Lecture 21

Unix Commands and Shell Scripting Part II

Shibo Li

shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

Final Exam: Dec 9th 2024

- 5:30 pm - 7:30 pm @ MCH 201
- multiple-choice questions (30 pts)
- short answer questions (30 pts)
- programming questions (50 pts)
- Paper exam, closed-book, no cheat sheet, no electronic devices (phone, tablet, laptop, calculator *etc.*)
- Covers all the topics of this course

Remaining schedule of this class

- Today - Final formal lecture
- Nov 25th (Mon) - Review Session
- Nov 27th (Wed) - No class (holidays)
- Dec 2nd (Mon) - No calss (cancelled)
- Dec 4th - Clarification-Oriented (Q&A)

Recap

- `grep` is a very useful utility that searches files for a particular pattern.
- The pattern can be a word, a string enclosed in single quotes,or a regular expression.
- Syntax: `grep options pattern files`
- grep has many options; a few are noted below
  - -i Ignore case
  - -n Display line numbers
  - -l Display only the names of the files and not the actual lines
  - -P pattern is a Perl regular expression, not a Unix regular expression

Recap

- A Shell Script is an executable file containing
  - Unix shell commands
  - Programming control constructs (if, then, while, until, case,for, break, continue, while, *etc.*)
  - Basic programming capabilities (assignments, variables, arguments, expressions, *etc.*)
- The file contents comprise the script

Recap

- Unlike a C++ program, that is compiled and then executed, shell scripts are **interpreted**.
- Usually, the first line of the script indicates which shell is used to interpret the script.

Recap

- The echo command can be used in a shell script to print text, to the terminal display
- Syntax: echo <zero or more values>
- Examples:

```
echo "Hello World"
echo "hello" "world" #two values
echo hello #need not always use quotes
echo "please enter your name"
```

- These are variables provided as part of the shell's operational
- They exist at startup but can be changed
- Examples are: USER, HOME, PATH, SHELL, HOSTNAME
- The "setenv" command (in tcsh) is used to set these, for example, by:
  - setenv PATH $PATH:/home/here/bin(this sets the PATH variable so that it's current value isappended by :/home/here/bin)
  - Note that setenv is how tcsh sets the environment variables

Recap

- You can also specify variables yourself and these can also be used inside a script
- In tcsh, the "set" command is used to set a variable to a string value
- Form: `set<name>=<value>`
- Examples:

```
set firstVar = "any string"
set secondVar = 3
set mypath = /home/special/public_html
```

Recap

- Arguments on the command line can be passed to a shell script, just as you can pass command line arguments to a program
- $1, $2, ..., $9 are used to refer to up to nine command line arguments (similar to C's argv[1], argv[2], ..., argv[9]).
- Note that $0 contains the name of the script (argv[0])
- Example

```
shprog.sh john 40
shprog.sh bob 45 "new york"
```

Testing Conditions

- There are two ways to test for conditions. The two general forms are:

```
test <condition>
// or
[ <condition> ]
```

- The second is easier to read and is more common
- Remember to include a space before and after the bracket
- A condition can be reversed with a `!` before the condition (this is the same as not condition)

```
[ !<condition> ]
```

- A `:` command in place of condition always returns true

Testing Conditions

- To test if a file is readable

```
[ -r prog.txt ]
[ -r $1.c ]
```

- To test if a file is writeable

```
[ -w specialfile.txt ]
```

- To test if a file is executable

```
[ -x prog4.sh ]
```

Testing Conditions

- To test if a file exists

```
[ -f temp.text ]
```

- Testing for the negation - use ! (eg. not writeable)

```
[ ! -w nochange.txt ]
```

Numeric Tests

- The following operators can be used for numeric tests: `-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`
- Examples:

```
[ $1 -lt $2 ]
[ $1 -gt 0 ]
[ $# -eq 2 ]
[ $# -lt 3 ]
```

The Simple if Statement

- General Form:

```
if [ <condition> ]
then
   one-or more commands
fi
```

- Example:

```
if [ -r tmp.text ]
then
   echo "temp.text is a readable file"
fi
```

General if statement

- General Form:

```
if <condition>
then
   one-or-more-commands
elif <condition>
then
   one-or-more-commands
...
else
   one-or-more-commands
fi
```

- Note that you can have 0 or more elif statements and that the else is optional.

Testing Strings

- Performing string comparisons. It is a good idea to put the shell variable being tested inside double quotes.

```
[ "$1" = "yes" ]
[ "$2" != "no" ]
```

- Note that the following will give a syntax error when $1 is empty since:

```
[ $1 != "no" ]
# becomes
[ != "no" ]
```

Quoting Rules

- Using single quotes

```
'xyz' # disables all special characters in xyz
```

- Using double quotes

```
"xyz" # disables all special characters in xyz except
    $, ', and \
```

- using the backslash

```
\x # disables the special meaning of character x
```

Quoting Examples

```
var1="alpha" #set the variable
echo $var1  #prints: alpha
echo "$var1" #prints: alpha
echo '$var1' #prints: $var1
```

Quoting Examples

```
cost=2000
echo 'cost:$cost' #prints: cost:$cost
echo "cost:$cost" #prints: cost:2000
echo "cost:\$cost" #prints: cost:$cost
echo "cost:\$$cost" #prints: cost:$2000
```

Testing with Multiple Conditions

- && is the and operator
- || is the or operator
- Checking for the and of several conditions

```
[ "$1" = "yes" ] && [ -r $2.txt ]
[ "$1" = "no" ] && [ $# -eq 1 ]
```

- Checking for the or of several conditions

```
[ "$1" = "no" ] || [ "$2" = "maybe" ]
```

String Relational Operators

- The set of string relational operators are: `=`, `!=`, `>`, `>=`, `<`, `<=`
- The `>`, `>=`, `<`, `<=` operators assume an ASCII ordering (for example "a" $<$ "c")
- These operators are used with the `expr` command that computes an expression. The backslash has to be used before the operators so that they are not confused with I/O redirection

Some if statement examples

```
if [ "$1" != "" ] || [ ! -r $1 ]
then
   echo "the file" $1 "is not readable"
fi

if [ $var1 -lt $var2 ]
then
   echo $var1 "is less than" $var2
elif [ $var1 -gt $var2 ]
then
   echo $var1 "is greater than" $var2
else
   echo $var1 "is equal to" $var2
fi
```

The case Statement

- Compares stringvalue to each of the strings in the patterns.
- At a match, it does the corresponding commands.
- ;; indicates to jump to the statement after the esac (end of case).
- *) means the default case.
- Form

```
case stringvalue in
pattern1)
   one or more commands;;
pattern2)
   one or more commands;;
...
*) one or more commands;;
esac
```

case Statement Example

```
echo "Would you like to remove the file $1?" echo "Please
    enter yes or no - "
read ans
case $ans in
"yes") rm $1
   echo "file removed"
   ;;
"no")
   echo "file not removed"
   ;;
*) echo "Response unclear"
esac
```

while and until statements

- The while and until statements are analogous to the c++ while loop
- while general form

```
while <condition>
do
   one or more commands
done
```

- until General form

```
until <condition>
do
   one or more commands
done
```

while and until examples

```
read cmd
while [ $cmd != "quit" ]
do
   ...
   read cmd
done

read cmd
until [ $cmd = "quit" ] do
   ...
   read cmd
done
```

for statement

- The for statement functions similarly to the for loop in several languages.
- General from:

```
for variable in set
do
   one or more commands
done;
```

- Example

```
for filename in *
do
   echo $filename
done;
```

More on the `for` statement

```
for <variable> [ in <word_list> ]
do
   one or more commands
done
```

- The `<variable>` is assigned each word in the list, where the set of commands is performed each time the word is assigned to the variable.
- If the "`in <word list>`" is omitted, then the variable is assigned each of the command line arguments

The exit command

- The exit command causes the current shell script to terminate. There is an implicit exit at the end of each shell script.
- The exit command can set the status at the time of exit. If the status is not provided, the script will exit with the status of the last command.
- General form

```
exit
#or
exit <status>
```

The `exit` command

- `$?` is set to the value of the last executed command
- Zero normally indicates success. Nonzero values indicate some type of failure. Thus, exit 0 is normally used to indicate that the script terminated without errors.
- It is thus good practice to ensure that if the shell script terminates properly, it is with an "exit 0" command.
- If the shell script terminates with some error that would be useful to a calling program, terminate with an "exit 1" or other nonzero condition.
- Most Unix utilities that are written in C will also call "`exit(<value>);`" upon termination to pass a value back to the shell or utility that called that utility.

The exit command

- The following shell script exits properly. It also distinguishes the response through the value returned.

```
#!/bin/sh
#determines a yes (0) or no (1) answer from user echo
    "Please answer yes or no";
read answer
while :
do
   case $answer in
      "yes") exit 0;;
      "no") exit 1;;
      *) echo "Invalid; enter yes or no only"
         read answer;;
   esac
done
```

Testing the Exit Status

- Conditions tested in control statements can also be the exit status of commands.
- Assume "script1.sh" is called in another script - "script2.sh".
- The following segment will test this as part of `script2.sh`

```
if script1.sh
then
   echo "enter file name"
   read file
else
   echo "goodbye"; exit 0
fi
```

expr Examples

```
var=‘expr $var + 1‘ #increment var by 1

#check if the value of s1 is less than value of s2
if [ ‘expr $s1 \< $s2‘= 1 ]

#multiply value of beta by 2
beta=‘expr $beta \* 2‘

set beta = 10;
expr $beta / 2 #using tcsh directly, result is 5

#output 1 if variable alpha is hello
expr "$alpha" = hello
```

Command Substitution

- Allows the output of a command to be captured and used as part of another command or stored in a variable
- Placing a string in back quotes ‘... ‘does command substitution
- The standard output of the command replaces the back quoted string

Command Substitution

- Examples
- the value of count is assigned the number of words in file $1

```
count=‘wc -w <$1‘
```

- checks if the number of lines in the file is <1000

```
if [ ‘wc -l < $2.txt‘ -lt 1000 ]
```

- print out all *.sh files containing the word exit

```
   cat ‘grep -l exit *.sh‘
```

Regular Expression

A **regular expression (regex)** is a sequence of characters that defines a search pattern. It is a powerful tool used in programming, scripting, and text processing to find, match, extract, or manipulate text based on specific patterns.

Common Uses of Regex

- **Search and Replace**: Find specific text patterns and replace them.
- **Validation**: Verify whether a string matches a specific format (e.g., email, phone numbers).
- **Text Extraction**: Extract parts of a string that match a pattern.
- **Splitting Strings**: Split strings into parts based on a delimiter or pattern.

Regular Expressions and Wildcards

- Many Unix utilities use regular expressions
- A regular expression is a compact representation of a set of strings
- Note that the shell uses wildcards (*, ?, etc.) for filename matching. The special characters are not necessarily used the same way in regular expressions
- Thus the pattern "alpha*.c" for filenames is not the same when used in the grep command (for example) to match a regular expression!
- In a regular expression, "*" means match zero or more of the preceding character

Regular expression operators

- **Concatenation**
  - This is implicit and is simply one character followed by another.
  - `ab` - matches the character "a" followed by "b"
  - `alpha` - several characters concatenated
- `*` **operator**
  - Indicates zero or more instances of the preceding character or preceding regular expression if grouping - parentheses ( ) - are used.
  - `ab*c` - matches ac, abc, abbc, etc.

Regular expression operators

- **+ operator**
  - Similar to * except matches **1** or more instances of the preceding character
- .(dot) **operator**
  - matches any single character except newline
  - `a.b` - matches a followed by any character, then b. For example adb, a&b, abb, etc.
- **– operator**
  - is used to define a range.

Regular expression operators

- `[]` **operator**: a set
  - `[adkr]` - match a, d, k, r
  - `[0-9]` - match 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - `[a-z]` - match lower case letters
- ∧ after "[" means match anything **not** in the set.
  - `[∧aeiou]` - match any character except a vowel
  - `[∧0-9]` - match any character except a decimal digit

Anchors

- Anchors ∧ and $ can be used to indicate that a pattern will only match when it is at the beginning or end of a line (note that the following use of "∧" is different from its use inside a set of characters)
- ∧`alpha` - match the string "alpha" only when it is at the beginning of the line
- `[A-Za-z]+$` - a name at the end of the line
- ∧`alpha*zeta$` - start with alph, end with zeta and any number of "a"s in between

Alternation and Grouping

- Use the "|" character to choose between alternatives. Parentheses are for grouping
  - a|b - match a or b
  - a*|b - match any number of a's or b.
  - (ab*a)* - any number of ab*a

grep and egrep

- egrep is extended grep and extends the syntax of regular Generally grep does not support the parentheses, the + operator, the | operator or the ? operator (zero or one occurrence).
- The flag –E in grep generally gives egrep behavior.

grep options

- `-i` will make the search case insensitive
- `-c` will count the number of lines matched to be printed
- `-w` will force the search to look for entire words (not part of a longer word)
- `-v` will count the lines that do not match to be output
- `-l` will return only the name of the file when grep finds a match

grep examples

- look for the substring “alpha” in file “filename”

```
grep alpha filename
```

- look for the substring of one or more i’s

```
grep "ii*" filename
```

- look for a line that starts with “begin”
  `grep ^begin filename`

grep examples

- find a “recieve” in any file ending in .sh

```
grep recieve *.sh
```

- find a substring with an a, b, or c, followed by any number of other characters

```
grep "[abc].*" filename
```