little languages

lecture 06:

Real World Regex Operators and VIM 201

Before lecture: **Start VM and pull 590 materials from upstream.**
Then...

```
$ cd comp590-material-<you>
$ git pull upstream master
$ cd comp590-material-<you>/lecture/06-regex
```

Regular Expressions - Additional Operators

- The three operators discussed last lecture are **fundamental**:
  - Concatenation
  - Alternation (Union)
  - Zero or More Repetitions (Closure / Kleene Star)

- There are very common real world patterns you will want to specify that are tedious using only those three operators.

- Most regex implementations offer additional operators for improved ergonomics. The ones we'll see today are built into egrep, Java, JavaScript, Python, etc.

Regex Character Classes - Character Lists (1/3)

- What regular expression matches single characters 'a' through 'f'?

```
r -> a | b | c | d | e | f
```

- Character classes allow you to express the above pattern as:

```
r -> [abcdef]
```

- When you need to match a specific set of individual characters, this is commonly helpful. For example, punctuations:

```
r -> [,.:;]
```

Regex Character Classes - Character Ranges (2/3)

- What regular expression matches single characters 'a' through 'z'?

```
r -> a | b | c | d | e | f | ... | x | y | z
```

- Character classes allow you to express the above pattern as:

```
r -> [a-z]
```

  - How does a regex library *know* the range? It's based on ASCII ordinal numbers for each char. ASCII code for a is 97 and z is 122, so it accepts chars whose ASCII ordinals are between those two numbers.

- You can combine multiple ranges in singular regular expressions. For example, valid hexadecimal digits which are case insensitive:

```
r -> [a-fA-F0-9]
```

Regex Character Classes - Escaping (3/3)

- You can directly capture *'s, ()'s, and |'s in character classes

```
r -> [*()|]
```

- Why? The square brackets signify "treat these characters as character literals."

- You usually need to *escape* the characters [ ] and - to use them inside a character class.
  - How regex implementations handle escaping inside of character classes varies.
  - No point in memorizing, just search references when needed.

Hands-on: Find Pairs of Digits on CS Faculty Page

- At the start of lecture you should have:
```
$ cd comp590-material-<you>
$ git pull upstream master
$ cd comp590-material-<you>/lecture/06-regex
```

- In today's lecture directory there is a file named `cs-faculty`

- Using egrep, find all pair of digits based on the regular definition below. You *should* express this using character class ranges as just shown on the previous page:

```
digit         -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit_pair    -> digit digit
```

```
$ egrep --color 'regular expression' cs-faculty
```

- Check in on **PollEv.com/compunc** with your regular expression.

Aside: Why **egrep** vs grep?

- The classic regular expression search command is **`grep`.**
- Where does the name **`grep`** come from?
  - Remember that non-visual editor named \``ed`\`?
  - In ed you can ***g**lobally* search for ***r**egular **e**xpressions* and ***p**rint* matches: **`g/<re>/p`**
    - Notice ***p*** character for the print command in *ed* is the same as in *dc.*
    - It's still a convention! Ctrl+p or Command+p on windows/mac is the print shortcut.
- **Why not use grep?** The original regular expression syntax required escaping common operators like |, (, and) with \'s. So the pattern (a|b) in grep is \(a\|b\)
  - This is how you still have to specify them using vim's regex features, unfortunately.
- **egrep's regular expression syntax is the same as most modern programming languages' and how we'll present regular expressions in this course.**
  - It's much more pleasant to work with.
  - Trivia: the **e** in **e**grep is from its origin as the "extended regular expression" mode of grep: **grep –E**

Aside - **match**ing Character Ranges in Rust

Not only can you *alternate* patterns in Rust's **`match`** statements, you can match character ranges with ..., too!

```rust
let input = "abcDEfghi;123";
println!("input: {}", input);
let mut some_chars = input.chars();
while let Some(c) = some_chars.next() {
    match c {
        'a' | 'e' | 'i' | 'o' | 'u' => {
            println!("vowel: {}", c);
        }
        'A'...'Z' => {
            println!("capital: {}", c);
        }
        'a'...'z' => {
            println!("lowercase: {}", c);
        }
        _ => {
            println!("other: {}", c);
        }
    }
}
```

When a subject matches multiple patterns, the first match wins.

Here's the output to the code left:

```
input: abcDEfghi;123
vowel: a
lowercase: b
lowercase: c
capital: D
capital: E
lowercase: f
lowercase: g
lowercase: h
vowel: i
other: ;
other: 1
other: 2
other: 3
```

Regex Repetitions - N to M repetitions

- Often you will want a pattern matched between a ranged number of times

  $d_{2\text{-}4}$ `-> r r | r r r | r r r r`

- The **{N,M}** operator provides ***N to M repetitions*** semantics

  d_2 `-> r{2,4}`

- For **at most M** repetitions, 0 inclusive, you can leave off the **N**:

  $d_{<=M}$ `-> r{,M}`

- For **at least N** repetitions, you can leave off the **`M`**

  $d_{>=N}$ `-> r{N,}`

Regex Repetitions - Exactly N repetitions

- Often you will want a pattern matched a specific number of times

  $\mathtt{d_5}$ **`-> r r r r r`**

- You could achieve this with N to M repetitions, but it's redundant:

  $\mathtt{d_5}$ **`-> r{5,5}`**

- The **{N}** operator provides ***N repetitions*** semantics

  $\mathtt{d_5}$ **`-> r{5}`**

Hands-on: Find Phone Numbers on CS Faculty Page

- Using egrep, find all lines containing a phone number.

```
$ egrep --color 'regular expression' cs-faculty
```

- Check in on **PollEv.com/compunc** with your regular expression.

Done? GOLF! Can you think of a way to specify the pattern in fewer characters?

Regex Repetitions - One or More Repetitions

- Often you will want *at least one* of some pattern

```
d -> r r*
```

- Using the N to M Repetitions operator, you could as:

```
d -> r{1,}
```

- This is *so commonly useful,* there's a special **+** operator for it:

```
d-> r+
```

Regex Repetitions - Zero or One - "Optional"

- Often you will want *at most one* of some pattern

```
d -> r | ε
```

- The empty string is **ε** and it matches against nothing.

- Using the N to M Repetitions operator, you could as:

```
d -> r{0,1}
```

- This is *so commonly useful,* there's a special **?** operator for it:

```
d-> r?
```

Regular Expression Operator Precedence

Highest

1. Repetitions (left binding, unary operators)
   - *
   - +
   - ?
   - {N,M}'s

2. Concatenation

3. | Alternations

Lowest

VIM 201

- More VIM locations (introduced last lecture, but let's demo)
- Text Objects
- Registers
- Macros
- Visual Mode

More **vim** Locations

Regular Expression Search

98% of the time you'll only use concatenation.

For the other 2%, you can use the Kleene Star * directly, but you must escape parentheses and alternations, i.e. `b(a|ee*)` is `/b\(a\|ee*\)`

Location	Key
jump to <regex>	/<regex><enter>
next match of last <regex>	n
previous match of <regex>	N

Locations in File

Go to line #<N> above cursor	<N>gg
Go to line #<N> below cursor	<N>G
Jump to the <N>% line of file	<N>%

Char Search Current Line

Find next char <C>	f<C>
Find previous char <C>	F<C>
To next <C>, stopping before it	t<C>
To previous <C>, stop before it	T<C>

vim Grammar - **Text Objects**

```
command     -> CURSOR_TO | operation | LINE_OPERATION | TO_INSERT_MODE

operation   -> N_TIMES? VERB CURSOR_TO | VERB text_object

text_object -> (inside | around) object

inside       -> 'i'

around       -> 'a'

object       -> surrounding | word

surrounding  -> '(' | ')' | '[' | ']' | '{' | '}' | '"'

word          -> 'w'
```

Text Object Operation Examples

"Change Inside Parentheses"

Before:	`foo(1, 2)`
Command:	`ci)`
After:	`foo(█)` (in insert mode)

"Change Around Parentheses"

Before:	`foo(1, 2)`
Command:	`ca)`
After:	`foo█` (in insert mode)

Vim's Registers - Variables that Hold Text

- When you carry out an action, the text under the operation is put into a **register**
    - In many old school unix programs (including *dc!)* a "register" is just a variable whose name is limited to a single character.
    - The only thing it shares in common with the CPU idea of a register is that you have a finite number of them.

- You address registers with the double quote "
    - "a is register a
    - "b is register b
    - "" is register " *and* the default register

- When you yank, change, or delete without a register the text goes in the default register "

- To place the text under the operation into a specific register, just like variable assignment in programming, you first specify the register first then what follows:
    - **"a**y$ - Assign to **register a** the yanked text to the end of the line. (copy)
    - **"b**d$ - Assign to **register b** the text deleted to the end of the line. (cut)
    - **"z**c$ - Assign to **register z** the text deleted when changing to the end of the line. (cut)
    - **"a**p – Paste the contents of **register a.**

vim Grammar - **Registers**

```
command      -> CURSOR_TO | operation | LINE_OPERATION | TO_INSERT_MODE | paste

operation    -> assign_to_register (N_TIMES? VERB CURSOR_TO | VERB TEXT_OBJECT)

paste         -> read_from_register 'p'

assign_to_register    -> register

read_from_register    -> register

register              -> default_register | '"' register_name

default_register      -> ε

register_name         -> [a-z]
```

vim Golf – Get rid of *the next* fax number line

- Starting from the top of the file, what is the fewest number of keystrokes you can think of to get rid of the first fax line?

- Start your cursor in the top left corner: gg

- Respond with your keys on PollEv.com/compunc

vim Macros
Record and Replay strings of commands

- To begin recording a vim macro, press the q key followed by a register name. For example:
  - qa – begin recording a macro in the a register
  - Notice the status bar tells you "recording @a"

- Then, enter your commands as you normally would.

- To stop recording a macro, press the q key again.

- To replay a macro, press the @ symbol followed by the macro name. For example:
  - @a – relays the macro in register a

- Are these the *same* registers as what we cut and copy to? ***YES!!!***
  - You can *paste* your macro into the document!
  - You can also write your macro in your document and then copy it to a register for use as a macro!

vim Grammar - **Macros**

```
command_or_macro -> command | record_macro

command -> CURSOR_TO | OPERATION | … | replay_macro

record_macro       -> 'q' register_name command* 'q'

replay_macro       -> N_TIMES? ('@' register_name | replay_macro_again)

replay_macro_again -> '@' '@'

register_name      -> [a-z]
```

We now have a construct in our grammar that lets us ***compose*** commands together and allows us to define our own compound commands!

Composition is a superpower of languages.

vim Macro Practice – Get rid of *all fax number lines*

- Undo any changes made to the phone-numbers file with 'u'
- Return back to the top of the file: gg
- Record a macro in register f (fax): qfjddq
- Replay the macro in register f 30 times over: 30@f
- Replay the last macro a few more times: @@, @@

vim Macro Practice in phone-numbers

Remove the parenthetical text after each phone number

Surround the first set of numbers in parenthesis

Surround the last set of numbers in parenthesis

Record 3 macros and then replay them all in a 4th macro.

vim Visual Mode 101
Like clicking and dragging your mouse around.

- **v** – Transition to **<u>visual</u>** mode. Select using *location_to* commands.
  - to_register? c – change
  - to_register? y – yank (copy)
  - to_register? d – delete (cut)

- **Shift+v** – Transition to **<u>visual line</u>** mode.
  - Verbs same as above
  - > - Indent
  - < - Unindent

- **Control+v** – Transition to **<u>visual block</u>** mode.
  - Shift+i – Insert in front of block.
    - Comment out block of code: Ctrl+v j j j Shift+i // Ctrl+[
  - Shift+a – Insert after block

vim - A Few More Useful Keys in Normal Mode

- x - Delete the character under the cursor
- <Ctrl>+A – Increase the number under the cursor by 1
- ~ - Toggle the case of the letter under the cursor
- r<char> - Replace the character under the cursor and stay in normal mode
- shift+J - Join the next line onto the end of the current line.
- Ctrl+o - Open the file explorer (this is a custom plugin on the VM called NERDTree and will not exist in all vim editors you use)
- ; - Repeat your last *find (f<char>)* or *to next (t<char)* location_to