CST8177 – Linux II

More Scripting and Regular Expressions

Todd Kelley
kelleyt@algonquincollege.com

Today's Topics

- Regular Expression Summary
- Regular Expression Examples
- Shell Scripting

Globbing versus Regex

- Do not confuse filename globbing and regular expressions: they use similar characters to mean different things
- filename globbing is for the shell in a command line, to match against existing pathnames in the current directory
- regex are used by vi, sed, awk, grep, and others, for matching patterns in any source of text, such as the contents of a file, or standard input

Regex versus Globbing

- ls a*.txt # this is filename globbing
  - matches existing filenames in current directory beginning with 'a', ending in '.txt'
- grep 'aa*' foo.txt # regular expression
  - matches strings in foo.txt beginning with 'a' followed by zero or more 'a's
  - the single quotes protect the '*' from filename globbing

- Be careful with quoting:
  - grep aa* foo.txt # no single quotes, bad idea
    - shell will try to do filename globbing on aa*, changing it into existing filenames that begin with aa before grep runs: we don't want that.

Globbing versus Regex

- BASH globbing: looks through existing filenames for matches
- grep regexp: looks through lines text for matches

- BASH: *.txt
- regexp: ^.*\.txt$ OR ^.*[.]txt$

- BASH: [abc].txt (don't use ranges unless POSIX)
- regexp: ^[abc]\.txt$ (don't use ranges unless POSIX)

- BASH: ????.txt
- regexp: ^....\.txt$

Match First and Longest

- Regular expressions match the first and longest possible match
- First and longest, in that order
- Remember the order: First comes first, then longest
- if the string is `aaabbaaaaaa`, `and the regular expression is` `aa*` then
  - the leading `aaa` is first and longest match
  - the leading `a` is a first possible match, but not longest
  - the trailing `aaaaaa` is longest possible, but not first

Metacharacter meanings

- . matches any single character
- [xyz] matches any single character inside []
- [^xyz] matches any single character not inside []
- [a-z] ranges are dangerous: use POSIX character classes instead (see below)
- ^ matches empty string at beginning of line
- $ matches empty string at the end of line

POSIX character classes

- [:alnum:] a – z, A – Z, and 0 – 9
- [:alpha:] a – z and A – Z
- [:cntrl:] control characters
- [:digit:] 0 – 9
- [:lower:] a – z
- [:print:] visible characters, plus [:space:]
- [:punct:] Punctuation characters and other symbols
  - !"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~
- [:space:] White space (space, tab)
- [:upper:] A – Z
- [:xdigit:] Hex digits: 0 – 9, a – f, and A – F
- [:graph:] (0x21 – 0x7E) (we won't use)

POSIX character classes (cont'd)

- POSIX character classes go inside [...]
- examples
  - [[:alnum:]] matches any alphanumeric character
  - [[:alnum:]}] matches one alphanumeric or }
  - [[:alpha:][:cntrl:]] matches one alphabetic or control character
- Take NOTE!
  - [:alnum:] matches one of :,a,l,n,u,m
  - [abc[:digit:]] matches one of a,b,c, or a digit

Repeat preceding (Repetition)

- * means match the preceding regular expression zero or more times
- The following might not be supported in non-extended regex, depending on the implementation
- For extended regex, leave out the \
- \? means match zero or one times
- \+ means match one or more times
- \{n\} means match exactly n times
- \{n,\} means n or more times
- \{n,m\} means at least n times, but not more than m times

Alternation

- this is an extended regular expression feature: note the backslash
- can do this kind of thing in grep with -e
  - example: `grep -e 'abc' -e 'def' foo.txt`
  - matches `abc` or `def`
- this is advanced for us, for now
- \| is an infix "or" operator
- a\|b means a or b but not both
- aa*\|bb* means one or more a's, or one or more b's
- for extended regex, leave out the \

Precedence

- repetition is tightest
  - xx* means x followed by x repeated, not xx repeated
- concatenation is next tightest
  - aa*\|bb* means aa* or bb*
- The following are for extended regular expressions (we won't worry about them for now)
- alternation is the loosest or lowest precedence
- Precedence can be overridden with grouping (next slide)

Grouping

- Extended regular expression feature (advanced for us, for now)
- \(and \) can be used to group regular expressions, and override the precedence rules
- abb* means ab followed by zero or more b's
- a\(bb\)*c would mean a followed by zero or more pairs of b's followed by c
- abbb\|cd would mean abbb or cd
- a\(bbb\|c\)d would mean a, followed by bbb or c, followed by d

Remove meaning of metacharacter

- To remove the special meaning of a metacharacter, put a backslash in front of it
- * matches a literal *
- \. matches a literal .
- \\ matches a literal \
- \$ matches a literal $
- \^ matches a literal ^

Tags or Backreferences

- Another extended regular expression feature (advanced for us, for now)
- When you use grouping, you can refer to the n'th group with \n
- \(..*\)\1 means any sequence of one or more characters twice in a row
- The \1 in this example means whatever the thing between the first set of \(\) matched

Testing Regular Expressions

- grep --color=auto 'expr'
- the above will show you the parts of the string that match expr (all matches, not just the first)

Shell scripting

- So far, we have the International Script Header:
  - The interpreter magic, or "shebang": #!/bin/sh -u
  - Set the PATH
  - Set the umask
  - Set the locale
- We then follow the header with commands like the ones we type at the shell prompt.
- The stdin, stdout, stderr of the shell script are the same those of the commands inside.

New Scripting techniques

- Today we'll add three scripting techniques
  - positional parameters and passing arguments to shell scripts
  - interacting with the user
  - if statements

Positional Parameters

- $# holds the number of arguments on the command line, not counting the command itself
- $0 is the name of the script itself
- $1 through $9 are the first nine arguments passed to the script on the command line
- After $9, there's ${10}, ${11}, and so on
- $* and $@ denote all of the arguments
- "$*" is one word with spaces in it
- "$@" produces a list where each argument is a separate word

Sample script

```
#!/bin/sh -u
PATH=/bin:/usr/bin ; export PATH
umask 022
unset LC_ALL                          # unset the over-ride
variable
LC_COLLATE=en_US.utf8 ; export LC_COLLATE # sort by character
set
LC_CTYPE=en_US.utf8 ; export LC_CTYPE # handle multi-byte chars
LANG=en_US.utf8

echo "The number of arguments is: $#"
echo "The command name is $0"
echo "The arguments are $*"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "The third argument is: $3"
```

Interacting with the user

- to get input from the user, we can use the `read` builtin
- `read` returns an exit status of 0 if it successfully reads input, or non-zero if it reaches EOF
- `read` with one variable argument reads a line from stdin into the variable
- Example:

```
#!/bin/sh
read aline #script will stop, wait for user
echo "you entered: $aline"
```

Interacting with the user (cont'd)

- Use the -p option to read to supply the user with a prompt
- Example

```
#!/bin/sh -u
read -p "enter your string:" aline
echo "You entered: $aline"
```

Interacting with the user (cont'd)

- `read var1` puts the line the user types into the variable `var1`
- `read var1 var2 var3` puts the first word of what the user types in to `var1`, the second word into `var2`, and the remaining words into `var3`

```
#!/bin/sh -u
read var1 var2 var3
echo "First word: $var1"
echo "Second word: $var2"
echo "Remaining words: $var3"
```

Exit Status

- Each command finishes with an exit status
- The exit status is left in the variable ? ($?)
- A non-zero exit status normally means something went wrong (grep is an exception)
- non-zero means "false"
- A exit status of 0 normally means everything was OK
- 0 means "true"
- grep returns 0 if a match occurred, 1 if not, and 2 if there was an error

If statement

```
if list1; then
    list2;
fi
```

- `list1` is executed, and if its exit status is 0, then `list2` is executed
- a list is a sequence of one or more pipelines, but for now, lets say it's a command

Test program

- A common command to use in the test list of an if statement is the `test` command
- `man test`
- Examples:
- `test -e /etc/passwd`
- `test "this" = "this"`
- `test 0 -eq 0`
- `test 0 -ne 1`
- `test 0 -le 1`

If statement with test

```
if test "$1" = "hello"; then
   echo "First arg is hello"
fi

if test "$2" = "hello"; then
   echo "Second arg is hello"
else
   echo "Second arg is not hello"
fi
```

The program named [

```
Todd-Kelleys-MacBook-Pro:CST8177-13W tgk$ ls -li /bin/test /bin/[
1733533 -r-xr-xr-x  2 root  wheel  43120 27 Jul  2011 /bin/[
1733533 -r-xr-xr-x  2 root  wheel  43120 27 Jul  2011 /bin/test
Todd-Kelleys-MacBook-Pro:CST8177-13W tgk$
```

- notice that `[` is another name for the `test` program:

```
if [ -e /etc/passwd ]; then
    echo "/etc/passwd exists"
fi
```

is the same as

```
if test -e /etc/passwd; then
    echo "/etc/passwd exists"
fi
```

Practicing with [

```
$ [ 0 -eq 0 ]
$ echo $?
0
$ [ "this" = "that" ]
$ echo $?
1
$ [ "this" = "this" ]
echo $?
0
$ ["this" = "this"]                    # forgot the space after [
-bash: [this: command not found
$ [ "this" = "this"]                   # forgot the space before ]
-bash: [: missing ']'
```