> paste names.txt numbers.txt
Mark Smith      555-1234
Bobby Brown     555-9876
Sue Miller      555-6743
Jenny Igotit    867-5309
```

split

- split is a Unix utility most commonly used to split a file into two or more smaller files.
- Usage: `split <options> <file to be split> <name>`
- `Common Options`:

  `-a suffix_length`: Use suffix_length letters to form the suffix of the file name.

  `-b byte_count[k|m]`: Create smaller files byte_count bytes in length.

  If "k" is appended to the number, the file is split into byte_count kilobyte pieces.
  If "m" is appended to the number, the file is split into byte_count megabyte pieces.

  `-l n`: (Lowercase L) Create smaller files n lines in length.
- The default behavior of split is to generate output files of a fixed size, default 1000 lines.
- The files are named by appending aa, ab, ac, etc. to output filename.
- If output filename (`<name>`) is not given, the default filename of x is used, for example, xaa, xab, etc

csplit

- The csplit command in Unix is a utility that is used to split a file into two or more smaller files determined by context lines.
- Usage: `csplit <options> <file> <args>`
- Common Options:

  `-f prefix`: Give created files names beginning with prefix. The default is "xx".

  `-k`: Do not remove output files if an error occurs or a HUP, INT or TERM signal is received.

  `-s`: Do not write the size of each output file to standard output as it is created.

  `-n number`: Use number of decimal digits after the prefix to form the file name. The default is 2.
- The args operands may be a combination of the following patterns:

  `/regexp/[[+|-]offset]`: Create a file containing the input from the current line to (but not including) the next line matching the given basic regular expression. An optional offset from the line that matched may be specified.

  `%regexp%[[+|-]offset]`: Same as above but a file is not created for the output.

  `line_no`: Create containing the input from the current line to (but not including) the specified line number.

  `{num}`: Repeat the previous pattern the specified number of times. If it follows a line number pattern, a new file will be created for each line_no lines, num times. The first line of the file is line number 1 for historic reasons.

split & csplit examples

- Example: Run a multi-step job using Gaussian 09, for example geometry optimization followed by frequency analysis of water molecule.
- Problem: Some visualization packages like molden cannot visualize such multi-step jobs. Each job needs to visualized separetly.
- Solution: Split the single output file into two files, one for the optimization calculation and the other for frequency calculation.
- Source Files in scripts/day2/examples/h2o-opt-freq.log (Google Drive Downloads).
- Examples:

```
split -l 1442 h2o-opt-freq.log
```

```
csplit h2o-opt-freq.log "/Normal termination of Gaussian 09/+1"
```

grep

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

  `-i` : ignore case during search
  `-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.

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

```
~/Tutorials/BASH/scripts/day1/examples> egrep node *
checknodes.pbs:#PBS -l nodes=4:ppn=4
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

```
~/Tutorials/BASH/scripts/day1/examples> egrep -in nodes *
checknodes.pbs:5:#PBS -l nodes=4:ppn=4
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’ ’
```

- Print files that contain the word ”counter”

```
~/Tutorials/BASH/scripts/day1/examples> grep -l counter *
factorial2.sh
factorial.csh
factorial.sh
```

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

```
~/Tutorials/BASH/scripts/day1/examples> 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
nestedloops.csh
nestedloops.sh
quotes.csh
quotes.sh
shift10.sh
shift.csh
shift.sh
```

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

```
~/Tutorials/BASH/scripts/day1/examples> 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
shift.csh
shift.sh
```

- print the line immediately before regexp

```
apacheco@apacheco:~/Tutorials/BASH/scripts/day2/csplit> grep -B1 Normal h2o-opt-freq.log
 File lengths (MBytes):  RWF=      5 Int=      0 D2E=      0 Chk=      1 Scr=      1
 Normal termination of Gaussian 09 at Thu Nov 11 08:44:07 2010.
--
 File lengths (MBytes):  RWF=      5 Int=      0 D2E=      0 Chk=      1 Scr=      1
 Normal termination of Gaussian 09 at Thu Nov 11 08:44:17 2010.
```

- print the line immediately after regexp

```
~/Tutorials/BASH/scripts/day2/csplit> grep -A1 Normal h2o-opt-freq.log
 Normal termination of Gaussian 09 at Thu Nov 11 08:44:07 2010.
 (Enter /usr/local/packages/gaussian09/g09/l1.exe)
--
 Normal termination of Gaussian 09 at Thu Nov 11 08:44:17 2010.
```

sed

- sed (”stream editor”) is Unix utility for parsing and transforming text 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 substituion command, *s*

```
~/Tutorials/BASH/scripts/day1/examples> cat hello.sh | sed ’s/bash/tcsh/g’
#!/bin/tcsh

# My First Script

echo ‘‘Hello World!’’
```

- List of sed pattern flags and commands line options

Pattern	Operation
s	substitution
g	global replacement
p	print
I	ignore case
d	delete
G	add newline
w	write to file
x	exchange pattern with hold buffer
h	copy pattern to hold buffer

Command	Operation
-e	combine multiple commands
-f	read commands from file
-h	print help info
-n	disable print
-V	print version info
-i	in file subsitution

- Add the -e to carry out multiple matches.

```
~/Tutorials/BASH/scripts/day1/examples> 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 forms

```
~/Tutorials/BASH/scripts/day1/examples> cat hello.sh | sed ’s/bash/tcsh/g’ | sed ’s/First/First tcsh/
      g’
OR
~/Tutorials/BASH/scripts/day1/examples> sed ’s/bash/tcsh/g; s/First/First tcsh/g’ hello.sh
#!/bin/tcsh

# My First tcsh Script

echo ‘‘Hello World!’’
```

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

```
~/Tutorials/BASH/scripts/day1/examples> sed ’s:/bin/bash:/usr/bin/env tcsh:g’ hello.sh
#!/usr/bin/env tcsh

# My First Script

echo ‘‘Hello World!’’
```

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

```
~/Tutorials/BASH/scripts/day1/examples> sed ’s/\/bin\/bash/\/usr\/bin\/env tcsh/g’ hello.sh
#!/usr/bin/env 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 the sed commands

```
~/Tutorials/BASH/scripts/day1/examples> cat sedscript
s/bash/tcsh/g
~/Tutorials/BASH/scripts/day1/examples> sed -f sedscript hello.sh
#!/bin/tcsh

# My First Script

echo ‘‘Hello World!’’
```

- sed can also delete blank files from a file

```
~/Tutorials/BASH/scripts/day1/examples> sed ’/^$/d’ hello.sh
#!/bin/bash
# My First Script
echo ‘‘Hello World!’’
```

- delete line n through m in a file

```
~/Tutorials/BASH/scripts/day1/examples> sed ’2,4d’ hello.sh
#!/bin/bash
echo ‘‘Hello World!’’
```

- insert a blank line above every line which matches “regex”

```
~/Tutorials/BASH/scripts/day1/examples> sed '/First/{x;p;x;}' hello.sh
#!/bin/bash


# My First Script

echo ''Hello World!''
```

- insert a blank line below every line which matches “regex”

```
~/Tutorials/BASH/scripts/day1/examples> sed '/First/G' hello.sh
#!/bin/bash

# My First Script


echo ''Hello World!''
```

- insert a blank line above and below every line which matches “regex”

```
~/Tutorials/BASH/scripts/day1/examples> sed '/First/{x;p;x;G;}' hello.sh
#!/bin/bash


# My First Script


echo ''Hello World!''
```

- delete lines matching pattern regex

```
~/Tutorials/BASH/scripts/day1/examples> sed ’/First/d’ hello.sh
#!/bin/bash


echo ‘‘Hello World!’’
```

- print only lines which match regular expression (emulates grep)

```
~/Tutorials/BASH/scripts/day1/examples> sed -n ’/echo/p’ hello.sh
echo ‘‘Hello World!’’
```

- print only lines which do NOT match regex (emulates grep -v)

```
~/Tutorials/BASH/scripts/day1/examples> sed -n ’/echo/!p’ hello.sh
#!/bin/bash

# My First Script
```

- print current line number to standard output

```
~/Tutorials/BASH/scripts/day1/examples> sed -n ’/echo/ =’ quotes.sh
5
6
7
8
9
10
11
12
13
```

- 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

```
~/Tutorials/BASH/scripts/day1/examples> cat hello1.sh
#!/bin/bash

# My First Script

echo ‘‘Hello World!’’
~/Tutorials/BASH/scripts/day1/examples> sed -i.bak -e ’s/bash/tcsh/g’ -e ’s/First/First tcsh/g’
     hello1.sh
~/Tutorials/BASH/scripts/day1/examples> cat hello1.sh
#!/bin/tcsh

# My First tcsh Script

echo ‘‘Hello World!’’
~/Tutorials/BASH/scripts/day1/examples> cat hello1.sh.bak
#!/bin/bash

# My First Script

echo ‘‘Hello World!’’
```

- double space a file

```
~/Tutorials/BASH/scripts/day1/examples> sed G hello.sh
#!/bin/bash



# My First Script



echo ‘‘Hello World!’’
```

- double space a file which already has blank lines in it. Output file should contain no more than one blank line between lines of text.

```
~/Tutorials/BASH/scripts/day1/examples> sed ’2,4d’ hello.sh | sed ’/^$/d;G’
#!/bin/bash

echo ‘‘Hello World!’’
```

- triple space a file sed ’G;G’
- undo double-spacing (assumes even-numbered lines are always blank)

```
~/Tutorials/BASH/scripts/day1/examples> sed ’n;d’ hello.sh
#!/bin/bash
# My First Script
echo ‘‘Hello World!’’
```

- print the line immediately before or after a regexp, but not the line containing the regexp

```
apacheco@apacheco:~/Tutorials/BASH/scripts/day2/csplit> sed -n ’/Normal/{g;1!p;};h’ h2o-opt-freq.log
 File lengths (MBytes):  RWF=       5 Int=       0 D2E=       0 Chk=       1 Scr=       1
 File lengths (MBytes):  RWF=       5 Int=       0 D2E=       0 Chk=       1 Scr=       1

apacheco@apacheco:~/Tutorials/BASH/scripts/day2/csplit> sed -n ’/Normal/{n;p;}’ h2o-opt-freq.log
 (Enter /usr/local/packages/gaussian09/g09/l1.exe)
```

- print section of file between two regex:

```
~/Tutorials/BASH/scripts/day2/awk-sed> cat nh3-drc.out | sed -n '/START OF DRC CALCULATION/,/END OF
      ONE-ELECTRON 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 ......
```

- print section of file from regex to end of file

```
~/Tutorials/BASH/scripts/day2/awk-sed> 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: `http://sed.sourceforge.net/sed1line.txt`
- sed is a handy utility very useful for writing scripts for file manipulation.

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

nawk : New AWK, AT&T's version of AWK

gawk : GNU AWK, all linux distributions come with gawk. In some distros, awk is a symbolic link to gawk.

- Simplest form of using awk
  - **awk** *pattern* {`action`}
  - Most common action: `print`
  - Print file dosum.sh: `awk '{print $0}' dosum.sh`
  - Print line matching bash in all files in current directory:
    `awk '/bash/{print $0}' *.sh`
- awk patterns may be one of the following

BEGIN : special pattern which is not tested against input.
Mostly used for preprocessing, setting constants, etc. before input is read.

END : special pattern which is not tested against input.
Mostly used for postprocessing after input has been read.

/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

- Example: Print list of files that are csh script files

```
~/Tutorials/BASH/scripts/day1/examples> awk ’/^#\!\/bin\/tcsh/{print FILENAME}’ *
dooper.csh
factorial.csh
hello1.sh
name.csh
nestedloops.csh
quotes.csh
shift.csh
```

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

```
~/Tutorials/BASH/scripts/day1/examples> awk ’/^#\!\/bin\/bash/,/echo/{print $0}’ hello.sh
#!/bin/bash

# My First Script

echo ‘‘Hello World!’’
```

- 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.
- By default the field separator is space or tab. To change the field separator use the -F command.
- 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.

```
~/Tutorials/BASH/scripts/day1/examples>awk ’{print NR,’’,’’,NF,’’:’’,$0}’ hello.sh
1 , 1 : #!/bin/bash
2 , 0 :
3 , 4 : # My First Script
4 , 0 :
5 , 3 : echo ‘‘Hello World!’’
~/Tutorials/BASH/scripts/day1/examples> uptime
 11:18am  up 14 days  0:40,  5 users,  load average: 0.15, 0.11, 0.17
apacheco@apacheco:~/Tutorials/BASH/scripts/day1/examples> uptime | awk ’{print $1,$NF}’
11:19am 0.17
apacheco@apacheco:~/Tutorials/BASH/scripts/day1/examples> uptime | awk -F: ’{print $1,$NF}’
 11  0.12, 0.10, 0.16
```

- *print expression* is the most common action in the awk statement. If formatted output is required, use the *printf format, expression* action.
- 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
- 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

```
~/Tutorials/BASH/scripts/day1/examples> echo hello 0.2485 5 | awk ’{printf ‘‘%s \t %f \n %d \v %0.5d\
      n’’,$1,$2,$3}’
hello    0.248500
 5
    00005
```

- The print command puts an explicit newline character at the end while the printf command does not.

- awk has in-built support for arithmetic operations

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**
Modulo	%

Assignment Operation	Operator
Autoincrement	++
Autodecrement	–
Add result to varibale	+=
Subtract result from variable	-=
Multiple variable by result	*=
Divide variable by result	/=

```
~/Tutorials/BASH/scripts/day1/examples> echo | awk '{print 10%3}'
1
~/Tutorials/BASH/scripts/day1/examples> echo | awk '{a=10;print a/=5}'
2
```

- awk also supports trignometric functions such as sin(expr) and cos(expr) where expr is in radians and atan2(y/x) where y/x is in radians

```
~/Tutorials/BASH/scripts/day1/examples> echo | awk '{pi=atan2(1,1)*4;print pi,sin(pi),cos(pi)}'
3.14159 1.22465e-16 -1
```

- Other Arithmetic operations supported are

  exp(expr) : The exponential function
  int(expr) : Truncates to an integer
  log(expr) : The natural Logarithm function
  sqrt(expr) : The square root function
  rand() : Returns a random number N between 0 and 1 such that $0 \leq N < 1$
  srand(expr) : Uses expr as a new seed for random number generator. If expr is not provided, time of day is used.

- **awk** supports the if and while conditional and for loops
- if and while conditionals work similar to that in C-programming

```
if ( condition ) {
  command1 ;
  command2
}
```

```
while ( condition ) {
  command1 ;
  command2
}
```

- awk supports if ... else if .. else conditionals.

```
if (condition1) {
  command1 ;
  command2
} else if (condition2 ) {
  command3
}  else {
  command4
}
```

- Relational operators supported by if and while

==	: Is equal to
!=	: Is not equal to
>	: Is greater than
>=	: Is greater than or equal to
<	: Is less than
<=	: Is less than or equal to
∼	: String Matches to
!∼	: Doesn't Match

```
~/Tutorials/BASH/scripts/day1/examples> awk ’{if (NR > 0 ){print NR,’’:’’, $0}}’ hello.sh
1 : #!/bin/bash
2 :
3 : # My First Script
4 :
5 : echo ‘‘Hello World!’’
```

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

```
~/Tutorials/BASH/scripts/day1/examples> echo $(seq 1 10) | awk 'BEGIN{a=6}{for (i=1;i<=NF;i++){a+=$i
      }}END{print a}'
61
```

- 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.

```
~/Tutorials/BASH/scripts/day1/examples> awk '{print $1}' hello.sh
#!/bin/bash

#

echo

~/Tutorials/BASH/scripts/day1/examples> awk '{col=$1;print col,$2}' hello.sh
#!/bin/bash

# My

echo ``Hello
```

- Unlike Shell, awk variables are referenced as is i.e. no $ prepended to variable name.
- awk one-liners: `http://www.pement.org/awk/awk1line.txt`

- 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. Use `which awk` to find the exact location
- On my Linux desktop, the location is /usr/bin/awk.
- If unsure, just use /usr/bin/env awk

hello.awk

```
#!/usr/bin/awk -f

BEGIN {
  print "Hello World!"
}
```

```
~/Tutorials/BASH/scripts/day2/examples>./hello.awk
Hello World!
```

- 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
- arrays are subscripted with an expression between square brackets ([· · ·])

hello1.awk

```
#!/usr/bin/awk -f

BEGIN {
  x[1] = "Hello,"
  x[2] = "World!"
  x[3] = "\n"
  for (i=1;i<=3;i++)
    printf " %s", x[i]
}
```

```
~/Tutorials/BASH/scripts/day2/examples>./hello1.awk
 Hello, World!
```

- Use the delete command to delete an array element
- awk has in-built functions to aid writing of scripts

length : length() function calculates the length of a string.

toupper : toupper() converts string to uppercase (GNU awk only)

tolower : tolower() converts to lower case (GNU awk only)

split : used to split a string. Takes three arguments: the string, an array and a separator

gsub : add primitive sed like functionality. Usage gsub(/pattern/,"replacement pattern",string)

getline : force reading of new line

- Similar to bash, GNU awk also supports user defined function

```
#!/usr/bin/gawk -f
{
    if (NF != 4) {
        error(``Expected 4 fields'');
    } else {
        print;
    }
}
function error ( message ) {
    if (FILENAME != ``-'') {
        printf(``%s: ``, FILENAME) > ``/dev/tty'';
    }
    printf(``line # %d, %s, line: %s\n'', NR, message, $0) >>
          ``/dev/tty'';
}
```

getcpmdvels.sh

```
#!/bin/bash

narg=($#)
if [ $narg -ne 2 ]; then
  echo "2 arguments needed:[Number of atoms] [Velocity file]\n"
  exit 1
fi

natom=$1
vels=$2

cat TRAJECTORY | \
  awk '{ if ( NR % '$natom' == 0){ \
           printf " %s %s %s\n",$5,$6,$7 \
         }else{ \
           printf " %s %s %s",$5,$6,$7 \
         } \
       }' > $vels
```

getengcons.sh

```
#!/bin/bash

GMSOUT=$1
grep 'TIME     MODE' $GMSOUT | head -1 > energy.dat
awk '/    FS       BOHR/{getline;print }' $GMSOUT >> energy.dat
```

```
#!/bin/bash

narg=($#)
if [ $narg -ne 6 ]; then
  echo "4 arguments needed: [GAMESS output file] [Number of atoms] [Time Step (fs)] [Coordinates file] [
        Velocity file] [Fourier Transform Vel. File]"
  exit 1
fi

gmsout=$1
natoms=$2
deltat=$3
coords=$4
vels=$5
ftvels=$6
au2ang=0.5291771
sec2fs=1e15
mass=mass.dat

rm -rf $vels $coords $ftvels

######## Atomic Masses (needed for MW Velocities) ##########
cat $gmsout | sed -n ’/ATOMIC ISOTOPES/,/1 ELECTRON/p’ | \
  egrep -i = | \
  sed -e ’s/=//g’ | \
  xargs | awk ’{for (i=2;i<=NF;i+=2){printf "%s\n",$i;printf "%s\n",$i;printf "%s\n",$i}}’ > $mass
## Use the following with grep####
#grep -i -A1 ’ATOMIC ISOTOPES’ $gmsout | \
# grep -iv atomic | \
# awk ’{for (i=2;i<=NF;i+=2){printf "%s\n",$i;printf "%s\n",$i;printf "%s\n",$i}}’ > $mass
## Use the following with grep and sed ####
#grep -i -A1 ’ATOMIC ISOTOPES’ $gmsout | \
# sed -e ’/ATOMIC/d’ -e ’s/[0-9]=//g’ | \
# awk ’{for (i=1;i<=NF;i+=1){printf "%s\n",$i;printf "%s\n",$i;printf "%s\n",$i}}’ > $mass

######## Coordinates and Velocities ########################
awk ’/           CARTESIAN COORDINATES / { \
  icount=3; \
  printf "%d\n\n",’$natoms’
  while (getline>0 && icount<=7){ \
    print $0 ;\
    ++icount \
  } \
}’ $gmsout | sed ’/----/d’ > tmp.$$

#egrep -i -A5 ’cartesian coordinates’ $gmsout | \
# sed -e ’/CARTESIAN/d’ -e ’/----/d’ > tmp.$$
#
cat tmp.$$ | cut -c -42 | \
  awk ’{if ( NF == 4){ \
    printf " %4.2f %9.6f %9.6f %9.6f\n",$1,$2*’$au2ang’,$3*’$au2ang’,$4*’$au2ang’ \
  } else { \
    print $0 \
  } \
  }’ > $coords
cat tmp.$$ | cut -c 42- | sed ’/^ *$/d’ | \
  awk ’{if ( NR % ’$natoms’ ==0){ \
    printf " %15.8e %15.8e %15.8e\n",$1*’$sec2fs’,$2*’$sec2fs’,$3*’$sec2fs’ \
    } \
  else { \
    printf " %15.8e %15.8e %15.8e",$1*’$sec2fs’,$2*’$sec2fs’,$3*’$sec2fs’ \
  } \
      }’ > $vels

rm -rf tmp.$$

octave -q <<EOF
vels=load("$vels");
atmass=load("$mass");
atmass=diag(atmass);
mwvels=vels*atmass;
ftmwvels=abs(fft(mwvels));
N=rows(ftmwvels);
M=columns(ftmwvels);
deltaw=1/N/$deltat;
fid=fopen("$ftvels","w");
for I=[1:N]
  sumft=0;
  for J=[1:M]
    sumft=sumft+ftmwvels(I,J)^2;
  endfor
  fprintf(fid," %15.8e  %21.14e\n",(I-1)*deltaw,sumft);
endfor
fclose(fid);
EOF
```

getmwvels.awk

```
#!/usr/bin/awk -f
BEGIN{
     if(ARGC < 3){
        printf "3 arguments needed:[Gaussian log file] [Number of atoms] [MW Velocity file]\n";
        exit;
     }
     gaulog = ARGV[1];
     natom  = ARGV[2];
     vels   = ARGV[3];
     delete ARGV[2];
     delete ARGV[3];
}
/^ *MW Cartesian velocity:/ {
   icount=1;
   while((getline > 0)&&icount<=natom+1){
                    if(icount>=2){
                       gsub(/D/,"E") ;
                       printf "%16.8e%16.8e%16.8e",$4,$6,$8 > vels;
                    }
                    ++icount;
             }
   printf "\n" > vels;

}
```

gettrajxyz.awk

```
#!/usr/bin/awk -f
BEGIN{
     if(ARGC < 3){
        printf "3 arguments needed:[Gaussian log file] [Number of atoms] [Coordinates file]\n";
        exit;
     }
     gaulog = ARGV[1];
     natom  = ARGV[2];
     coords = ARGV[3];
     delete ARGV[2];
     delete ARGV[3];
}
/^ *Input orientation:/ {
   icount=1;
   printf "%d\n\n",natom > coords;
   while((getline > 0)&&icount<=natom+4){
                    if(icount>=5){
                       printf "%5d%16.8f%16.8f%16.8f\n",$2,$4,$5,$6 > coords;
                    }
                    ++icount;
             }
}
```

Wrap Up

References & 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`
- CSH Programming `http://www.grymoire.com/Unix/Csh.html`
- csh Programming Considered Harmful `http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/`
- Wiki Books `http://en.wikibooks.org/wiki/Subject:Computing`