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 *
  271 274 5323 AST.c
  270 273 5323 AST.h
  127 1455 20978 AST.o
  1476 5436 64761 ASTdefs.c
  1370 6247 65657 ASTdefs.h
  271 844 9483 ASTparent.c
  410 7270 87963 ASTprint.c
  265 946 7040 ASTutils.c
  59 274 2154 ASTutils.h
  50 440 28754 ASTutils.o
  866 2757 25873 Makefile
  270 725 1078 Makefile.am
  270 725 1078 Makefile.in
  38 275 1234 alloca.c
  2035 4516 47721 aloctypes.c
  86 350 3286 aloctypes.h
  104 1051 6384 alloctypes.h
  104 1051 6384 alloctypes.o
```

Do away or I will replace you with a very small shell script.
Climate Data for IAD in August, 2005

| 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   | 2    | 0.00 | 0   | 0   | 2.5  | 9   | 200     | M    | M    | 7      | 18    |
| 2   | 92      | 67     | 80      | 4    | 0   | 15   | 0.00 | 0   | 0   | 3.5  | 10  | 10      | M    | M    | 3      | 18    |
| 3   | 93      | 69     | 81      | 5    | 0   | 16   | 0.00 | 0   | 0   | 4.1  | 13  | 360     | M    | M    | 2      | 18    |
| 4   | 95      | 69     | 82      | 6    | 0   | 17   | 0.00 | 0   | 0   | 3.6  | 9   | 310     | M    | M    | 3      | 18    |
| 5   | 94      | 73     | 84      | 8    | 0   | 19   | 0.00 | 0   | 0   | 5.9  | 18  | 10      | M    | M    | 3      | 18    |
| 6   | 89      | 70     | 80      | 4    | 0   | 15   | 0.02 | 0   | 0   | 5.3  | 20  | 200     | M    | M    | 6      | 138   |
| 7   | 89      | 69     | 79      | 3    | 0   | 14   | 0.00 | 0   | 0   | 3.6  | 14  | 200     | M    | M    | 7      | 1     |
| 8   | 86      | 70     | 78      | 3    | 0   | 13   | 0.74 | 0   | 0   | 4.4  | 17  | 150     | M    | M    | 10     | 18    |
| 9   | 76      | 70     | 73      | -2   | 0   | 8    | 0.19 | 0   | 0   | 4.1  | 9   | 90      | M    | M    | 9      | 18    |
| 10  | 87      | 71     | 79      | 4    | 0   | 14   | 0.00 | 0   | 0   | 2.3  | 6   | 210     | M    | M    | 8      | 10    |

...
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>Overhead?</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
```

Methods are invoked with `puts sayN("hello", 3)`

- List parameters at definition
- May omit parens on call
- Invoke method

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` is the 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-f else stmt-t end"

- `until cond body end`  
  - Same as "while not cond body end"

Using If and Unless as Modifiers

- Can write `if` and `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

- `case x when 1, 3..5 end`
- `when 2, 6..8 end`

To try with a neighbor

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

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

```
def even(x)
  (1..x).each{|i|
    if i % 2 == 0
      puts i
    end
  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`
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
    => 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
```

- A class 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
# Ruby
x = 3
x = "foo"  # gives x a new type
x = "foo";  # not allowed in C
```
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, @y)"
  end
end

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

Inheritance

```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
b.plusplus(3)
```

Global Variables in Ruby

```ruby
class Global
  @@x = 0
  def Global.inc
    @@x = @@x + 1
  end
  def Global.get
    return @@x
  end
end

def Global.init
  $x = 0
  Global.inc
  $x = $x + 1
end

def Global.set
  return $x
end

$y = 4
Global.set
puts(Global.get)
puts($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.now}"  
  - The contents of \#{} may be an arbitrary expression  
  - Can also use single-quote to quote strings  
  - \"hi\"
  - No expression substitution, fewer escaping characters

- Here-documents
  - \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\t\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)

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

- Which of these occurs?

```ruby
x  # reference
y  # reference
```

- Deep copy

```ruby
x  # "groundhog" (object)
y  # "groundhog" (object)
```

- Shallow copy

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:

```ruby
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:
- 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