CMSC 330: Organization of Programming Languages

Ruby

Reminders and Announcements

• If you’re not on the list, you’re not in the class (I have the list)
• Project 1 was posted on Sep. 3
  – It is due on Sep. 24
  – Start immediately
• Check glue access
• Use the class forum
• Read complete syllabus online
• Leave 24 hours for email responses
Review

- Why study programming languages?
- What makes a good programming language?
- Compilers vs. Interpreters
- What kind of language is...
  - C
  - Java
  - Ruby
  - OCaml

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")
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.h
  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

<table>
<thead>
<tr>
<th>AVG</th>
<th>MX</th>
<th>2MIN</th>
<th>DY</th>
<th>MAX</th>
<th>MIN</th>
<th>AVG</th>
<th>DEP</th>
<th>HDD</th>
<th>CDD</th>
<th>WTR</th>
<th>SNW</th>
<th>DPTH</th>
<th>SPD</th>
<th>SPD</th>
<th>DIR</th>
<th>MIN</th>
<th>PSBL</th>
<th>S-S</th>
<th>WX</th>
<th>SFD</th>
<th>DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>87</td>
<td>66</td>
<td>77</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>2.5</td>
<td>9</td>
<td>200</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>12</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>92</td>
<td>67</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>3.5</td>
<td>10</td>
<td>10</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>12</td>
<td>320</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>93</td>
<td>69</td>
<td>81</td>
<td>5</td>
<td>0</td>
<td>16</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>4.1</td>
<td>13</td>
<td>360</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>17</td>
<td>360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>95</td>
<td>69</td>
<td>82</td>
<td>6</td>
<td>0</td>
<td>17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>3.6</td>
<td>9</td>
<td>310</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>12</td>
<td>290</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>94</td>
<td>73</td>
<td>84</td>
<td>8</td>
<td>0</td>
<td>19</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>5.9</td>
<td>18</td>
<td>10</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>23</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>89</td>
<td>70</td>
<td>80</td>
<td>4</td>
<td>0</td>
<td>15</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>5.3</td>
<td>20</td>
<td>200</td>
<td>M</td>
<td>M</td>
<td>6</td>
<td>138</td>
<td>23</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>89</td>
<td>69</td>
<td>79</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>3.6</td>
<td>14</td>
<td>200</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>1</td>
<td>16</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>86</td>
<td>70</td>
<td>78</td>
<td>3</td>
<td>0</td>
<td>13</td>
<td>0.74</td>
<td>0.00</td>
<td>0.00</td>
<td>4.4</td>
<td>17</td>
<td>150</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>23</td>
<td>150</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>76</td>
<td>70</td>
<td>73</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td>0.19</td>
<td>0.00</td>
<td>0.00</td>
<td>4.1</td>
<td>9</td>
<td>90</td>
<td>M</td>
<td>M</td>
<td>9</td>
<td>18</td>
<td>13</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>87</td>
<td>71</td>
<td>79</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>2.3</td>
<td>8</td>
<td>260</td>
<td>M</td>
<td>M</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Raw Census 2000 Data for DC

| u108_S,DC,000,01,0000001|572059,72264,572059,12.6,572059,572059,572059,0,0,0,0,572059,175306,343213,2006,14762,383,21728,14661,572059,527044,159861,7,340061,1560,14605,291,1638,10272,45015,16689,3152,446,157,92,20090,43,89,572059,268827,3362,3048,3170,3241,3504,3286,3270,3475,3939,3647,3525,3044,2928,2913,2769,2752,2933,2703,4056,5501,5217,4969,13555,24995,2422,16,23726,20721,18802,16521,12318,4345,5810,3423,4690,7105,5739,3260,2347,303232,3329,3057,2935,3429,3326,3456,3257,3754,3192,3523,3336,3276,29,89,2838,2824,2624,2807,2871,4941,4588,5626,5563,17777,27475,24377,22818,213139,20851,19117,15260,5066,6708,4257,6117,10741,9427,6807,6175,57205,9,536737,370675,115963,55603,60360,57949,129440,112518,3754,3168,22448,9967,4638,14110,16160,165698,61049,47694,13555,71578,60875,10703,33071,35684,7573,28123,248590,108569,47694,60875,140021,113963,58050,21654,36,396,57913,10355,4065,6290,47558,25229,22329,24058,13355,10703,70088,65737,371122,21742,12267,9475,9723,2573,2314,760,28625,8207,7469,738,19185,18172,1013,1233,4351,3610,741,248590,199456,94221,46274,21443,24831,479,47,8705,3979,4726,39242,25175,14067,105235,82928,22307,49134,21742,1177,6,211,11565,9966,1650,86,1564,8316,54,8262,27392,25641,3751,248590,1159,63,9999,22466,26165,24062,16529,12609,7594,1739,132627,11670,32445,2322,5,21661,16234,12795,10563,4034,248590,115963,48738,28914,19259,10312,47,48,3992,132627,108569,19284,2713,1209,509,218,125,48,3992,132627,108569,19284,2713,1209,509,218,125 |
A Simple Example

• Let’s start with a simple Ruby program

```
# 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
- 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
      ```ruby
      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
  ```ruby
  #!/usr/local/bin/ruby -w
  print("Hello, world!\n")
  ```
  - `./filename` # run program
    - The first line 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?

<table>
<thead>
<tr>
<th>Explicit Declarations</th>
<th>Implicit Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overhead?</strong></td>
<td>Overhead?</td>
</tr>
<tr>
<td>Helps prevent typos</td>
<td>Easy to mistype variable name</td>
</tr>
<tr>
<td>Forces programmer to document types</td>
<td>Figures out types of variables automatically</td>
</tr>
</tbody>
</table>
Methods in Ruby

Methods are declared with def...end

```
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)

List parameters at definition

May omit parens on call

Invoke method

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
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

```ruby
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

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

- 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 \textit{cond} then \textit{stmt-f} else \textit{stmt-t} end
  - Same as “if not \textit{cond} then \textit{stmt-t} else \textit{stmt-f} end”

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

- until \textit{cond body} end
  - Same as “while not \textit{cond body} end”

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

Using If and Unless as Modifiers

- Can write \textit{if} and \textit{unless after} an expression
  - puts "You got an A" if grade >= 90
  - puts "You got an A" unless grade < 90

- Why so many control statements?
  - Is this a good idea?
  - Advantages? Disadvantages?
Other useful control statements

```ruby
Other useful control statements
```

```ruby
case x
  when 1, 3..5
  puts elt
end
```

```ruby
while i>n
  break
  next
  puts message
  redo
end
```

```ruby
for i in (1..3)
  puts i
end
```

```ruby
(1..3).each {
  |elt|
  puts elt
}
```

```ruby
IO.foreach(filename)
  {
    |x|
    puts x
  }
```

```ruby
generates a string. Also see to_i
```

To try with a neighbor

Write (on paper) a Ruby function to print all even numbers from 1 to some given value x.

```ruby
def even(x)
  for i in (1..x)
    if i % 2 == 0
      puts i
    end
  end
end
```

```ruby
def even(x)
  (1..x).each{
    |i|
    if i % 2 == 0
      puts i
    end
  }
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` is an example of a method call on an object.
  - `3 + 4` is an example of an expression involving objects.
  - `"programming".length` is an example of a method call on a string object.
  - `String.new` is a constructor method for creating a new `String` object.
  - `(4.13).class` is an example of using the `class` method to get the class for an object.
  - Floating point numbers are instances of `Float`.

Writing `elt` as `#{elt}` makes it clear that it is a variable to be evaluated, not a literal word to be printed. This is a cleaner way to express output; it builds a single string and presents it as a single argument to `puts`.

```
ruby> for elt in [100,-9.6,"pickle"]  
  |   puts "#{elt}\t#{elt.class}" 
  | end 
  | 100  (Fixnum) 
  | -9.6  (Float)  
  |  pickle (String) 
... 
```
Objects and Classes

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

<table>
<thead>
<tr>
<th>Object</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Fixnum</td>
</tr>
<tr>
<td>-3.30</td>
<td>Float</td>
</tr>
<tr>
<td>&quot;CMSC 330&quot;</td>
<td>String</td>
</tr>
<tr>
<td>String.new</td>
<td>String</td>
</tr>
<tr>
<td>Fixnum</td>
<td>Class</td>
</tr>
<tr>
<td>String</td>
<td>Class</td>
</tr>
</tbody>
</table>

- 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
  - ...  
  - y = x.new  # creates a String or a Time, depending upon p

- 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

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

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
    ```ruby
    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
# 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
    
    ```java
    Object x = new Object();
    x.println("hello"); // No such method error at compile time
    ```
  - But other things are checked dynamically
    
    ```java
    Object o = new Object();
    String s = (String) o; // No compiler warning, fails at runtime
    // (Some Java compilers may be smart enough to warn about above cast)
    ```

Tradeoffs?

<table>
<thead>
<tr>
<th>Static types</th>
<th>Dynamic types</th>
</tr>
</thead>
<tbody>
<tr>
<td>More work to do when writing code</td>
<td>Less work when writing code</td>
</tr>
<tr>
<td>Helps prevent some subtle errors</td>
<td>Can use objects incorrectly and not realize until execution</td>
</tr>
<tr>
<td>Fewer programs type check</td>
<td>More programs type check</td>
</tr>
</tbody>
</table>
Classes and Objects in Ruby

```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)
```

Classes and Objects in Ruby (cont’d)

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

- Recall that every class inherits from Object

```ruby
class A
  def plusplus(x)
    return x + 1
  end
end

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

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

Global Variables in Ruby

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

```ruby
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)
```
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:
  ```ruby
  gets  # implicitly reads input into $
  print  # implicitly writes $
  ```
- Using $_ leads to shorter programs
  - but 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 to create strings ‘\hi’
    - No expression substitution, fewer escaping characters
- Here-documents
  ```ruby
  s = <<END
  This is a long text message on multiple lines and typing \n is annoying
  END
  Can be any text
  ```
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\tr2\tr4".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 copy | Shallow copy
---|---
\[x\] (reference) | \["groundhog"\] (object)
\[y\] (reference) | \["groundhog"\] (object)

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:

\[
x = "groundhog"
y = String.new(x)
\]

- In Java, this is done by implementing the cloneable interface and calling clone()
Deep vs. Shallow Equality

- Consider these cases again:

  ![Diagram showing references and objects](image)

  - 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 \texttt{equal?} method
    - Inherited from the \texttt{Object} class

- It’s always important to know whether you’re doing a deep or shallow copy
  - And deep or shallow comparison