CSE 374 Lecture 5

Regular expressions and grep

Okay, lets make a script!

1. First line of file is #!/bin/bash (specifies which interpreter to execute)
2. Make file executable (chmod u+x)
3. Run a file ./myNewScript
4. Shell sees the shell program (/bin/bash) and launches it to run the script
5. Can include
    a. String tests (string returns true if non-zero length, string < string, etc.)
    b. Logic (&&,||,!) - use double brackets
    c. File tests (-d : is directory, -f: is file, -w: file has write permission etc.)
    d. Math - use double parens

Arithmetic

Variables hold strings, so we need a way to tell the shell to evaluate them numerically:

K=$i+$j does not add the numbers

Use the shell function ((

k=$(($i+$j))

Or let k=”$i+$j”

The shell will automatically convert the strings to the numbers

Functions and local variables

Yes, possible

Generally, a script's variables are global

```
name () compound-command [ redirections ]
or
function name [()] compound-command [
redirections ]

Ex:

func1()
{
    local var='func1 local'
    func2
}
```

Conditionals

Binary operators: -eq -ne -lt -le -gt -ge

Can use the [[shell command to use < , > , ==

Syntax is a little different, but commands works as expected

if *test*; then
    commands
fi

while *test*; do
    commands
done

for *variable* in *words*; do
    commands
done

Flow control

test *expression* or [*expression*]

```
if [ -f .bash_profile ]; then
    echo "You have a .bash_profile.
Things are fine."
else
    echo "Yikes! You have no
.bash_profile!"
fi
```

http://linuxcommand.org/lc3_man_pages/testh.html

Stuff to watch out for

White space: spacing of words and symbols matters

Assign WITHOUT spaces around the equal, brackets are WITH SPACES

Typo on left creates new variable, typo on right returns empty string.

Reusing variable name replaces the old value

Must put quotes around values with spaces in them

Non number converted to number produces ‘0’

Shell-scripting Notes

Bash Scripting	Java Programming
Interpreted	Compiled
Esoteric variable access	Highly structured, Strongly typed
Everything is a string	Strings have library processing
Easy access to files and program	Data structures and libraries
Good for quick & interactive programs	Good for large complex programs

Scripting Style Guide

Do one thing and do it well.

Scripts should generally be <200 lines

Always use spaces, not tabs (indent line with two spaces)

Comment code with ‘#’

https://google.github.io/styleguide/shell.xml

Practice ...

Globbing and Regex

Globbing: the shell filename expansion; matches some patterns

Regular expressions (regex): a set of rules for matching patterns in text

(We see regular expressions in math, as formal grammars in cs, and other variations as well. Different applications (egrep) my have slightly different rules.)

Regex theory

1. A set of rules for matching a pattern (P) to a string (S)
2. All strings are made of a combination of the null (empty) set, the empty string ε, and a single character.
3. Regular expressions match a string if
   3.1. P is a literal character (a, b, …) that matches the string S
   3.2. P_1P_2 matches S if $S = S_1S_2$ such that P_1 matches S_1 and P_2 matches S_2
   3.3. $P_1|P_2$ matches S if P_1 matches S OR P_2 matches S
   3.4. P* matches S if there is an *i* such that P...P (*i* times) matches S. Includes *i=0* which matches ε.

Regex rules

Regular expressions have

Characters: the literal characters [a b 9] (S is an exact duplicate of P)

Anchors: sets the position in the line where P may be found (^ or $)

Modifiers: modify the range of text P may match (* or [set_of_chars])

Note: Regex details & implementation may vary between application, but general rules apply.

Regex special characters *(some are \escaped)*

\ : escape following character

‘.’ : matches any single character at least once

$p_1|p_2$: matches p_1 OR p_2

‘*’ : matches zero or more of the previous p

‘?’ : matches zero or one of the previous p ($\varepsilon|p$)

‘+’ : matches one or more of previous p (pp*)

() : group patterns for order of operations

{} :

[] : contain literals to be matched (single or range)

^ : Anchors to beginning of line

$: anchors to end of line

<> : word boundaries

Classes of characters

```
.      == any character
[a-z]  == a, b, c, d … z
[A-Z]  == A, B, C, D … Z
[0-9]  == 0,1,2,3,4,5,6,7,8,9
abc    == literally abc
```

```
c.t          → cat, cut, cota
[Hh]ello!    → Hello!, hello!
[BLERG]      -> B, L, E, R or G
[0-5][5-9]   → 15,16,17,18,19
               25,26,27,28,29
               35,36,37,38,39
               45,46,47,48,49
               55,56,57,58,59
```

Repipipitititition

* == *zero or more*, a* → {, a, aa, aaa, aaaa, …}

+ == *one or more*, a+ → {a, aa, aaa, …}

? == *zero or one* of the preceding, a? → {, a}

{n} == exactly n repetitions of the preceding, a{3} → aaa

a|b == a or b, this|that|when|how → this, that, when, how

All but | are **POSTFIX OPERATORS** (they come after the pattern)

Invisible characters

^ == the start of a line

$ == the end of a line

\t == a tab

? == *zero or one* of the preceding

Extras

[^abc] : matches everything NOT abc

‘*’ : is greedy; matches as much as possible

grep

(run man grep now!)

Grep - a program to do matching using regular expressions

Grep -e / egrep - uses extended regex

Grep Regex

By default, grep matches each line against .*p.*

You can anchor the pattern with ^ (beginning) and/or $ (end) or both (match whole line exactly)

These are still “real” regular expressions

Backreference & repeated matches

Up to 9 times in a pattern, you can group with (p) and refer to the matched text later!

You can refer to the text (most recently) matched by the nth group with \n.

Simple example: double-words ^\([a-zA-Z]*\)\1$

You cannot do this with actual regular expressions; the program must keep the previous strings.

\(p\)\{n\} will match the p n times. \{n,m\} matches at least n, but not more than m times.

Bash Regex Gotchya's

- Modern (i.e., gnu) versions of grep and egrep use the same regular expression engine for matching, but the input syntax is different for historical reasons
    - For instance, \{ for grep vs { for egrep – See grep manual sec. 3.6
- Must quote patterns so the shell does not muck with them – and use single quotes if they contain $ (why?)
- Must escape special characters with \ if you need them literally: \. and . are very different
    - But inside [ ] many more characters are treated literally, needing less quoting (\ becomes a literal!)