CMSC 330: Organization of Programming Languages

Ruby Regular Expressions
String Processing in Ruby

- Earlier, we motivated scripting languages using a popular application of them: string processing
- The Ruby **String** class provides many useful methods for manipulating strings
  - Concatenating them, grabbing substrings, searching in them, etc.
- A key feature in Ruby is its native support for regular expressions
  - Very useful for parsing and searching
  - First gained popularity in Perl
String Operations in Ruby

- "hello".index("l", 0)
  - Return index of the first occurrence of string in s, starting at n
- "hello".sub("h", "j")
  - Replace first occurrence of "h" by "j" in string
  - Use gsub ("global" sub) to replace all occurrences
- "r1\tr2\t\tr3".split("\t")
  - Return array of substrings delimited by tab

Consider these three examples again
- All involve searching in a string for a certain pattern
- What if we want to find more complicated patterns?
  - Find first occurrence of "a" or "b"
  - Split string at tabs, spaces, and newlines
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
Example Regular Expressions in Ruby

- `/Ruby/`
  - Matches exactly the string "Ruby"
  - Regular expressions can be delimited by /’s
  - Use \ to escape /’s in regular expressions

- `/Ruby|OCaml|Java)/`
  - Matches either "Ruby", "OCaml", or "Java"

- `/Ruby|Regular)/` or `/R(uby|egular)/`
  - Matches either "Ruby" or "Regular"
  - Use ( )’s for grouping; use \ to escape ( )’s
Using Regular Expressions

- Regular expressions are instances of Regexp
  - We’ll see use of a Regexp.new later
- 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

offset = line.index(/(MAX|MIN)/)   # search starting from 0
line.sub(/(Perl|Python)/, "Ruby")  # replace
line.split(/((\t|\n)| )/)            # split at tab, space, newline
```

- Can use regular expressions in index, search, etc.
Repetition in Regular Expressions

- /Ruby\/*/
  - \{"", "Ruby", "RubyRuby", "RubyRubyRuby", ... \}
  - * means zero or more occurrences

- /Ruby+/
  - \{"Ruby", "Rubyy", "Rubyyy", ... \}
  - + means one or more occurrence
  - so /e+/
    - is the same as /ee*/

- /Ruby\/?/
  - \{"", "Ruby"\}
  - ? means optional, i.e., zero or one occurrence
Repetition in Regular Expressions

- `/\(Ruby\)\{3\}`
  - `{“RubyRubyRuby”}`
  - `{x}` means repeat the search for exactly x occurrences

- `/\(Ruby\)\{3,\}`
  - `{“RubyRubyRuby”, “RubyRubyRubyRubyRuby”,...}`
  - `{x,}` means repeat the search for at least x occurrences

- `/\(Ruby\)\{3, 5\}`
  - `{“RubyRubyRuby”, “RubyRubyRubyRubyRuby”, “RubyRubyRubyRubyRubyRuby”}`
  - `{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", ...}`

In general
- `*` \{n\} and `+` bind most tightly
- Then concatenation (adjacency of regular expressions)
- Then `|`

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  Using /\^pattern$/ ensures entire string/line must match pattern
^  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
Summary

Let \textit{re} represents an arbitrary pattern; then:

- \texttt{/re} – matches regexp \textit{re}
- \texttt{/(re_1|re_2)/} – match either \textit{re}_1 or \textit{re}_2
- \texttt{/(re)/} – match 0 or more occurrences of \textit{re}
- \texttt{/(re)+/} – match 1 or more occurrences of \textit{re}
- \texttt{/(re)?/} – match 0 or 1 occurrences of \textit{re}
- \texttt{/(re){2}/} – match exactly two occurrences of \textit{re}
- \texttt{/[a-z]/} – same as \texttt{(a|b|c|...|z)}
- \texttt{/[^[0-9]/} – match any character that is not 0, 1, etc.
- \texttt{^, $} – match start or end of string
Try out regexps at rubular.com
Regular Expression Practice

- Make Ruby regular expressions representing
  - All lines beginning with a or b: `/^\(a|b\)/`
  - All lines containing at least two (only alphabetic) words separated by white-space: `/[a-zA-Z]+s+[a-zA-Z]+/`
  - All lines where a and b alternate and appear at least once: `/^((ab)+ a?) | ((ba)+ b?)$/`
  - An expression which would match both of these lines (but not radically different ones):
    - CMSC330: Organization of Programming Languages: Fall 2016
    - CMSC351: Algorithms: Fall 2016
Quiz 1

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

ε or nothing

Matches any character
Quiz 2

Which regex is not equivalent to the others?

A. ^[computer]$  
B. ^(c|o|m|p|u|t|e|r)$  
C. ^([comp]|[uter])$  
Quiz 2

Which regex is not equivalent to the others?

A. ^[computer]$  
B. ^(c|o|m|p|u|t|e|r)$  
C. ^([comp]|[uter])$  
Quiz 3

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

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

A. “cmsc\d\d\d”
B. “cmsc330”
C. “hello\_cmsc330”
D. “cmsc330world”
Quiz 3

Which string does not match the regex?

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

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

A. "cmsg\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

- Extract information from a report

```ruby
gets =~ /^Min: (\d+)  Max: (\d+)$/
min, max = $1, $2
```

- Warning
  - Despite their names, $1 etc are local variables

```ruby
def m(s)
    s =~ /\(Foo\)/
    puts $1  # prints Foo
end
m("Foo")
puts $1  # prints nil
```

sets min = $1 and max = $2
Another Back Reference Example

Warning 2

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

```
gets =~ /(h)(e)(l)lo/ puts $1
puts $2
gets =~ /h(e)llo/ puts $1
puts $2
gets =~ /hello/ puts $1
```

```
hello
h
l
hello
```
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
Quiz 4

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
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
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

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

\begin{center}
\begin{verbatim}
s = "CMSC 330 Fall 2007"
s.scan(/\S+ \S+/)
# returns array ["CMSC 330", "Fall 2007"]
\end{verbatim}
\end{center}

- Note: these strings are chosen sequentially from as yet unmatched portions of the string, so while “330 Fall” \textit{does} match the regular expression above, it is \textit{not} returned since “330” has already been matched by a previous substring.

\begin{center}
\begin{verbatim}
s.scan(/\S\{2}/)
# => ["CM", "SC", "33", "Fa", "ll", "20", "07"]
\end{verbatim}
\end{center}
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 2007"
  s.scan(/(\S+) (\S+)/)  # ["CMSC", "330"],
  # ["Fall", "2007"]
  ```

  ➢ 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

```
-rw-------  1 sorelle  sorelle  674 Jun  1 15:27 calendar
```

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
Quiz 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

```
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

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
### Standard Library: File

- **Lots of convenient methods for IO**
  - `File.new("file.txt", "rw")`  # open for rw access
  - `f.readline`  # reads the next line from a file
  - `f.readlines`  # returns an array of all file lines
  - `f.eof`  # return true if at end of file
  - `f.close`  # close file
  - `f << object`  # convert object to string and write to f
  - `$stdin, $stdout, $stderr`  # global variables for standard UNIX IO

  By default stdin reads from keyboard, and stdout and stderr both write to terminal

- **File inherits some of these methods from IO**
Exceptions

- Use `begin...rescue...ensure...end`
  - Like `try...catch...finally` in Java

```ruby
begin
  f = File.open("test.txt", "r")
  while !f.eof
    line = f.readline
    puts line
  end
rescue Exception => e
  puts "Exception:" + e.to_s + " (class " + e.class.to_s + ")"
ensure
  f.close if f != nil
end
```
Command Line Arguments

- Stored in predefined global constant ARGV

- Example
  - If
    - Invoke test.rb as “ruby test.rb a b c”
  - Then
    - ARGV[0] = “a”
    - ARGV[1] = “b”
    - ARGV[2] = “c”
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)

gcggcattcagcacccgtataactgttaagcaatccagatgtgtgtataacataccggc
catactgaagcattcattgaggctagcgctgataacagtagcgctaataggggaatgtggtggcaatacggtgcgattactaagagcagccgggaccacacacaccgcggtaagggtgagccgtggtaacataaatccgttcaagcagtggcgaaggtggagatgttccagtaagaatagttgg.taacahtataatccgttcaagcagtggcgaaggtggagatgttccagtaagaatagttg
gggccctactacccatgggtacataattaagagatcggtcaatccttgagacgggtcaatggtac
cgagactatatcatctcaactccggacgtatgcgctttactgtggtcacctcgttactgacgga
Practice: Amino Acid counting in DNA

```ruby
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
```

- get the file handle
- array of lines from the file
- for each line in the file
- for each triplet in the line
- initialize the hash, or you will get an error when trying to index into an array with a string
- get an array of triplets in the line
Comparisons

- Sorting requires ability to compare two values
- Ruby comparison method `<=>`
  - `-1 = less`
  - `0 = equals`
  - `+1 = greater`

- Examples
  - `3 <=> 4` returns `-1`
  - `4 <=> 3` returns `+1`
  - `3 <=> 3` returns `0`
Sorting

- Two ways to sort an Array
  - Default sort (puts values in ascending order)
    - `[2,5,1,3,4].sort` # returns `[1,2,3,4,5]`
  - Custom sort (based on value returned by code block)
    - `[2,5,1,3,4].sort { |x,y| y <=> x }` # returns `[5,4,3,2,1]`
    - Where -1 = less, 0 = equals, +1 = greater
    - Code block return value used for comparisons
Ruby Summary

- Interpreted
- Implicit declarations
- Dynamically typed
- Built-in regular expressions
- Easy string manipulation
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...

Makes it quick to write small programs

Hallmark of scripting languages
Other Scripting Languages

- Perl and Python are also popular scripting languages
  - Also are interpreted, use implicit declarations and dynamic typing, have easy string manipulation
  - Both include optional “compilation” for speed of loading/execution
- Will look fairly familiar to you after Ruby
  - Lots of the same core ideas
  - All three have their proponents and detractors
  - Use whichever language you personally prefer
Example Perl Program

```perl
#!/usr/bin/perl
foreach (split(/, $ARGV[0])) {
    if ($G{$_}) {
        $RE .= "\" . $G{$_};
    } else {
                     join("\"\", values(%G)) . ")\w' : '(\w)';
        $G{$_} = ++$N;
    }
}  
```

CMSC 330 - Summer 2018
Example Python Program

```python
#!/usr/bin/python
import re
list = ("deep", "deer", "duck")
x = re.compile("^\S{3,5}.[aeiou]"")
for i in list:
    if re.match(x, i):
        print i
    else:
        print 52
```