CST8177 – Linux II

Regular Expressions

Todd Kelley
kelleyt@algonquincollege.com

Topics

- POSIX character classes
- Some Regular Expression gotchas
- Regular Expression Resources
- Assignment 3 on Regular Expressions
- Basic Regular Expression Examples
- Extended Regular Expressions
- Extended Regular Expression Examples

Character classes

- Character classes are lists of characters inside square brackets
- The work the same in regex as they do in globbing
- Character class expressions always match EXACTLY ONE character (unless they are repeated by appending '*')
- [azh] matches "a" or "h" or "z"

Character Classes (cont'd)

- Non-special characters inside the square brackets form a set (order doesn't matter, and repeats don't affect the meaning):
  - `[azh]` and `[zha]` and `[aazh]` are all equivalent
- Special characters lose their meaning when inside square brackets, but watch out for `^`, `]`, and `-` which do have special meaning inside square brackets, depending on where they occur

Character classes (cont'd)

- `^` inside square brackets makes the character class expression mean "any single character UNLESS it's one of these"
- `[^azh]` means "any single character that is NOT `a`, `z`, or `h`"
- `^` has its special "inside square brackets" meaning only if it is the first character inside the square brackets
- `[a^zh]` means `a`, `h`, `z`, or `^`
- Remember, leading `^` outside of square brackets has special meaning "match beginning of line"

Character classes (cont'd)

- `]` can be placed inside square brackets but it has to be first (or second if `^` is first)
- `[]azh]` means `]`, `a`, `h`, or `z`
- `[^]azh]` means "any single character that is NOT `]`, `a`, `h`, or `z`"
- Attempting to put `]` inside square brackets in any other position is a syntax error:
  - `[ab]d]` is a failed attempt at `[ab][d]`
  - `[]` is a failed attempt at `[]]`

Character class ranges (avoid)

- - inside square brackets represents a range of characters, unless it is first or last
- `[az-]` means `a`, `z`, or -
- `[a-z]` means any one character between `a` and `z` inclusive (but what does that mean?)
- "Between `a` and `z` inclusive" used to mean something, because there was only one locale
- Now that there is more than one locale, the meaning of "between `a` and `z` inclusive" is ambiguous because it means different things in different locales

Internationalization (i18n)

- i18n basically means "support for more than one locale"
- Not all computer users use the same alphabet
- When we write a shell script, we want it to handle text and filenames properly for the user, no matter what language they use
- In the beginning, there was ASCII, a 7 bit code of 128 characters
- Now there's Unicode, a table that is meant to assign an integer to every character in the world
- UTF-8 is an implementation of that table, encoding the 7-bit ASCII characters in a single byte with high order bit of 0
- The 128 single-byte UTF-8 characters are the same as true ASCII bytes (both have a high order bit of 0)
- UTF-8 characters that are not ASCII occupy more than one byte, and these give us our accented characters, non-Latin characters, etc
- Locale settings determine how characters are interpreted and treated, whether as ASCII or UTF-8, their ordering, and so on

What is locale

- A locale is the definition of the subset of a user's environment that depends on language and cultural conventions.
- For example, in a French locale, some accented characters qualify as 'lower case alphabetic", but in the old "C" locale, ASCII a-z contains no accented characters.
- Locale is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behavior of components of the system.
- Category names correspond to the following environment variable names (the first three especially can affect the behavior of our shell scripts):
  - *LC_ALL:* Overrides any individual setting of the below categories.
  - *LC_CTYPE*: Character classification and case conversion.
  - *LC_COLLATE*: Collation order.
  - *LC_MONETARY*: Monetary formatting.
  - *LC_NUMERIC*: Numeric, non-monetary formatting.
  - *LC_TIME*: Date and time formats.
  - *LC_MESSAGES*: Formats of informative and diagnostic messages and interactive responses.

Ranges depend on locale

```
$ export LC_ALL=C
$ echo *
A B C Z a b c z
$ echo [a-z]*
a b c z
$ export LC_ALL=en_CA.UTF-8
$ echo *
A a B b C c Z z
$ echo [a-z]*
a B b C c Z z
$
```

POSIX character classes

- Do not use ranges in bracket expressions
- We now use special symbols to represent the sets of characters that we used to represent with ranges.
- These all start with `[:` and end with `:]`
- For example lower case alphabetic characters are represented by the symbol `[:lower:]`
  - `[[:lower:]]` matches any lower case alpha char
  - `[AZ[:lower:]12]` matches `A`, `Z`, `1`, `2`, or any lower case alpha char

POSIX character classes

- **[:alnum:] alphanumeric characters**
- **[:alpha:] alphabetic characters**
- **[:cntrl:]** control characters
- **[:digit:] digit characters**
- **[:lower:] lower case alphabetic characters**
- **[:print:]** visible characters, plus **[:space:]**
- **[:punct:]** Punctuation characters and other symbols
  - !"#$%&'()*+,\-./:;<=>?@[]^_`{|}~
- **[:space:]** White space (space, tab)
- **[:upper:] upper case alphabetic characters**
- **[:xdigit:]** Hexadecimal digits
- **[:graph:] Visible characters (anything except spaces and control characters)**

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` (but grep on the CLS will give an error by default)
  - `[abc[:digit:]]` matches one of a,b,c, or a digit

POSIX character classes (cont'd)

- The exact content of each character class depends on the local language.
- Only for plain ASCII is it true that "letters" means English a-z and A-Z.
- Other languages have other "letters", e.g. é, ç, etc.
- When we use the POSIX character classes, we are specifying the correct set of characters for the local language as per the POSIX description

Gotchas

- Remember any match will be a long as possible
  - `aa*` matches the `aaa` in `xaaax` just once, even though you might think there are three smaller matches in a row
- Unix/Linux regex processing is line based
  - our input strings are processed line by line
  - newlines are not considered part of our input string
  - we have `^` and `$` to control matching relative to newlines

Gotchas (cont'd)

- expressions that match zero length strings
  - remember that the repetition operator `*` means "zero or more"
  - any expression consisting of zero or more of anything can also match zero
  - For example, `x*`, "meaning zero or more `x` characters", will match ANY line, up to `n+1` times, where n is the number of (non-`x`) characters on that line, because there are zero `x` characters before and after every non-`x` character
  - `grep` and regexpal.com cannot highlight matches of zero characters, but the matches are there!

Gotchas (cont'd)

- quoting (don't let the shell change regex before grep sees the regex)

```
$ mkdir empty
$ cd empty
$ grep [[:upper:]] /etc/passwd | wc
    503    2009   39530
$ touch Z
$ grep [[:upper:]] /etc/passwd | wc
      7      29     562
$ touch A
$ grep [[:upper:]] /etc/passwd | wc
     87     343    7841
$ chmod 000 Z
$ grep [[:upper:]] /etc/passwd | wc
grep: Z: Permission denied
     87     343    7841
```

Gotchas (cont'd)

- quoting (don't let the shell change regex before grep sees the regex)

```
$ mkdir empty
$ cd empty
$ grep [[:upper:]] /etc/passwd | wc
    503    2009   39530
$ touch Z
$ grep [[:upper:]] /etc/passwd | wc
      7      29     562
$ touch A
$ grep [[:upper:]] /etc/passwd | wc
     87     343    7841
$ chmod 000 Z
$ grep [[:upper:]] /etc/passwd | wc
grep: Z: Permission denied
     87     343    7841
```

Gotchas (cont'd)

- To explain the previous slide, use `echo` to print out the grep command you are actually running:

```
$ echo grep [[:upper:]] /etc/passwd
grep A Z /etc/passwd

$ rm ?

$ echo grep [[:upper:]] /etc/passwd
grep [[:upper:]] /etc/passwd
```

Gotchas

- we will not use range expressions
- we'll standardize on en_CA.UTF-8 so that the checking script for assignments always sees things formatted the same way
- We don't set locale environment variables in our scripts (why?)

Regex Resources

- http://www.regular-expressions.info/tutorial.html
- http://lynda.com
- http://regexpal.com

Lynda.com

- Some students are already comfortable with the command line
- For those who aren't, yet another tutorial source that might help is Lynda.com
- All Algonquin students have free access to Lynda.com
- Unix for Mac OSX users:

http://www.lynda.com/Mac-OS-X-10-6-tutorials/Unix-for-Mac-OS-X-Users/78546-2.html

Lynda.com and regex

- Lynda.com has a course on regular expressions
- The problem is that it covers our material as well as some more advanced topics that we won't cover
- It is a good presentation, and the following chapters should have minimal references to the "too advanced" material
  - Chapter 2 Characters
  - Chapter 3 Character Sets
  - Chapter 4 Repetition Expressions
- On campus use this URL:

http://www.lynda.com/Regular-Expressions-tutorials/Using-Regular-Expressions/85870-2.html

- Off campus use this URL:

http://wwwlyndacom.rap.ocls.ca/Regular-Expressions-tutorials/Using-Regular-Expressions/85870-2.html

Assignment 3 on regex

- Assignment 3 asks you to write shell scripts
- These are simple scripts: just the script header, and a grep command where coming up with the regex is your work to be done
- You don't need extended regular expression functionality, and the checking script will disallow it
- We will cover extended regular expression functionality below

Basic Regular Expression Examples

- phone number
  - 3 digits, dash, 4 digits

```
[[:digit:]][[:digit:]][[:digit:]]-[[:digit:]][[:digit:]][[:digit:]][[:digit:]]
```

- postal code
  - A9A 9A9

```
[[:upper:]][[:digit]][[:upper:]] [[:digit:]][[:upper:]][[:digit:]]
```

- email address (simplified, lame)
  - someone@somewhere.com
  - domain name cannot begin with digit

```
[[:alnum:]_-][[:alnum:]_-]*@[[:alpha:]][[:alnum:]-]*\.[[:alpha:]][[:alpha:]]*
```

Basic Regular Expression Examples

- any line containing only alphabetic characters (at least one), and no digits or anything else

  `^[[:alpha:]][[:alpha:]]*$`

- any line that begins with digits (at least one)
  - In other words, lines that begin with a digit

  `^[[:digit:]]`

  `^[[:digit:]].*$` would match the exact same lines in grep

- any line that contains at least one character of any kind

.

`^..*$` would match the exact same lines in grep

vi examples

- To do search and replace in vi, can search for a regex, then make change, then repeat search, repeat command
- in `vi (and sed, awk, more, less)` we delimit regular expressions with `/`
- capitalize sentences
  - any lower case character followed by a period and one or two spaces should be replaced by a capital
  - search for `/\.  [[:lower:]]/`
  - then type `4~`
  - then type `n.` as many times as necessary
  - `n` moves to the next occurrence, and `.` repeats the capitalization command

vi examples (cont'd)

- uncapitalize in middle of words
  - any upper case character not preceded by whitespace should be uncapitalized
  - type `/[[:lower:]][[:upper:]]`
  - notice the second `/` is optional and not present here
  - then type `l` to move one to the left
  - type `~` to change the capitalization
  - type `nl.` as necessary
  - the `l` is needed because vi will position the cursor on the first character of the match, which in this case is a character that doesn't change.

Regular Expressions (again)

- Now three kinds of matching
  1. Filename globbing
     - used on shell command line, and shell matches these patterns to filenames that exist
     - used with the `find` command (quote from the shell)
  2. Basic Regular Expressions, used with
     - vi (use delimiter)
     - more (use delimiter)
     - sed (use delimiter)
     - awk (use delimiter)
     - grep (no delimiter, but we quote from the shell)
  3. Extended Regular Expressions
     - less (use delimiter)
     - grep –E (no delimiter, but quote from the shell)
     - perl regular expressions (not in this course)

Regex versus Globbing

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

Extended versus Basic

- All of what we've officially seen so far, except that one use of parenthesis many slides back, are the Basic features of regular expressions
- Now we unveil the Extended features of regular expressions
- In the old days, Basic Regex implementations didn't have these features
- Now, all the Basic Regex implementations we'll encounter have these features
- The difference between Basic and Extended Regular expressions is whether you use a backslash to make use of these Extended features

Repeat preceding (Repetition)

Basic	Extended	Repetition Meaning
*	*	zero or more times
\?	?	zero or one times
\+	+	one or more times
\{n\}	{n}	n times, n is an integer
\{n,\}	{n,}	n or more times, n is an integer
\{n,m\}	{n,m}	at least n, at most m times, n and m are integers

Alternation (one or the other)

- can do this with Basic regex in grep with -e
  - example: `grep -e 'abc' -e 'def' foo.txt`
  - matches lines with `abc` or `def` in `foo.txt`
- `\|` 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 `\`, as in `a|b`

Precedence

- repetition is tightest (think exponentiation)
  - xx* means x followed by x repeated, not xx repeated
- concatenation is next tightest (think multiplication)
  - aa*\|bb* means aa* or bb*
- alternation is the loosest or lowest precedence (think addition)
- Precedence can be overridden with parenthesis to do grouping

Grouping

- `\(` and `\)` can be used to group regular expressions, and override the precedence rules
- For Extended Regular Expressions, leave out the `\`, as in `(` and `)`
- `abb*` means `ab` followed by zero or more `b`'s
- `a\(bb\)*c` means `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`

Precedence rules summary

Operation	Regex	Algebra
grouping	() or \(\)	parentheses brackets
repetition	* or ? or + or {n} or {n,} or {n,m} * or \? or \+ or \{n\} or \{n,\} or \{n,m\}	exponentiation
concatenation	ab	multiplication
alternation	| or |	addition

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 ^
- For the extended functionality,
  - backslash turns it on for basic regex
  - backslash turns it off for extended regex

Tags or Backreferences

- Another extended regular expression feature
- 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
- Example (basic regex):

`\(aa*\)b\1` means any number of `a`'s followed by b followed by exactly the same number of `a`'s

Extended Regex Examples

- phone number
  - 3 digits, optional dash, 4 digits
  - we couldn't do optional single dash in basic regex

```
[[:digit:]]{3}-?[[:digit:]]{4}
```

- postal code
  - A9A 9A9
  - Same as basic regex

```
[[:upper:]][[:digit]][[:upper:]] [[:digit:]][[:upper:]][[:digit:]]
```

- email address (simplified, lame)
  - someone@somewhere.com
  - domain name cannot begin with digit or dash

```
[[:alnum:]_-]+@([[:alpha:]][[:alnum:]-]+\.)+[[:alpha:]]+
```