

CMSC 330: Organization of Programming Languages

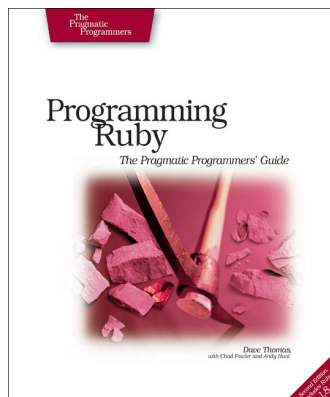
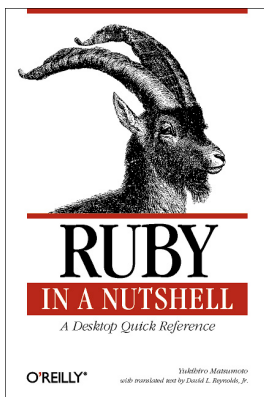
Ruby and Regular Expressions

Introduction

- Ruby is an *object-oriented, imperative scripting language*
 - “I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That’s why I decided to design my own language.”
 - “I believe people want to express themselves when they program. They don’t want to fight with the language. Programming languages must feel natural to programmers. I tried to make people enjoy programming and concentrate on the fun and creative part of programming when they use Ruby.”

– Yukihiro Matsumoto (“Matz”)

Books on Ruby



- Earlier version of Thomas book available on web
 - See course web page

Applications of Scripting Languages

- Scripting languages have many uses
 - Automating system administration
 - Automating user tasks
 - Quick-and-dirty development
- Major application:

Text processing



Output from Command-Line Tool

```
% wc *
 271    674    5323 AST.c
  100    392    3219 AST.h
  117   1459   238788 AST.o
 1874   5428   47461 AST_defs.c
 1375   6307   53667 AST_defs.h
  371    884    9483 AST_parent.c
  810   2328   24589 AST_print.c
  640   3070   33530 AST_types.h
  285    846    7081 AST_utils.c
  59    274    2154 AST_utils.o
  50    400   28756 AST_utils.o
  866   2757   25873 Makefile
  270    725    5578 Makefile.am
  866   2743   27320 Makefile.in
  38    175    1154 alloca.c
 2035   4516   47721 aloctypes.c
  86    350    3286 aloctypes.h
  104   1051   66848 aloctypes.o
...

```

Climate Data for IAD in August, 2005

```
=====
 1  2  3  4  5  6A 6B  7  8  9  10 11 12 13 14 15 16 17 18
                                AVG MX 2MIN
DY MAX MIN AVG DEP HDD CDD  WTR  SNW DPTH SPD SPD DIR MIN PSBL S-S WX  SPD DR
=====
 1  87  66  77  1  0  12 0.00 0.0  0  2.5  9 200  M  M  7 18  12 210
 2  92  67  80  4  0  15 0.00 0.0  0  3.5 10 10  M  M  3 18  17 320
 3  93  69  81  5  0  16 0.00 0.0  0  4.1 13 360  M  M  2 18  17 360
 4  95  69  82  6  0  17 0.00 0.0  0  3.6  9 310  M  M  3 18  12 290
 5  94  73  84  8  0  19 0.00 0.0  0  5.9 18 10  M  M  3 18  25 360
 6  89  70  80  4  0  15 0.02 0.0  0  5.3 20 200  M  M  6 138 23 210
 7  89  69  79  3  0  14 0.00 0.0  0  3.6 14 200  M  M  7 1  16 210
 8  86  70  78  3  0  13 0.74 0.0  0  4.4 17 150  M  M 10 18 23 150
 9  76  70  73 -2  0  8 0.19 0.0  0  4.1  9  90  M  M  9 18 13  90
10  87  71  79  4  0  14 0.00 0.0  0  2.3  8 260  M  M  8 1  10 210
...

```

Raw Census 2000 Data for DC

```
u108_s,DC
/
000,01,0000001,572059,72264,572059,12.6,572059,572059,572059,0,0,0,0,5
72059,175306,343213,2006,14762,383,21728,14661,572059,527044,158617,34
0061,1560,14605,291,1638,10272,45015,16689,3152,446,157,92,20090,4389,
572059,268827,3362,3048,3170,3241,3504,3286,3270,3475,3939,3647,3525,3
044,2928,2913,2769,2752,2933,2703,4056,5501,5217,4969,13555,24995,2421
6,23726,20721,18802,16523,12318,4345,5810,3423,4690,7105,5739,3260,234
7,303232,3329,3057,2935,3429,3326,3456,3257,3754,3192,3523,3336,3276,2
989,2838,2824,2624,2807,2871,4941,6588,5625,5563,17177,27475,24377,228
18,21319,20851,19117,15260,5066,6708,4257,6117,10741,9427,6807,6175,57
2059,536373,370675,115963,55603,60360,57949,129440,122518,3754,3168,22
448,9967,4638,14110,16160,165698,61049,47694,13355,71578,60875,10703,3
3071,35686,7573,28113,248590,108569,47694,60875,140021,115963,58050,21
654,36396,57913,10355,4065,6290,47558,25229,22329,24058,13355,10703,70
088,65737,37112,21742,12267,9475,9723,2573,2314,760,28625,8207,7469,73
8,19185,18172,1013,1233,4351,3610,741,248590,199456,94221,46274,21443,
24831,47947,8705,3979,4726,39242,25175,14067,105235,82928,22307,49134,
21742,11776,211,11565,9966,1650,86,1564,8316,54,8262,27392,25641,1751,
248590,115963,4999,22466,26165,24062,16529,12409,7594,1739,132627,1167
0,32445,23225,21661,16234,12795,10563,4034,248590,115963,48738,28914,1
9259,10312,4748,3992,132627,108569,19284,2713,1209,509,218,125

```

A Simple Example

- Let's start with a simple Ruby program

```
ruby1.rb: # This is a ruby program
          x = 37
          y = x + 5
          print(y)
          print("\n")

```

```
% ruby -w ruby1.rb
42
%
```

Language Basics

comments begin with #, go to end of line

variables need not be declared

no special main() function or method

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

line break separates expressions (can also use ";" to be safe)

Run Ruby, Run

- There are three ways to run a Ruby program
 - `ruby -w filename` – execute script in *filename*
 - tip: the `-w` will cause Ruby to print a bit more if something bad happens
 - `irb` – launch interactive Ruby shell
 - can type in Ruby programs one line at a time, and watch as each line is executed

```
irb(main):001:0> 3+4
=> 7
irb(main):002:0> print("hello\n")
hello
=> nil
```

Run Ruby, Run (cont'd)

- Suppose you want to run a Ruby script as if it were an executable

```
#!/usr/local/bin/ruby -w
print("Hello, world!\n")
```

- `./filename # run program`
 - The first line (“shebang”) tells the system where to find the program to interpret this text file
 - Must `chmod u+x filename` first
 - Or `chmod a+x filename` so everyone has exec permission
 - Warning: Not very portable
 - Depends on location `/usr/local/bin/ruby`

Explicit vs. Implicit Declarations

- Java and C/C++ use **explicit variable declarations**
 - variables are named and typed before they are used
 - `int x, y; x = 37; y = x + 5;`
- In Ruby, variables are **implicitly declared**
 - first use of a variable declares it and determines type
 - `x = 37; y = x + 5;`
 - `x, y` exist, will be integers

Tradeoffs?

Explicit Declarations

Higher overhead

Helps prevent typos

Forces programmer to document types

Implicit Declarations

Lower overhead

Easy to mistype variable name

Figures out types of variables automatically

Methods in Ruby

Methods are declared with `def...end`

List parameters at definition

May omit parens on call

Invoke method

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end

x = sayN("hello", 3)
puts(x)
```

(Methods must begin with lowercase letter and be defined before they are called)

Method (and Function) Terminology

- Formal parameters
 - The parameters used in the body of the method
 - `message, n` in our example
- Actual parameters
 - The arguments passed in to the method at a call
 - `"hello", 3` in our example
 - Note: Some people make this distinction with “parameters” versus “arguments,” but I can never remember which is which...

More Control Statements in Ruby

- A *control statement* is one that affects which instruction is executed next
 - We’ve seen two so far in Ruby
 - `while` and function call
- Ruby also has conditionals

```
if grade >= 90 then
  puts "You got an A"
elsif grade >= 80 then
  puts "You got a B"
elsif grade >= 70 then
  puts "You got a C"
else
  puts "You're not doing so well"
end
```

What is True?

- The *guard* of a conditional is the expression that determines which branch is taken

```
if grade >= 90 then
  ...
```

Guard

- The *true* branch is taken if the guard evaluates to anything except
 - false
 - nil
- **Warning** to C programmers: `0` is *not* false!

Yet More Control Statements in Ruby

- `unless cond then stmt-f else stmt-t end`
 - Same as “`if not cond then stmt-t else stmt-f end`”

```
unless grade < 90 then
  puts "You got an A"
else unless grade < 80 then
  puts "You got a B"
end
end
```

- `until cond body end`
 - Same as “`while not cond body end`”

```
until i >= n
  puts message
  i = i + 1
end
```

Even More Control Statements in Ruby

- Can write `if` and `unless` after an expression
 - `puts "You got an A" if grade >= 90`
 - `puts "You got an A" unless grade < 90`
- Case is a multi-way branch

```
case grade
when 90..100
  puts "You got an A"
when 80..89
  puts "You got a B"
when 70..79
  puts "You got a C"
else
  puts "You failed"
end
```

Why So Many Conditionals?

- Is this a good idea?
- Advantages? Disadvantages?

Looping with while

- Basic loop construct is `while..end`

```
i = 0
while i < 5
  puts i.to_s
  i = i + 1
end
```

- Inside of while
 - `break` exits the while loop
 - `next` jumps to the next iteration of the loop
 - `redo` “restarts” the current iteration
 - I.e., jumps back to the top of the loop

Other Looping Constructs

- Ruby also has “for”
 - Though it’s just syntactic sugar, as we’ll see later

```
for elt in [1, "math", 3.4]
  puts elt.to_s
end
```

```
for i in 1..3
  puts i
end
```

Classes and Objects

- Class names begin with an uppercase letter
- The “new” method creates an object
 - `s = String.new` creates a new `String` and makes `s` refer to it
- Every class inherits from `Object`

Everything is an Object

- In Ruby, **everything** is in fact an object
 - `(-4).abs`
 - integers are instances of `Fixnum`
 - `3 + 4`
 - infix notation for “invoke the `+` method of `3` on argument `4`”
 - `"programming".length`
 - strings are instances of `String`
 - `String.new`
 - classes are objects with a `new` method
 - `(4.13).class`
 - use the `class` method to get the class for an object
 - floating point numbers are instances of `Float`

Objects and Classes

- Objects are data
- Classes are types (the kind of data which things are)
- But in Ruby, classes themselves are objects!

Object	Class
10	Fixnum
-3.30	Float
"CMSC 330"	String
String.new	String
Fixnum	Class
String	Class

- Fixnum, Float, String, etc., (including Class), are objects of type Class

Two Cool Things to Do with Classes

- Since classes are objects, you can manipulate them however you like
 - if p then x = String else x = Time end # Time is # another class
 - ... # creates a String or a Time, # depending upon p
 - y = x.new
- You can get names of all the methods of a class
 - Object.methods
 - => ["send", "name", "class_eval", "object_id", "new", "autoload?", "singleton_methods", ...]

The nil Object

- Ruby uses a special object **nil**
 - All uninitialized fields set to **nil** (@ refers to a class field)

```
irb(main):004:0> @x
=> nil
```
 - Like **NULL** or **0** in C/C++ and **null** in Java
- **nil** is an object of class **NilClass**
 - It's a *singleton object* – there is only one instance of it
 - **NilClass** does *not* have a new method
 - **nil** has methods like **to_s**, but not other methods that don't make sense

```
irb(main):006:0> @x + 2
NoMethodError: undefined method '+' for nil:NilClass
```

What is a Program?

- In C/C++, a program is...
 - A collection of declarations and definitions
 - With a distinguished function definition
 - `int main(int argc, char *argv[]) { ... }`
 - When you run a C/C++ program, it's like the OS calls **main(...)**
- In Java, a program is...
 - A collection of class definitions
 - With a class **Cl** that contains a method
 - `public static void main(String[] args)`
 - When you run `java Cl`, the **main** method of class **Cl** is invoked

A Ruby Program is...

- The class `Object`
 - When the class is loaded, any expressions not in method bodies are executed

defines a method of `Object`

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end
```

invokes `self.sayN`

invokes `self.puts`
(part of `Object`)

```
x = sayN("hello", 3)
puts(x)
```

Ruby is Dynamically Typed

- Recall we don't declare types of variables
 - But Ruby does keep track of types at run time
- ```
x = 3; x.foo
NoMethodError: undefined method 'foo' for 3:Fixnum
```
- We say that Ruby is **dynamically typed**
    - Types are determined and checked at run time
  - Compare to C, which is **statically typed**

```
Ruby
x = 3
x = "foo" # gives x a
 # new type
```

```
/* C */
int x;
x = 3;
x = "foo"; /* not allowed */
```

## Types in Java and C++

- Are Java and C++ statically or dynamically typed?
    - A little of both
    - Many things are checked statically
- ```
Object x = new Object();
x.println("hello"); // No such method error at compile time
```
- But other things are checked dynamically
- ```
Object o = new Object();
String s = (String) o; // No compiler warning, fails at run time
// (Some Java compilers may be smart enough to warn about
// above cast)
```

## Tradeoffs?

### Static types

More work to do when writing code

Helps prevent some subtle errors

Fewer programs type check

### Dynamic types

Less work when writing code

Can use objects incorrectly and not realize until execution

More programs type check

## Classes and Objects in Ruby

```
class Point
 def initialize(x, y)
 @x = x
 @y = y
 end

 def addX(x)
 @x += x
 end

 def to_s
 return "(" + @x.to_s + "," + @y.to_s + ")"
 end
end

p = Point.new(3, 4)
p.addX(4)
puts(p.to_s)
```

class contains method/  
constructor definitions

constructor definition

instance variables prefixed with "@"

method with no arguments

instantiation

invoking no-arg method

CMSC 330

33

## Classes and Objects in Ruby (cont'd)

- Recall classes begin with an uppercase letter
- `inspect` converts *any* instance to a string

```
irb(main):033:0> p.inspect
=> "#<Point:0x54574 @y=4, @x=7>"
```
- Instance variables are prefixed with `@`
  - Compare to local variables with no prefix
  - Cannot be accessed outside of class*
- The `to_s` method can be invoked implicitly
  - Could have written `puts(p)`
    - Like Java's `toString()` methods

CMSC 330

34

## Inheritance

- Recall that every class inherits from `Object`

```
class A
 def add(x)
 return x + 1
 end
end

class B < A
 def add(y)
 return (super(y) + 1)
 end
end

b = B.new
puts(b.add(3))
```

extend superclass

invoke add method  
of parent

CMSC 330

35

## Global Variables in Ruby

- Ruby has two kinds of global variables
  - Class variables beginning with `@@`
  - Global variables across classes beginning with `$`

```
class Global
 @@x = 0

 def Global.inc
 @@x = @@x + 1; $x = $x + 1
 end

 def Global.get
 return @@x
 end
end

$x = 0
Global.inc
$x = $x + 1
Global.inc
puts(Global.get)
puts($x)
```

define a class  
("singleton") method

CMSC 330

36

## Special Global Variables

- Ruby has a bunch of global variables that are implicitly set by methods
- The most insidious one: `$_`
  - Default method return, argument in many cases
- Example:

```
gets # implicitly reads input into $_
print # implicitly writes $_
```
- Using `$_` leads to shorter programs
  - And confusion
  - It's suggested you avoid using it

## Creating Strings in Ruby

- Substitution in double-quoted strings with `#{}` 
  - `course = "330"; msg = "Welcome to #{course}"`
  - `"It is now #{Time.new}"`
  - The contents of `#{}`  may be an arbitrary expression
  - Can also use single-quote as delimiter
    - No expression substitution, fewer escaping characters
- Here-documents

```
s = <<END
This is a long text message
on multiple lines
and typing \n is annoying
```

## Creating Strings in Ruby (cont'd)

- Ruby also has `printf` and `sprintf`
  - `printf("Hello, %s\n", name);`
  - `sprintf("%d: %s", count, Time.now)`
    - Returns a string
- The `to_s` method returns a `String` representation of a class object

## Standard Library: String

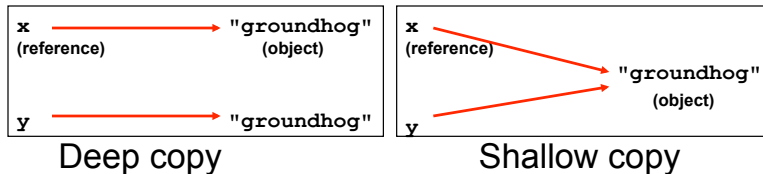
- The `String` class has many useful methods
  - `s.length` # length of string
  - `s1 == s2` # “deep” equality (string contents)
  - `s = "A line\n"; s.chomp` # returns "A line"
    - Return new string with `s`'s contents except newline at end of line removed
  - `s = "A line\n"; s.chomp!`
    - Destructively removes newline from `s`
    - *Convention:* methods ending in `!` modify the object
    - *Another convention:* methods ending in `?` observe the object
  - `"r1\t r2\t\t r4".each("\t") { |rec| puts rec }`
    - Apply code block to each tab-separated substring

## Digression: Deep vs. Shallow Copy

- Consider the following code
  - Assume an object/reference model like Java or Ruby
    - (Or even two pointers pointing to the same structure)

```
x = "groundhog" ; y = x
```

- Which of these occurs?



## Deep vs. Shallow Copy (cont'd)

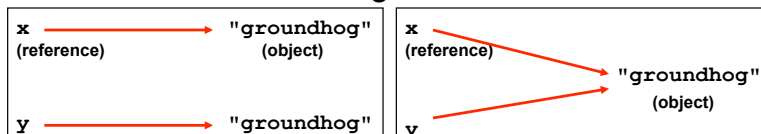
- Ruby and Java would both do a shallow copy in this case
- But this Ruby example would cause deep copy:

```
x = "groundhog"
y = String.new(x)
```

- Note: In Java, `new String(x)` is probably not useful; in Ruby, `String.new` might be. Why?

## Deep vs. Shallow Equality

- Consider these cases again:



- If we compare `x` and `y`, what is compared?
  - The references, or the contents of the objects they point to?
- If references are compared the first would return false but the second true
- If objects are compared both would return true

## String Equality

- In Java, `x == y` is shallow equality, always
  - Compares references, not string contents
- In Ruby, `x == y` for strings uses deep equality
  - Compares contents, not references
  - `==` is a method that can be overridden in Ruby!
  - To check shallow equality, use the `equal?` method
    - Inherited from the `Object` class
- It's always important to know whether you're doing a deep or shallow copy
  - And deep or shallow comparison

## Standard Library: String (cont'd)

- `"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\t r2\t r3".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
  - We'll see that soon

## 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`
  - But you won't often use its methods
- 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'd)

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

## Repetition in Regular Expressions

- `/(Ruby)*/`
  - { "", "Ruby", "RubyRuby", "RubyRubyRuby", ... }
  - \* means *zero or more occurrences*
- `/Ruby+/`
  - { "Ruby", "Rubyy", "Rubyyyy", ... }
  - + means *one or more occurrence*
  - so `/e+/` is the same as `/ee*/`
- `/(Ruby)?/`
  - { "", "Ruby" }
  - ? means *optional*, i.e., zero or one occurrence

## More Repetition

- `/(Ruby){3}/`
  - { "RubyRubyRuby" }
  - {n} means exactly n occurrences
- `/(Ruby){3,}/`
  - { "RubyRubyRuby", "RubyRubyRubyRuby", ... }
  - {n,} means n or more occurrences
- `/(Ruby){2,4}/`
  - { "RubyRuby", "RubyRubyRuby", "RubyRubyRubyRuby" }
  - {n,m} means at least n through at most m occurrences
- Do these add any new power to regexps?

## 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
- `/[\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]   |
| \\w  | word character, [A-Za-z0-9_] |
| \\D  | non-digit, [^0-9]            |
| \\S  | non-space, [^\\t\\r\\n\\f]   |
| \\W  | non-word, [^A-Za-z0-9_]      |

## Potential Character Class Confusions

---

- `^`
  - Inside char classes: not
  - Outside char classes: beginning of line
- `[]`
  - Inside regexps: character class
  - Outside regexps: Ruby Array
- `()`
  - Inside char classes: literal characters ( and )
    - Note `/(0..2)/` does not mean {0, 1, 2}
  - Outside char classes: used for grouping
- `-`
  - Inside char classes: range
  - Outside char classes: subtraction

## Regular Expression Practice

---

- 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?)$/`

## 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|-)(x|-)(w|-)(x|-)(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

- Ruby remembers which strings matched the parenthesized parts of r.e.'s
- These parts can be referred to using special variables called **backreferences** (named \$1, \$2,...)
- Examples:

```
- /^Status: (.*)/
```

- Capture all chars to the right on lines beginning with "Status"

```
- /^Min: (\d+) Max: (\d+)/
```

- Capture digits following "Min" and "Max"

## Backreference Example

- Extract information from a report

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

```
"hello" =~ /(h)e(l)llo/
puts $1 # h
puts $2 # ll
"hello" =~ /h(e)llo/
puts $1 # e
puts $2 # nil
"hello" =~ /hello/
puts $1 # nil
```

## Standard Library: Array

- Arrays of objects are instances of class **Array**
  - Arrays may be heterogeneous  
`a = [1, "foo", 2.14]`
  - C-like syntax for accessing elements, indexed from 0  
`x = a[0]; a[1] = 37`
- Arrays are **growable**
  - Increase in size automatically as you access elements  
`irb(main):001:0> b = []; b[0] = 0; b[5] = 0; puts b.inspect`  
`[0, nil, nil, nil, nil, 0]`
  - `[]` is the empty array, same as `Array.new`

## Standard Library: Arrays (cont'd)

- Arrays can also shrink
  - Contents shift left when you delete elements  
`a = [1, 2, 3, 4, 5]`  
`a.delete_at(3)` # delete at position 3; `a = [1,2,3,5]`  
`a.delete(2)` # delete element = 2; `a = [1,3,5]`
- Can use arrays to model stacks and queues  
`a = [1, 2, 3]`  
`a.push("a")` # `a = [1, 2, 3, "a"]`  
`x = a.pop` # `x = "a"`  
`a.unshift("b")` # `a = ["b", 1, 2, 3]`  
`y = a.shift` # `y = "b"`

**Note:** push, pop, shift, and unshift all permanently modify the array

## Iterating through Arrays

- It's easy to iterate over an array with **while**

```
a = [1,2,3,4,5]
i = 0
while i < a.length
 puts a[i]
 i = i + 1
end
```

- Looping through all elements of an array is very common
  - And there's a better way to do it in Ruby

## Iteration and Code Blocks

- The `Array` class also has an `each` method, which takes a code block as an argument

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

code block delimited by  
{}'s or do...end

parameter name

body

## More Examples of Code Blocks

- Sum up the elements of an array

```
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

- Print out each segment of the string as divided up by commas
  - Can use any delimiter

```
s = "Student,Sally,099112233,A"
s.each(',') { |x| puts x }
```

("delimiter" = symbol used to denote boundaries)

## Yet More Examples of Code Blocks

```
3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }
```

- `n.times` runs code block `n` times
- `n.upto(m)` runs code block for integers `n..m`
- `a.find` returns first element `x` of array such that the block returns true for `x`
- `a.collect` applies block to each element of array and returns new array

## Still Another Example of Code Blocks

```
File.open("test.txt", "r") do |f|
 f.readlines.each { |line| puts line }
end
```

- `open` method takes code block with file argument
  - File automatically closed after block executed
- `readlines` reads all lines from a file and returns an array of the lines read
  - Use `each` to iterate

## Using Yield To Call Code Blocks

- Any method can be called with a code block
  - Inside the method, the block is called with `yield`
- After the code block completes
  - Control returns to the caller after the `yield` instruction

```
def countx(x)
 for i in 1..x
 puts i
 yield
 end
end

countx(4) { puts "foo" }
```

```
1
foo
2
foo
3
foo
4
foo
```

CMSC 330

69

## So What are Code Blocks?

- A code block is just a special kind of method
  - `{ |y| x = y + 1; puts x }` is almost the same as
    - `def m(y) x = y + 1; puts x end`
- The `each` method takes a code block as an argument
  - This is called *higher-order programming*
    - In other words, methods take other methods as arguments
    - We'll see a lot more of this in OCaml
- We'll see other library classes with `each` methods
  - And other methods that take code blocks as arguments
  - Your own methods can also take code block args

CMSC 330

70

## Code blocks and the scan Method

- `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
- (There are also some other ways to call scan)

CMSC 330

71

## Example of Using scan

```
12 34 23
20 77 87
13 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

CMSC 330

72

## Standard Library: Hash

---

- A hash acts like an associative array
  - Elements can be indexed by any kind of values
  - Every Ruby object can be used as a hash key, because the `Object` class has a `hash` method
- Elements are referred to using `[]` like array elements, but `Hash.new` is the `Hash` constructor

```
italy["population"] = 58103033
italy["continent"] = "europe"
italy[1861] = "independence"
```

## Hash (cont'd)

---

- The `Hash` method `values` returns an array of a hash's values (in some order)
- And `keys` returns an array of a hash's keys (in some order)
- Iterating over a hash:

```
italy.keys.each {
 |key| puts("key: #{key}, value: #{italy[key]}")
}
```

## Hash (cont'd)

---

### Convenient syntax for creating literal hashes

- Use `{ key => value, ... }` to create hash table

```
credits = {
 "cmsc131" => 4,
 "cmsc330" => 3,
}

x = credits["cmsc330"] # x now 3
credits["cmsc311"] = 3
```

## 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

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

Class of exception  
to catch

Local name  
for exception

## Command Line Arguments

- Stored in predefined array variable `$*`
  - Can refer to as 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)

```
gcggcattcagcaccocgtatactgttaagcaatccagatTTTTgtgtataacataccggc
catactgaagcattcattgaggctagcgctgataacagtagcgctaacaatgggggaatg
tggaatacgggtcgactactaagagccgggaccacacaccccgtaaggatggagcgtgg
taacataataatccggttcaagcagtgggcgaaggtggagatgttccagtaagaatagtg
gggctactaccatggtacataattaagagatcgtaactcttgagacgggtcaatgggtac
cgagactatatcaactcggacgctatgcgcttactggtcacctcgttactgacgga
```

## Practice: Amino Acid counting in DNA

```
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

## Considering Ruby Again

---

- Interpreted
- Implicit declarations
- Dynamically typed
  - These three make it quick to write small programs
- Built-in regular expressions and easy string manipulation
  - This and the three above are the hallmark of scripting languages
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...

## 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 one you like best