CMSC 330: Organization of Programming Languages

Ruby Regular Expressions
String Processing in Ruby

- Scripting languages provide many useful libraries for manipulating strings

- The Ruby **String** class provides many useful, such as
  - Concatenate two strings
  - Grabbing substrings
  - Searching and Replacing
String Operations in Ruby

- What if we want to find more complicated patterns?
  - Find Steve, Stephen, Steven, Stefan, Esteve
  - Count the number of words that have even number vowels

- We need Regular Expressions
Regular Expressions

- A way of describing patterns or sets of strings
  - Searching and matching
  - Formally describing strings
    - The symbols (lexemes or tokens) that make up a language

- Common to lots of languages and tools
  - awk, sed, perl, grep, Java, OCaml, C libraries, etc.
    - Popularized (and made fast) as a language feature in Perl

- Based on some really elegant theory
  - Future lecture
Regular Expressions

- Regular expressions consist of
  - Constants
    - empty set $\emptyset$
    - empty string $\epsilon$
    - literal character $a$ in $\Sigma$, a finite alphabet
  - Operations over these sets
    - Concatenation: $a \cdot b$
    - Union: $a | b$
    - Kleene star: $a^*$
  - We can build complicated patterns by recursively applying the 3 operation on those 3 constants
Example Regular Expressions in Ruby

- /Ruby/
  - Matches exactly the string "Ruby"
  - Concatenation: /r \* u \* b \* y/

- /Ruby | OCaml/
  - Matches either "Ruby" or "OCaml"

- /(ab)*/
  - 0 or more occurrences of “ab”, matches “”, “ab”, “abab”, “ababab”,…
Using Regular Expressions

- Regular expressions are instances of Regexp
- Basic matching using =~ method of String

```ruby
line = gets               # read line from standard input
if line =~ /Ruby/ then    # returns nil if not found
    puts "Found Ruby"
end
```

CMSC 330 - Summer 2020
Repetition in Regular Expressions

- `*`: zero or more
- `+`: one or more
  - so `/e+` is the same as `/ee*/`
- `?:` zero or one occurrence
- `{x}` means repeat the search for exactly `x` occurrences
- `{x,}` means repeat the search for at least `x` occurrences
- `{x, y}` means repeat the search for at least `x` occurrences and at most `y` occurrences
Watch Out for Precedence

- /Ruby*/ means {""", "Ruby", "RubyRuby", ...}

- /Ruby*/ means {"Rub", "Ruby", "Rubyy", ...}

- Best to use parentheses to disambiguate
  - Note that parentheses have another use, to extract matches, as we’ll see later
Character Classes

- `/[abcd]/`
  - `{"a", "b", "c", "d"}` (Can you write this another way?)

- `/[a-zA-Z0-9]/`
  - Any upper or lower case letter or digit

- `/[^0-9]/`
  - Any character except 0-9 (the ^ is like not and must come first)

- `[/\t\n ]/`
  - Tab, newline or space

- `//[a-zA-Z_\$][a-zA-Z_\$0-9]*/`
  - Java identifiers ($ escaped...see next slide)
Special Characters

- any character
- ^ beginning of line
- $ end of line
- \$ just a $
- \d digit, [0-9]
- \s whitespace, [\t\r\n\f\s]
- \w word character, [A-Za-z0-9_]
- \D non-digit, [^0-9]
- \S non-space, [^\t\r\n\f\s]
- \W non-word, [^A-Za-z0-9_]
Potential Character Class Confusions

- ^
  - Inside character classes: *not*
  - Outside character classes: beginning of line

- [ ]
  - Inside regular expressions: character class
  - Outside regular expressions: array
    - Note: [a-z] does not make a valid array

- ( )
  - Inside character classes: literal characters ( )
    - Note /(0..2)/ does not mean 012
  - Outside character classes: used for grouping

- –
  - Inside character classes: range (e.g., a to z given by [a-z])
  - Outside character classes: subtraction
Let \( re \) represents an arbitrary pattern; then:

1. \( /re/ \) – matches regexp \( re \)
2. \( /(re_1|re_2)/ \) – match either \( re_1 \) or \( re_2 \)
3. \( /(re)^*/ \) – match 0 or more occurrences of \( re \)
4. \( /(re)^+/ \) – match 1 or more occurrences of \( re \)
5. \( /(re)?/ \) – match 0 or 1 occurrences of \( re \)
6. \( /(re)\{2}/ \) – match exactly two occurrences of \( re \)
7. \( /\[a-z]/ \) – same as \( (a|b|c|...|z) \)
8. \( /[^0-9]/ \) – match any character that is not 0, 1, etc.
9. \( ^, \$ \) – match start or end of string
Try out regexps at rubular.com
Regular Expression Practice

Contains 2 b's, may not be consecutive.

```
/^       b          b      $/
```

```
/^ [^b]* b [^b]* b [^b]* $/
```

Any number of not b
Regular Expression Practice

- Starts with c, followed by one vowel, and any number of letters

```
/^c                     $/
/^c [aouei] [a-z]* $/
```
Regular Expression Practice

- All letters are in alphabetic order

/^a*b*c*d*e*f*g*h*i*j*k*l*m*n*o*p*r*t*$/
Regular Expression Practice

Contains sss or ccc

/s{3}|c{3}/
Regular Expression Practice

- contains 2 ab

/\(ab\)\{2\}/

- contains 2 b

/\b\{2\}/
Regular Expression Practice

- Starts with a, 0 or 1 letter after that

```
/^a[a-z]?$/
```
Regular Expression Practice

- Contains one or more ab or ba

/\(ab|ba\)\+/
Regular Expression Practice

- steve, steven, or stephen

/^ste(ve|phen|ven)$/
Regular Expression Practice

- Even length string

/^\(\_\_\)\*$/
Regular Expression Practice

- Even number of vowels

/^([^aoeui]*[aoeui][^aoeui]*[aoeui][^aoeui]*)*$/
Regular Expression Practice

- starts with not-b, followed by one or more a’s

/^[^b]+a+$/
How many different strings could this regex match?

/^Hello, Anyone awake?$/

A. 1  
B. 2  
C. 4  
D. More than 4
Quiz 1

How many different strings could this regex match?

/^Hello, Anyone awake?$/

A. 1
B. 2
C. 4
D. More than 4
Which regex is not equivalent to the others?

A. ^[crab]$  
B. ^c?r?a?b?$  
C. ^(c|r|a|b)$  
D. ^([cr]|[ab])$
Which regex is not equivalent to the others?

A. `^[crab]$`
B. `^c?r?a?b?$`
C. `^(c|r|a|b)$`
D. `^([cr]|[ab])$`
Quiz 3

Which string does not match the regex?

/\[a-z\]{4}\d\{3\}/

A. “
cmsc\d\d\d"
B. “cmsc330”
C. “hellocmsc330”
D. “cmsc330world”
Quiz 3

Which string does **not** match the regex?

Recall that without `^` and `$`, a regex will match any **substring**

```
/[a-zA-Z]{4}\d{3}/
```

A. “cmsgs\d\d\d”
B. “cmsg330”
C. “hellocmsg330”
D. “cmsg330world”
Extracting Substrings based on R.E.’s
Method 1: Back References

Two options to extract substrings based on R.E.’s:

- Use back references
  - Ruby remembers which strings matched the parenthesized parts of r.e.’s
  - These parts can be referred to using special variables called back references (named $1, $2,...)
Back Reference Example

```
gets =~ /^Min: \(\d+\) Max: \(\d+\)/
min, max = $1, $2
puts "mini=#{min} maxi=#{max}"
```

Input

Min: 1 Max: 27
Min: 10 Max: 30
Min: 11 Max: 30
Min: a Max: 24

Output

mini=1 maxi=27
mini=10 maxi=30
mini= maxi=
mini= maxi=

Extra space messes up match
Not a digit; messes up match

sets min = $1 and max = $2
Back References are Local

- Warning
  - Despite their names, $1 etc are local variables
  - (Normally, variables starting with $ are global)

```ruby
def m(s)
    s =~ /(Foo)/
    puts $1   # prints Foo
end
m("Foo")
puts $1     # prints nil
```
Back References are Reset

Warning 2

• If another search is performed, all back references are reset to nil

```
gets =~ /(h)e(ll)o/
puts $1
puts $2
gets =~ /h(e)llo/
puts $1
puts $2
gets =~ /hello/
puts $1

hello
h
l
hello
e
nil
nil
```
Quiz 4

What is the output of the following code?

```ruby
s = "help I'm stuck in a text editor"
s =~ /([A-Z]+)/
pus $1
```

A. help
B. I
C. I’m
D. I’m stuck in a text editor
What is the output of the following code?

```ruby
s = "help I'm stuck in a text editor"
s =~ /[A-Z]+/;
puts $1
```

A. help
B. I
C. I’m
D. I’m stuck in a text editor
What is the output of the following code?

```
"Why was 6 afraid of 7?" =~ /\d\s(\w+).*(\d)/
puts $2
```

A. afraid
B. Why
C. 6
D. 7
Quiz 5

What is the output of the following code?

```
“Why was 6 afraid of 7?” =~ /\d\s(\w+)\.*(\d)/
puts $2
```

A. afraid  
B. Why  
C. 6  
D. 7
Method 2: **String.scan**

- Also extracts substrings based on regular expressions.
- Can optionally use parentheses in regular expression to affect how the extraction is done.
- Has two forms that differ in what Ruby does with the matched substrings:
  - The first form returns an array.
  - The second form uses a code block.
    - We’ll see this later.
First Form of the Scan Method

- `str.scan(regexp)`
  - If `regexp` doesn't contain any parenthesized subparts, returns an array of matches
    - An array of all the substrings of `str` which matched

```ruby
s = "CMSC 330 Fall 2018"
s.scan(/\S+ \S+/)
# returns array ["CMSC 330", "Fall 2018"]
```

```ruby
s.scan(/\S{2}/)
# => ["CM", "SC", "33", "Fa", "11", "20", "18"]
```
First Form of the Scan Method (cont.)

• If `regexp` contains parenthesized subparts, returns an array of arrays
  ➢ Each sub-array contains the parts of the string which matched one occurrence of the search
    ```ruby
    s = "CMSC 330 Fall 2018"
s.scan(/(\S+) (\S+)/)  # [['CMSC', '330'], ['Fall', '2018']]
    ```
  ➢ Each sub-array has the same number of entries as the number of parenthesized subparts
  ➢ All strings that matched the first part of the search (or $1$ in back-reference terms) are located in the first position of each sub-array
Practice with Scan and Back-references

```
> ls -l
drwx------- 2 sorelle sorelle 4096 Feb 18 18:05 bin
-rw------- 1 sorelle sorelle 674 Jun  1 15:27 calendar
drwx------- 3 sorelle sorelle 4096 May 11  2006 cmsc311
drwx------- 2 sorelle sorelle 4096 Jun  4 17:31 cmsc330
drwx------- 1 sorelle sorelle 4096 May 30 19:19 cmsc630
drwx------- 1 sorelle sorelle 4096 May 30 19:20 cmsc631
```

Extract just the file or directory name from a line using

- **scan**
  ```
  name = line.scan(/\S+$/)  # [“bin”]
  ```

- **back-references**
  ```
  if line =~ /\S+$/
    name = $1  # “bin”
  end
  ```
What is the output of the following code?

```ruby
s = "Hello World"
t = s.scan(/\w{2}/).length
puts t
```

A. 3
B. 4
C. 5
D. 6
What is the output of the following code?

```ruby
s = "Hello World"
t = s.scan(/\w{2}/).length
puts t
```

A. 3

B. 4

C. 5

D. 6
Quiz 7

What is the output of the following code?

```ruby
s = "To be, or not to be!"
a = s.scan(/(\S+) (\S+)/)
puts a.inspect
```

A. ["To", "be," , "or", "not", "to", "be!"]
B. [["To", "be,"], ["or", "not"], ["to", "be!"]]
C. ["To", "be,"]
D. ["to", "be!"]
What is the output of the following code?

```ruby
s = "To be, or not to be!"
a = s.scan(/(\S+) (\S+)/)
puts a.inspect
```

A. ["To", "be," "or", "not", "to", "be!"]
B. [["To", "be,"], ["or", "not"], ["to", "be!"]]
C. ["To", "be,"]
D. ["to", "be!"]
Second Form of the Scan Method

- Can take a **code block** as an optional argument

\[
\text{str.scan(regexp) \{ |match| block \}}
\]

- Applies the code block to each match
- Short for **str.scan(regexp).each \{ |match| block \**
- The regular expression can also contain parenthesized subparts
Example of Second Form of Scan

<table>
<thead>
<tr>
<th>12</th>
<th>34</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>77</td>
<td>87</td>
</tr>
<tr>
<td>11</td>
<td>98</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>0</td>
</tr>
</tbody>
</table>

input file:
will be read line by line, but
column summation is desired

```
sum_a = sum_b = sum_c = 0
while (line = gets)
    line.scan(/(\d+)/s+(\d+)/s+(\d+)/) { |a,b,c|
        sum_a += a.to_i
        sum_b += b.to_i
        sum_c += c.to_i
    }
end
printf("Total: %d %d %d\n", sum_a, sum_b, sum_c)
```

Sums up three columns of numbers
Practice: Amino Acid counting in DNA

Write a function that will take a filename and read through that file counting the number of times each group of three letters appears so these numbers can be accessed from a hash.

(assume: the number of chars per line is a multiple of 3)

gcg gcattcag cacccgtata actgtta agca atccag atttttt gtgtataa cata accc gcgc
cata ctga agcattc attg aggct agcgt gata aacagtgt agcgt aacca at gggg gaatg
tggca atacc ggtgc gatt acta aga gccggg accac acac ac ccgc taag gat ggag cgtgg
taacata atacc gttc aagcg catggg aggt ggag atgt gtt ccagta aaga atag tgg
gggcct acta cccat ggtacata atta ag gagat cggt caac ttgagac ggtc aatgtg tac
cgag acatat at at cact caact cccgg acgt at gcgt at actggt cact cgttt actg ggcga
def countaa(filename)
    file = File.new(filename, "r")
    lines = file.readlines
    hash = Hash.new
    lines.each{|line|
        acids = line.scan(/.../)
        acids.each{|aa|
            if hash[aa] == nil
                hash[aa] = 1
            else
                hash[aa] += 1
            end
        }
    }
end