CSc 372

Comparative Programming Languages

30 : Ruby — Regular Expressions

Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

The String#scan method

- `str.scan(RE)` iterates through the string `str` matching the regular expression.
- It can be called as a function, returning an array of results.
- It can also be called with a block attached in which case the block gets invoked for every match with the matched substring as the argument.

```
d = "Duckburg, Calistona"

puts d.scan(/.../)

d.scan(/.../) do |x|
   puts x
end
```

The `String#match` method

- `str.match(RE)` returns the first substring of `str` that matches, or `nil` if there's no match.

```
puts d.match(/./)

puts d.match(/.$/)

puts d.match(/^[a-zA-z]*/)

puts d.match(/[a-zA-z]*$/)
```

The String#split method

- `str.split(RE)` splits `str` wherever the regular expression matches. The results are returned as an array.

```
d = "Duckburg, Calistona"
puts d.split(/ /)
puts d.split(/[, ]/)
puts d.split(/[A-Z]/)
puts d.split(//)
puts "donald@duckburg.edu".split(/[@\.]/)
puts "donald@cs.duckburg.edu".split(/[@\.]/)
```

The String#sub/gsub methods

- `str.sub(RE,rep)` returns a copy of `str` where the first occurence of `RE` has been replaced with `rep`.
- `str.sub(RE) {|x| ...}` instead passes a block to the method, and the block returns what should be replaced.
- `gsub` is similar but replaces *all* matches in the string.
- `sub!` and `gsub!` perform the substitutions in-place.

```
puts "donald@cs.duckburg.edu".sub(
          /duckburg/,"QuackU")
puts "donald@cs.duckburg.edu".sub(/[a-z]/) {
          |x| x.upcase}
puts "donald@cs.duckburg.edu".gsub(/./) {
          |x| x.upcase}
```

Grouping syntax

- You can group parts of REs using parentheses. Whatever matches the groups will be assigned to special variables numbered 1,2,. . . .
- Within the RE you can refer to these groups as \1, \2, . . ., and outside as `$1,$2,...`.

```
a = "donald@cs.duckburg.edu"
a =~ /([a-z]+)@([a-z\.]+)/
puts $1
puts $2

puts a.sub(/([a-z]+)@([a-z\.]+)/,
           'username=\1,host=\2')

puts "odandl".gsub(/(.)(.)/,'\2\1')
```

Pattern Syntax: Anchors

- ^ matches the beginning of the line, $ the end of the line.

```
f = "One duck\nTwo duck\nRed duck\nBlue duck!"

puts f.gsub(/^[A-Za-z]+/,"Many")

puts f.gsub(/[A-Za-z]+$/,"fish")
```

Pattern Syntax: Character classes

- `[...]` is a set of characters. It matches any character in the set.
- `[^...]` negates the character class. `[^0-9]`, for example, is the set of all characters *except* the digits.
- There are some standard abbreviations:
  - `\d` ≡ `[0-9]`
  - `\D` ≡ `[^0-9]`
  - `\s` ≡ `[␣\t\r\n\f]`
  - `\S` ≡ `[^␣\t\r\n\f]`
  - `\w` ≡ `[A-Za-z0-9]`
  - `\W` ≡ `[^A-Za-z0-9]`

Pattern Syntax: repetition

- RE+ matches one or more of RE.
- RE∗ matches zero or more of RE.
- RE? matches zero or one of RE.

```
f = "DuckDuckDuckDuckFishDuck"

puts f.sub(/(Duck)+/,"NO MORE DUCKS!")

puts f.gsub(/(Duck)+/,"NO MORE DUCKS!")

puts f.gsub(/(Duck)*/,"NO MORE DUCKS!")

puts f.sub(/(Duck)?/,"NO MORE DUCKS!")
```

Pattern Syntax: repetition...

- RE$\{m\}$ matches exactly m RE.
- RE$\{m, n\}$ matches exactly $m \dots n$ RE.

```
f = "DuckDuckDuckDuckFishDuck"

puts f.sub(/(Duck){2}/,"NO MORE DUCKS!")

puts f.sub(/(Duck){2,3}/,"NO MORE DUCKS!")
```

Pattern Syntax: Alternation

- $\mathtt{RE}_1$|$\mathtt{RE}_2$ matches either RE.

```
f = "DuckDuckMonkeykDuckFishDuck"

puts f.gsub(/(Duck)|(Fish)/,"Banana")
```

Exercise

- Write a function `filepath(path)` that parses a unix filename (directory-names separated by slashes ending in a filename) and returns them as an array:

```
puts filepath("aaa/bbb/ccc/ddd.txt")
puts filepath("a%$aa/b&*bb/c$@!cc/dd++d.txt")
```

  should print

```
aaa
bbb
ccc
ddd.txt

a%$aa
b&*bb
c$@!cc
dd++d.txt
```

  HINT: Use `scan` and character class inversion.

Exercise

- Do the same as the previous exercise, but use `split` instead.

Exercise

- Write a function `BSOD(path)` that turns a Unix filepath into a Windows one, by replacing all forward slashes by backslashes:

  ```
  puts BSOD("aaa/bbb/ccc/ddd.txt")
  ```

  should print

  ```
  aaa\bbb\ccc\ddd.txt
  ```

Exercise

- Write a function `protocol(url)` that returns the protocol part of a `url`. If no protocol is found, return `"http"`. For example, these calls

  ```
  puts protocol("http://www.cs.arizona.edu")
  puts protocol("https://www.cs.arizona.edu")
  puts protocol("file://www.cs.arizona.edu")
  puts protocol("www.cs.arizona.edu")
  ```

  should print

  ```
  http
  https
  file
  http
  ```

  HINT: Use the grouping syntax.

Exercise

- Write a function `parseURL(url)` which splits a URL in three pieces and returns them as an array: the protocol, the address, and the file path. For example, this call

  ```
  puts parseURL("http://www.cs.az.edu/~collberg/i.html")
  puts parseURL("www.cs.az.edu/~collberg/i.html")
  ```

  should print

  ```
  http
  www.cs.az.edu
  /~collberg/i.html

  nil
  www.cs.az.edu
  /~collberg/i.html
  ```

  HINT: Use the grouping syntax and the ?-operator.

Exercise

- Write a function `parseHTML(htm)` which takes a piece of HTML such as `<b>hello!</b>` and returns the list `[tag,contents]` if the HTML is valid (i.e., the tags match), and `nil` otherwise. Examples:

  ```
  puts parseHTML("<b>hello there!</b>")
  puts parseHTML("<b>hello there!</burp>")
  ```

  should print

  ```
  b
  hello there!

  nil
  ```

  HINT: Use the grouping syntax and backslash sequences.

Exam Problem I(a) — 372 Fall 2008

Write a Ruby function which parses floating point numbers. It should be defined like this:

```
def parse(f)
   ...
end
```

Here are some examples:

```
parse("+1.44E+10")  ⇒  ["+1","44","+10"]
parse("1.44E10")    ⇒  ["1","44","10"]
parse("+1.44")      ⇒  ["+1","44",nil]
parse("+1.0")       ⇒  ["+1","0",nil]
parse("1.0")        ⇒  ["1","0",nil]
parse("1.")         ⇒  nil
parse(".0")         ⇒  nil
parse("a.0")        ⇒  nil
parse("1.0E")       ⇒  nil
```

Exam Problem I(a) — 372 Fall 2008

I.e., on success (the floating point number has the correct syntax) `parse` returns an array of three strings: the part before the decimal point, the part after the decimal point, and the exponent part (if any). If there's no exponent, that part is returned as `nil`.
If the syntax of the input is wrong, `parse` returns `nil`.
In particular, the syntax of a floating-point number conforms to these rules:

1. There's an optional + or - sign.
2. There's at least one digit before the decimal point.
3. There's at least one digit after the decimal point.
4. The exponent is optional, can start with `E` or `e`, can have an optional + or - sign, and must have at least one digit.

Implement your function using *one* regular expression.

Exam Problem I(b) — 372 Fall 2008

Extend the `parse` function from the previous problem so that it will either return its result as an array, or will yield the results, one at a time, if called with a block. Here are some examples:

```
parse("+1.44E+10") do |x|
   puts x
end
puts "-------------"
parse("1.44") do |x|
   puts x
end
puts "-------------"
parse("monkey") do |x|
   puts x
end
puts "-------------"
```

which should produce this output:

Exam Problem I(c) — 372 Fall 2008

Assume that we've defined a class `Degree` which represents a degree someone might have. We'll assume that there are only three kinds of degrees, `"BS"`, `"MS"`, and `"PHD"`. It's possible to compare two degrees to see which one is higher. To illustrate, these statements

```
bs = Degree.new("BS")
ms = Degree.new("MS")
phd = Degree.new("PHD")

puts bs < ms
puts bs < phd
puts ms < phd
puts phd == phd
puts bs <= phd
puts phd > phd
puts phd <=> ms
puts phd.to_s()
```

Exam Problem I(c) — 372 Fall 2008

Write a Ruby class `Degrees` which encapsulates an array of the degrees (instances of the `Degree` class above) that a person has. The following operations should be supported:

1. You should be able to compare two persons' sets of degrees to see who is the better educated. A set of degrees A is considered better than a set of degrees B if A's *highest* degree is higher than B's *highest* degree.
2. You should be able to perform standard collection operations such as `sort()`, `collect()`, `inject`, `member`, and `find`, on instances of `Degrees`.

Exam Problem I(c) — 372 Fall 2008

To illustrate the first point, consider these statements:

```
bob = Degrees.new([Degree.new("BS"),Degree.new("MS")])
alice = Degrees.new([Degree.new("BS"),Degree.new("PHD")])
charles = Degrees.new([Degree.new("PHD"),Degree.new("BS"),De
carol = Degrees.new([])

puts bob < alice
puts charles < alice
puts carol < bob

if charles == alice then
   puts "Charles and Alice have the same highest degree"
end
if alice > bob then
   puts "Alice is higher educated than Bob"
end
```

Exam Problem I(c) — 372 Fall 2008

They should produce this output:

```
true
false
true
Charles and Alice have the same highest degree
Alice is higher educated than Bob
```

Exam Problem I(c) — 372 Fall 2008

To illustrate the second point, consider these statements:

```
puts "Alice has these degrees: " + alice.inject() {|x,y| x.t
puts "Charles has " + charles.inject(0) {|x,y| 1+x}.to_s + '
puts charles.collect() {|x| "Charles has a " + x.to_s}
if charles.member?(Degree.new("BS")) then
   puts "Charles has a BS degree"
end
```

They should produce this output:

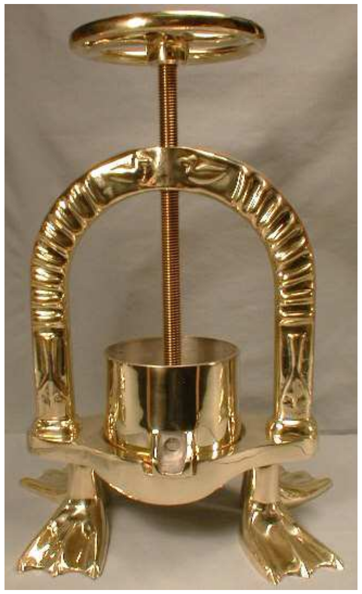
```
Alice has these degrees: BS,PHD
Charles has 3 degrees
Charles has a BS
Charles has a MS
Charles has a PHD
Charles has a BS degree
```

Readings

- Read Chapter 2, page 19–20, in *Programming Ruby*.
- Read Chapter 5, page 59–77, in *Programming Ruby — The Pragmatic Programmers Guide*, by Dave Thomas.
- The Ruby `String`-class, page 606-625 in *Programming Ruby*.

French Duckpress— $1559.99

A Duck Press is used to press out the juice, which is used as seasoning over the meat slices.



Generally, the duck is cooked for around 20 minutes, the cooked breast meat is sliced for serving, and the partially cooked legs are finished cooking separately. The carcass is then pressed, together with some good red wine, brandy, etc., and the resulting juices poured over the slices. From `http://www.fantes.com/duck_press.htm`.