CST8177 – Linux II

Regular Expressions

Topics

- Basic Regular Expression Examples
- Extended Regular Expressions
- Extended Regular Expression Examples

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:]]`

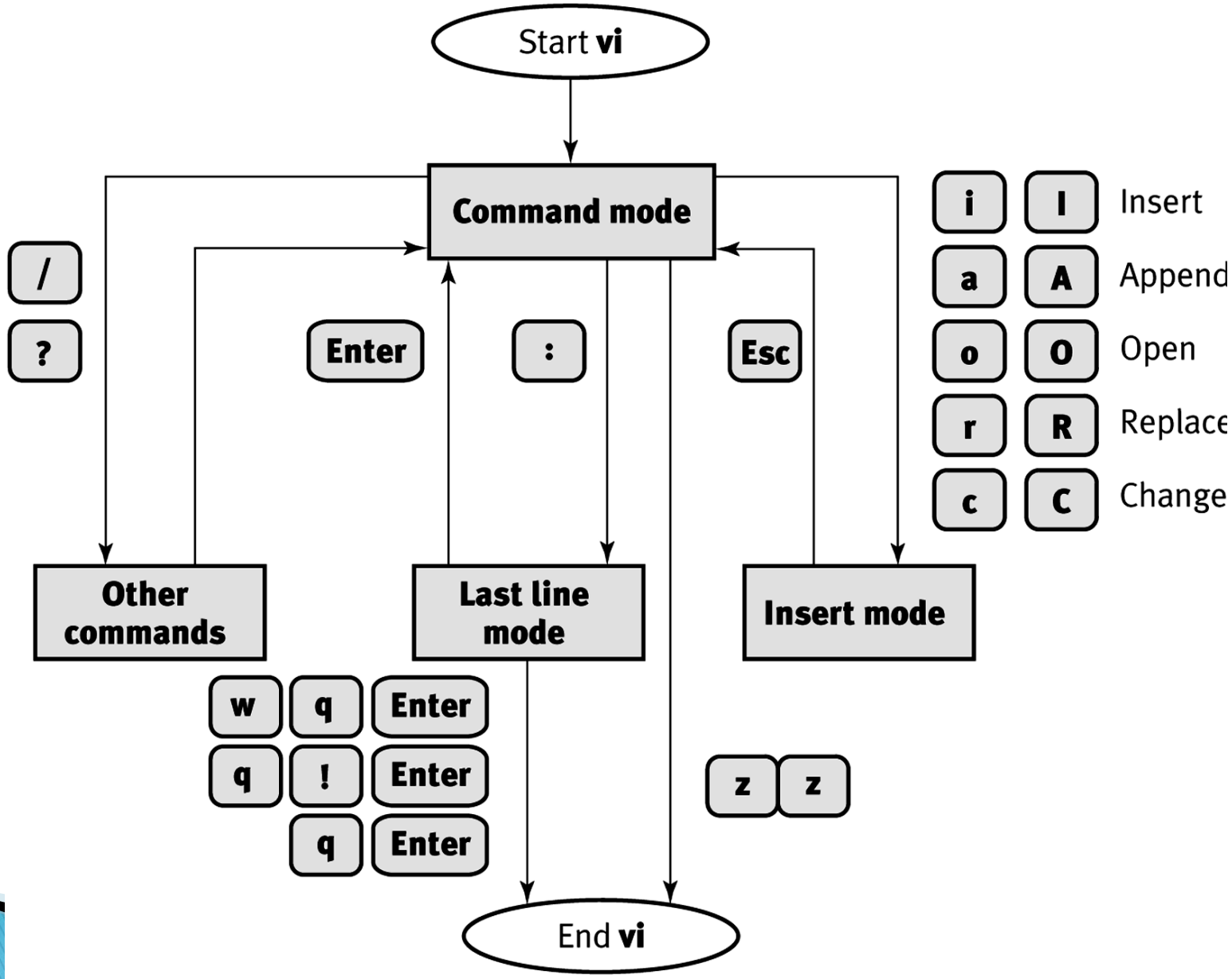
  `^[[:digit:]].*$` would match the exact same lines in grep

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

.

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

Operating modes of the vi text editor



The format of a vi command

The generic syntax: [#1]operation[#2]target

Examples:

Command	Action
5dw	Deletes five words, starting at the current cursor position
7dd	Deletes seven lines, starting at the current line
7o	Opens seven blank lines after the current line
7O	Opens seven blank lines before the current line
c2b	Changes back two words
d7,14	Deletes lines 7 through 14 in the buffer
1G	Puts the cursor on the first line of the file
10yy	Yanks (copies) the next (starting with the current line) 10 lines into a temporary buffer

Ref: http://www.tutorialspoint.com/unix/unix-vi-editor.htm

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 following by a period and 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 following a lower case character should be made lower case
  - type `/[[:lower:]][[:upper:]]`
  - notice the second `/` is optional and not present here
  - then type `l` to move one to the right
  - 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 meta character, 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:]]+
```

Regular Expression Metacharacters

.	Any single character except newline
[...]	Any character in the list
[^...]	Any character not in the list
*	Zero or more of the preceding item
^	Start of the string or line
$	End of the string or line
\<	Start of word boundary
\>	End of word boundary
\(...\)	Form a group of items for tags
\n	Tag number n
\{n\}	Exactly n of preceding item
\{n,\}	n or more of preceding item
\{n,m\}	Between n and m of preceding item
\	The following single character is normal unchanged, , or escaped. Note its use in `[a\-z]`, changing it from `a` to `z` into `a`, `-` or `z`.

Extended metacharacters for egrep

+	One or more of the preceding item
?	None or one (0 or 1) of the preceding item
|	Separates a list of choices (logical OR)
(...)	Form a group of items for lists or tags
n	Tag number *n*
{*n*}	Exactly *n* of preceding item
{*n*,}	*n* or more of preceding item
{*n*,*m*}	Between *n* and *m* of preceding item

Many of these also exist in Regular Expression-intensive languages like Perl. But be sure to check your environment and tools before using any unusual extensions.