Session 3

Shell Programming

Yong Zhang, Ph D

Mar 21, 2019

Outline

- Shell variables
- Shell scripts

Shell Environment

- Shell environment
  - Consists of a set of variables with values.
  - These values are important information for the shell and the programs that run from the shell.
  - You can define new variables and change the values of the variables.
- Examples:
  - PATH determines where the shell looks for the file corresponding to your command.
  - SHELL indicates what kind of shell you are using.

Shell Variables

- How do we use the values in the shell variables ?
  - Put a $ in front of their names.
    - e.g: echo $HOME
    - Prints the value that is stored in the variable HOME.
- Two kinds of shell variables:
  - Environment variables
    - Available in the current shell and the programs invoked from the shell.
  - Regular shell variables
    - Not available in programs invoked from this shell.

Shell Variables (contd.)

- Many are defined in *.cshrc* and *.login* for the *C* shell and in *.bashrc* and *.bash_profile* for *bash*.
- Example *.bashrc* file:

```
#Global variables here
export PATH TERM HOME HISTFILE
export
PATH=$PATH:/usr/afsws/bin:/usr/local/bin:/usr/local/sbin:/usr/local
/X11/bin:/usr/sbin:/usr/bin:.:~/bin:/usr/local/j2se/bin

#some nice aliases here
export PS1='[\u@\h \W]$ '
export PRINTER=ps2
```

Shell Variables (contd.)

- Comments on examples:
  - Examples are shown for *bash* only.
  - Which shell you are working in:
    - **echo $SHELL**
- Declaring regular variables in *bash*:
  - **varname=varvalue**
  - No space between *varname* and *varvalue*.
  - Sets the variable *varname* to have value *varvalue*.

Shell Variables (contd.)

- Example:

```
[axgopala@nitrogen tmp] test="this is a test"
[axgopala@nitrogen tmp] echo $test
this is a test
[axgopala@nitrogen tmp] echo test
test
[axgopala@nitrogen tmp]
```

NOTE: The $ is important to access the value in a shell variable.

- Example with space b/w *varname* and *varvalue*.

```
[axgopala@nitrogen axgopala] val = "this is a test"
bash: val: command not found
[axgopala@nitrogen axgopala] val= "this is a test"
bash: this is a test: command not found
[axgopala@nitrogen axgopala]
```

Shell Variables (contd.)

- Declaring environment variables in *bash*:
  - Using the **export** command.
  - To change a regular variable to an environment variable, we need to *export* them.
  - **varname=varvalue**
  - **export varname**
  - Sets the environment variable *varname* to have value *varvalue*.
- Example:

```
[axgopala@nitrogen tmp] test="this is a test"
[axgopala@nitrogen tmp] export test
[axgopala@nitrogen tmp] export test="this is a test"
```

Shell Variables (contd.)

- Remove declaration of regular variables and environment variables :
  - Use the unset command
    - **unset *varname***
    - Once variable is unset, the value that previously was assigned to that variable does not exist anymore.
- Example:

```
[axgopala@nitrogen axgopala] var="this is a test"
[axgopala@nitrogen axgopala] echo $var
this is a test
[axgopala@nitrogen axgopala] unset var
[axgopala@nitrogen axgopala] echo $var

[axgopala@nitrogen axgopala]
```

Regular Variables vs. Environment Variables

- We can use regular variables, just like environment variables, so why have environment variables ?
  - Regular variables are only available to the current shell.
  - Environment variables are accessible across shells and to all running programs.
- Use **printenv** to list all currently set environment variables.

Example

```
[axgopala@nitrogen axgopala] var="testing the variables"
[axgopala@nitrogen axgopala] echo $var
testing the variables
[axgopala@nitrogen axgopala] bash
[axgopala@nitrogen axgopala] echo $var

[axgopala@nitrogen axgopala]
```

```
[axgopala@nitrogen axgopala] var="testing the variables“
[axgopala@nitrogen axgopala] export var
[axgopala@nitrogen axgopala] echo $var
testing the variables
[axgopala@nitrogen axgopala] bash
[axgopala@nitrogen axgopala] echo $var
testing the variables
[axgopala@nitrogen axgopala]
```

NOTE: with the command *bash*, a new shell (*bash*) is invoked.

Common Shell Variables

- SHELL: the name of the login shell of the user.
- PATH: the list of directories searched to find executables to execute.
- MANPATH: where man looks for man pages.
- LD_LIBRARY_PATH: where libraries for executables exist.
- USER: the user name of the user who is logged into the system.
- HOME: the user's home directory.
- MAIL: the user's mail directory.
- TERM: the kind of terminal the user is using.
- DISPLAY: where X program windows are shown
- HOST: the name of the machine logged on to.
- REMOTEHOST: the name of the host logged in from.

Quotes in Shell (Single Quote)

- Quotes in Unix have a special meaning
  - Single quotes:
    - Stops shell variable expansion.

```
[axgopala@nitrogen axgopala] echo "Welcome $USER"
Welcome axgopala
[axgopala@nitrogen axgopala] echo 'Welcome $USER'
Welcome $USER
```

Quotes in Shell (Back Quote)

- Back quotes:
  - Replace the quotes with the result of the execution of the command.

```
[yzhang@ Shell]var=`ls`
[yzhang@ Shell]echo $var
S1.Regular.Expressions S2.Sed.Awk S3.Shell.Programming
[yzhang@ Shell]
```

Quotes in Shell (Double Quote)

- Double quotes:
  - No difference if they are used or not.

```
[axgopala@nitrogen axgopala]$ echo Welcome $USER
Welcome axgopala
[axgopala@nitrogen axgopala]$ echo "Welcome $USER"
Welcome axgopala
[axgopala@nitrogen axgopala]$
```

Outline

- Shell variables
- Shell scripts

Shell scripts

- A shell script is a text file with Unix commands in it.
- Shell scripts usually begin with a #! and a shell name (complete pathname of shell).
  - Pathname of shell be found using the ***which*** command.
  - The shell name is the shell that will execute this script.
    - E.g: #!/bin/bash
  - If no shell is specified in the script file, the default is chosen to be the currently executing shell.

Shell scripts (contd.)

- Any Unix command can go in a shell script
  - Commands are executed in order or in the flow determined by control statements.
- Different shells have different control structures
  - The #! line is very important.
  - We will write shell scripts with the *bash*.

Shell scripts (contd.)

- A shell script as a standalone is an executable program:
  - Must use *chmod* to change the permissions of the script to be executable also.
- Can run script explicitly also, by specifying the shell name.
  - E.g: **bash myscript**
  - Invokes the *bash* shell and then runs the script using it.
  - Its almost as if we had the line **#! /bin/bash** in the file *myscript*.

Why write shell scripts ?

- To avoid repetition:
  - If you do a sequence of steps with standard Unix commands over and over, why not do it all with just one command?
  - Or in other words, store all these commands in a file and execute them one by one.
  - To automate difficult tasks:
    - Many commands have subtle and difficult options that you don't want to figure out or remember every time.

Example

- Assume that I need to execute the following commands once in a while when I run out of disk space:

```
rm -rf $HOME/.netscape/cache
rm -f $HOME/.netscape/his*
rm -f $HOME/.netscape/cookies
rm -f $HOME/.netscape/lock
rm -f $HOME/.netscape/.nfs*
rm -f $HOME/.pine-debug*
rm -fr $HOME/nsmail
```

Example (contd.)

- We can put all those commands into a shell script, called *myscript*.

```
[axgopala@nitrogen axgopala]$ cat myscript
#! /bin/bash
rm -rf $HOME/.netscape/cache
rm -f $HOME/.netscape/his*
rm -f $HOME/.netscape/cookies
rm -f $HOME/.netscape/lock
rm -f $HOME/.netscape/.nfs*
rm -f $HOME/.pine-debug*
rm -fr $HOME/nsmail
```

Example (contd.)

- To run the script:
  - Step 1:
    - chmod u+x myscript
  - Step 2:
    - Run the script:
    - ./myscript
- Each line of the script is processed in order.

Variables in Scripts

- Shell variables:
  - Declared by:
  - **varname=varvalue**
  - To make them an environment variable, we ***export*** it.
  - **export varname=varvalue**

expr command

- The **expr** command:
  - Calculates the value of an expression.

```
[axgopala@nitrogen axgopala] value=`expr 1 + 2`
[axgopala@nitrogen axgopala] echo $value
3
[axgopala@nitrogen axgopala] count=`expr $value + 1`
[axgopala@nitrogen axgopala] echo $count
4
[axgopala@nitrogen axgopala]
```

- Why do we need the **expr** command ?

```
[axgopala@nitrogen public] file=1+2
[axgopala@nitrogen public] echo $file
1+2
[axgopala@nitrogen public]
```

Notes on *expr*

- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |
  - parentheses: (,)
  - precedence is the same as C, Java

Control statements

- Without control statements, execution within a shell scripts flows from one statement to the next in succession.
- Control statements control the flow of execution in a programming language.
- The three most common types of control statements:
  - conditionals: if/then/else, case, ...
  - loop statements: while, for, until, do, ...
  - branch statements: subroutine calls (good programming practice), goto (usage not recommended).

for loops

- *for* loops allow the repetition of a command for a specific set of values.
- Syntax:

```
for var in value1 value2 ...
do
  command_set
done
```

  - command_set is executed with each value of var (value1, value2, ...) in sequence

for loops

- Example: Listing all files in a directory.

```
#! /bin/bash

for i in *
do
  echo $i
done
```

NOTE: * is a wild card that stands for all files in the current directory, and *for* will go through each value in *, which is all the files and $i has the filename.

Conditionals

- Conditionals are used to “test” something.
  - In a bash script, the only thing you can test is whether or not a command is “successful”.
- Every well behaved command returns back a return code.
  - 0 if it was successful
  - Non-zero if it was unsuccessful (actually 1..255)

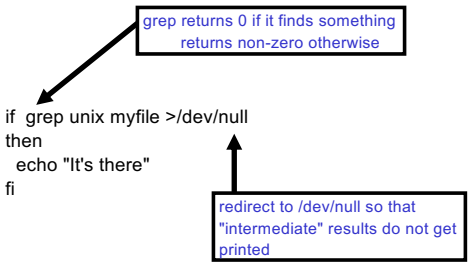
The *if* command

- Simple form:

```
if decision_command_1
then
    command_set_1
fi
```

- Importance of having then on the next line:
  - Each line of a shell script is treated as one command.
  - *then* is a command in itself
    - Even though it is part of the *if* structure, it is treated separately.

Example



Using *else* with *if*

- Example:

```
#! /bin/bash
if grep "UNIX" myfile >/dev/null
then
  echo  UNIX occurs in myfile
else
  echo  No!
  echo  UNIX does not occur in myfile
fi
```

Using *elif* with *if*

```
#! /bin/bash
if grep "UNIX" myfile >/dev/null
then
  echo  UNIX occurs in myfile
elif grep “DOS” myfile > /dev/null
then
  echo DOS appears in myfile not UNIX
else
  echo  nobody is here in myfile
fi
```

Using *colon* in shell scripts

- Sometimes, we do not want a statement to do anything.
  - In that case, use a colon ‘**:**’

```
#! /bin/bash
if grep "UNIX" myfile >/dev/null
then
  :
fi
```

  - Does not do anything when UNIX is found in *myfile* .

The *test* command

- Use for checking validity.
- Three kinds:
  - Check on files.
  - Check on strings.
  - Check on integers

Notes on *test*

- Testing on files.
  - test -f file: does file exist and is not a directory?
  - test -d file: does file exist and is a directory?
  - test -x file: does file exist and is executable?
  - test -s file: does file exist and is longer than 0 bytes?

Example

```
#!/bin/bash
count=0
for i in *; do
if test -x $i
then
  count=`expr $count + 1`
fi
done
echo Total of $count files executable
```

Notes on *test*

- Testing on strings.
  - test -z string: is string of length 0?
  - test string1 = string2: does string1 equal string2?
  - test string1 != string2: not equal?

Example

```
#! /bin/bash
if test -z $REMOTEHOST
then
:
else
  DISPLAY="$REMOTEHOST:0"
  export DISPLAY
fi
```

NOTE: This example tests to see if the value of REMOTEHOST is a string of length > 0 or not, and then sets the DISPLAY to the appropriate value.

Notes on *test*

- Testing on integers.
  - test int1 -eq int2: is int1 equal to int2 ?
  - test int1 -ne int2: is int1 not equal to int2 ?
  - test int1 -lt int2: is int1 less than to int2 ?
  - test int1 -gt int2: is int1 greater than to int2 ?
  - test int1 -le int2: is int1 less than or equal to int2 ?
  - test int1 -ge int2: is int1 greater than or equal to int2 ?

Example

```
#!/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
do
if test $i -lt $smallest
then
  smallest=$i
fi
done
echo $smallest
```

Notes on *test*

- The *test* command has an alias '[]'.
  - Each bracket must be surrounded by spaces

```
#!/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
do
if [ $i -lt $smallest ]
then
  smallest=$i
fi
done
echo $smallest
```

The *while* loop

- While loops repeat statements as long as the next Unix command is successful.

```
#! /bin/bash
i=1
sum=0
while [ $i -le 100 ]
do
  sum=`expr $sum + $i`
  i=`expr $i + 1`
done
echo The sum is $sum.
```

The *until* loop

- Until loops repeat statements until the next Unix command is successful.

```
#! /bin/bash
x=1
until [ $x -gt 3 ]
do
  echo x = $x
  x=`expr $x + 1`
done
```

Shell Script Debugging

–sh command to debug a shell script

- –-n flag causes the shell to parse the syntax structures without execution
- –-v flag displays shell input lines as they are read
- –-x display commands and their arguments as they are executed.

Examples

```
[yzhang@ data]sh -n test.sh
[yzhang@ data]sh -v test.sh
#!/bin/bash
for f in *
do
   file $f
done
CTCF.bed: ASCII text
…
test.sh: Bourne-Again shell script text executable
yeast_gene.fa: ASCII text
[yzhang@ data]sh -x test.sh
+ for f in '*'
+ file CTCF.bed
CTCF.bed: ASCII text
…
+ for f in '*'
+ file test.sh
test.sh: Bourne-Again shell script text executable
+ for f in '*'
+ file yeast_gene.fa
yeast_gene.fa: ASCII text
```

Use of set builtin command

```
#!/bin/bash
clear
# turn on debug mode
set -x
for f in *
do
   file $f
done
# turn OFF debug mode
set +x
ls
# more commands
```

$0, $1, etc ...

```
[yzhang@ code]more ex1.sh
#!/bin/bash
echo "The script name is       ==> $0"
echo "Total parameter number is ==> $#"
echo "Your whole parameter is   ==> '$@'"
echo "The 1st parameter        ==> $1"
echo "The 2nd parameter        ==> $2"
[yzhang@ code]
[yzhang@ code]./ex1.sh par1 par2 par3 par4
The script name is       ==> ./ex1.sh
Total parameter number is ==> 4
Your whole parameter is   ==> 'par1 par2 par3 par4'
The 1st parameter        ==> par1
The 2nd parameter        ==> par2
```

shift command

```
[yzhang@ code]more ex2.sh
#!/bin/bash
echo "Total parameter number is ==> $#"
echo "Your whole parameter is   ==> '$@'"
shift
echo "Total parameter number is ==> $#"
echo "Your whole parameter is   ==> '$@'"
shift 3
echo "Total parameter number is ==> $#"
echo "Your whole parameter is   ==> '$@'"
[yzhang@ code]
[yzhang@ code]./ex2.sh par1 par2 par3 par4 par5 par6
Total parameter number is ==> 6
Your whole parameter is   ==> 'par1 par2 par3 par4 par5 par6'
Total parameter number is ==> 5
Your whole parameter is   ==> 'par2 par3 par4 par5 par6'
Total parameter number is ==> 2
Your whole parameter is   ==> 'par5 par6'
```

More shell script examples

– Print out the current user name and directory

```
#!/bin/bash
echo "Your name is ==> $(whoami)"
echo "The current directory is ==> $(pwd)"
```

```
# another solution
#!/bin/bash
echo "Your name is ==> `whoami`"
echo "The current directory is ==> `pwd`"
```

$(( )): integer calculation

```
[yzhang@ code]echo $((4 + 3))
7
[yzhang@ code]echo $((4 - 3))
1
[yzhang@ code]echo $((4 * 3))
12
[yzhang@ code]echo $((4 / 3))
1
[yzhang@ code]echo $((4 % 3))
1
[yzhang@ code]echo $((4 * 3.1))
-bash: 4 * 3.1: syntax error: invalid arithmetic operator (error token is
".1")
```

1 + 2 + 3 + … + n

```
#!/bin/bash
read -p "Please input an integer number: " number
i=0
s=0
while [ "$i" != "$number" ]
do
   i=$(($i+1))
   s=$(($s+$i))
done
echo "the result of '1+2+3+...$number' is ==> $s"
```

1 + 2 + 3 + … + n (cont'd)

```
#!/bin/bash
echo "input a number"
read n
i=0
s=0
for ((i=0;i<=n;i++))
do
   s=$(($s+$i))
done
echo "1+2+3+...+$n=$s"
```

More shell script examples

```
#!/bin/bash
if [ ! -e logical ]; then
   touch logical
   echo "Just make a file logical"
   exit
elif [ -f logical ]; then
   rm logical
   mkdir logical
   echo "remove file ==> logical"
   echo "and make directory logical"
   exit
elif [ -d logical ]; then
   rm -rf logical
   echo "remove directory ==> logical"
   exit
else
   :
fi
```

EMBOSS

- The European Molecular Biology Open Software Suite
- List of programs at
  http://emboss.sourceforge.net/apps/
  ex: Smith-Waterman local alignment (water)
- Programs have two formats: interactive and one-line
- Conducive to embedding in scripts for batch analysis
- Traditionally command-line but web interfaces are becoming available

Install the latest version of EMBOSS

```
wget ftp://ftp.ebi.ac.uk/pub/software/unix/EMBOSS/emboss-latest.tar.gz
tar xfvz emboss-latest.tar.gz
cd EMBOSS-6.6.0
sudo ./configure
sudo make
sudo make install

[yzhang@ code]wossname
Finds programs by keywords in their short description
Text to search for, or blank to list all programs: motif
SEARCH FOR 'MOTIF'
antigenic      Finds antigenic sites in proteins
dreg           Regular expression search of nucleotide sequence(s)
epestfind      Finds PEST motifs as potential proteolytic cleavage sites
fuzznuc        Search for patterns in nucleotide sequences
fuzzpro        Search for patterns in protein sequences
```

EMBOSS examples

- needle: Needleman-Wunsch global alignment
  needle seq1.fa seq2.fa -auto -outfile seq1.seq2.needle
- water: Smith-Waterman local alignment of sequences
  water seq1.fa seq2.fa -auto -outfile seq1.seq2.water
- dreg: regular expression search of a nucleotide sequence
  dreg -sequence seq2.fa -pattern GAT[TC]T -outfile mySeq_dreg.txt

Shell script example

```
#!/bin/bash
# alignSeqs.sh: align a pair of sequences
# Check to make sure you get two arguments (sequence files)
if [ $# != 2 ]
then
   echo "Usage: $0 seq1 seq2"; exit
fi
# Local alignment
localOut=$1.$2.water.out
water $1 $2 -auto -outfile $localOut
echo "Wrote local alignment to $localOut"
# Global alignment
globalOut=$1.$2.needle.out
needle $1 $2 -auto -outfile $globalOut
echo "Wrote global alignment to $globalOut"
```