CMSC 330: Organization of Programming Languages

Ruby Regular Expressions

Last Lecture

- Ruby language
  - Implicit variable declarations
  - Dynamic typing
  - Many control statements
  - Classes & objects
  - Strings

Introduction

- Ruby language
  - Regular expressions
    - Definition & examples
    - Back references
    - Scan
  - Code blocks
  - File
  - Exceptions

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.
- 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

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

- Can use regular expressions in index, search, etc.

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

Using Regular Expressions (cont.)

- Invert matching using !~ method of String
  - Matches strings that don't contain an instance of the regular expression

  - s = “hello”
  - s !~ /hello/ => false
  - s !~ /hel/ => false
  - s !~ /hello!/ => true
  - s !~ /bye/ => true

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”, “RubyRubyRubyRuby”, …}
  - {x,} means repeat the search for at least x occurrences
- /(Ruby){3, 5}/
  - {“RubyRubyRuby”, “RubyRubyRubyRuby”, “RubyRubyRubyRubyRuby”}
  - {x, y} means repeat the search for at least x occurrences and at most y occurrences

Watch Out for Precedence

- /(Ruby)*/ means {"", "Ruby", "RubyRuby", ...}
  - But /Ruby*/ matches {"Rub", "Ruby", "Rubyy", ...}
- In general
  - * {n} and + bind most tightly
  - Then concatenation (adjacency of regular expressions)
  - Then |
- Best to use parentheses to disambiguate

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_]

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

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 *re* represents an arbitrary pattern; then:
  - /*re*/ – matches regexp *re*
  - /(*re*$_1$|*re*$_2$)/ – match either *re*$_1$ or *re*$_2$
  - /(*re*)*/ – match 0 or more occurrences of *re*
  - /(*re*)+/ – match 1 or more occurrences of *re*
  - /(*re*)?/ – match 0 or 1 occurrences of *re*
  - /(*re*){2}/ – match exactly two occurrences of *re*
  - /[a-z]/ – same as (a|b|c|...|z)
  - / [^0-9]/ – match any character that is not 0, 1, etc.
  - ^, $ – match start or end of string

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 2007
    - CMSC351: Algorithms: Fall 2007

Regular Expression Coding Readability

```
> 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 12:19 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
```

What if we want to specify the format of this line exactly?

```
/^(d|-)(r|-)(w|-)(x|-)(r|-)(w|-)(x|-)(r|-)(w|-)(x|-)
(\s+)(\d+)(\s+)(\w+)(\s+)(\w+)(\s+)(\d+)(\s+)(Jan|Feb
|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)(\s+)(\d\d)
(\s+)(\d\d:\d\d)(\s+)(\S+)$/
```

This is unreadable!

Regular Expression Coding Readability

Instead, we can do each part of the expression separately and then combine them:

```
oneperm_re = '((r|-)(w|-)(x|-))'
permissions_re = '(d|-)' + oneperm_re + '{3}'
month_re = '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)'
day_re = '\d{1,2}';   time_re = '(\d{2}:\d{2})'
date_re = month_re + '\s+' + day_re + '\s+' + time_re
total_re = '\d+';  user_re = '\w+';   group_re = '\w+'
space_re = '\d+';  filename_re = '\S+'

line_re = Regexp.new('^' + permissions_re + '\s+' + total_re
      + '\s+' + user_re + '\s+' + group_re + '\s+' +
      space_re + '\s+' + date_re + '\s+' + filename_re + '$')

if line =~ line_re
        puts "found it!"
end
```

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

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

```
sets min = $1
 and max = $2
```

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

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

Another Back Reference Example

- 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
ll
hello
e
nil
hello
nil
```

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 which 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

```
s = "CMSC 330 Fall 2007"
s.scan(/\S+ \S+/)
# returns array ["CMSC 330", "Fall 2007"]
```

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

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

```
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
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
```

Revisiting Code Blocks

- Recall our earlier code block example with arrays

```
a = [1,2,3,4,5]
a.each { |x| puts x }
```

- A code block is a piece of code that is invoked by another piece of code
  - In this case, the { |x| puts x } code is called five times by the each method
- Code blocks are useful for encapsulating repetitive computations

Second Form of the Scan Method

- The scan method can also take a code block as an argument
  - Rather than returning an array of matches

- 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

```
12  34  23
19  77  87
11  98  3
2   45  0
```

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)
```

converts the string
to an integer

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

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)

```
gcggcattcagcacccgtatactgttaagcaatccagattttgtgtataacataccggc
catactgaagcattcattgaggctagcgctgataacagtagcgctaacaatgggggaatg
tggcaatacggtgcgattactaagagccgggaccacacaccccgtaaggatggagcgtgg
taacataataatccgttcaagcagtgggcgaaggtggagatgttccagtaagaatagtgg
gggcctactacccatggtacataattaagagatcgtcaatcttgagacggtcaatggtac
cgagactatatcactcaactccggacgtatgcgcttactggtcacctcgttactgacgga
```

Practice: Amino Acid counting in DNA

get the file handle

array of lines from the file

for each line in the file

for each triplet in the line

```
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
```

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