CMSC 330:
Organization of Programming Languages

Introduction to Ruby:
Ruby

- An object-oriented, imperative, dynamically typed (scripting) language
  - Similar to other scripting languages (e.g., Python)
  - Notable in being **fully object-oriented**, and embracing **higher-order programming** style
    - Functions taking function(al code) as arguments

- Created in 1993 by Yukihiro Matsumoto (Matz)
  - “Ruby is designed to make programmers happy”

- Adopted by **Ruby on Rails** web programming framework in 2005 (a key to Ruby’s popularity)
Books on Ruby

- See course web page
Applications of Scripting Languages

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

- Motivating application

  Text processing
% wc *

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

...
Ruby is a **Scripting Dynamic Language**

- Ruby started with special purpose, but has grown into a *general-purpose* language
  - As have related languages, like Python and Perl

- But Ruby has distinctive features when compared to traditional general-purpose languages
  - Such as lightweight syntax, dynamic typing, evaluating code in strings, …

- We will call them *scripting languages*, still, but also *dynamic languages*
A Simple Example

Let’s start with a simple Ruby program

```ruby
# This is a ruby program
x = 1
n = 5
while n > 0
    x = x * n
    n = n - 1
end
print(x)
print(

% ruby -w ruby1.rb
120
%
```
Language Basics

- comments begin with #, go to end of line
- variables need not be declared
- no special main() function or method

```ruby
# This is a ruby program
x = 1
n = 5
while n > 0
  x = x * n
  n = n - 1
end
print(x)
print("\n")
```
Run Ruby, Run

There are two basic 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
  ➢ Ruby filenames should end with ‘.rb’ extension

• 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
  ➢ Can load Ruby programs via load command
    • E.g.: load ‘foo.rb’

Ruby is installed on Grace cluster
Some Ruby Language Features

- Implicit declarations
  - Java, C have explicit declarations

- Dynamic typing
  - Java, C have (mostly) static typing

- Everything is an object
  - No distinction between objects and primitive data
  - Even “null” is an object (called nil in Ruby), as are classes

- No outside access to private object state
  - Must use getters, setters

- No method overloading

- Class-based and Mixin inheritance
Implicit vs. Explicit Declarations

- In Ruby, variables are implicitly declared
  - First use of a variable declares it and determines type
    - `x = 37; // no declaration needed – created when assigned to`
    - `y = x + 5`
      - `x`, `y` now exist, are integers

- Java and C/C++ use explicit variable declarations
  - Variables are named and typed before they are used
    - `int x, y; // declaration`
    - `x = 37; // use`
    - `y = x + 5; // use`
Tradeoffs?

Explicit Declarations | Implicit Declarations
More text to type | Less text to type
Helps prevent typos | Easy to mistype variable name

```
var = 37
if (rare-condition)
y = vsr + 5
```

**Typo!**

Only caught when this line is actually run.
Bug could be latent for quite a while
Static Type Checking (Static Typing)

- **Before program is run**
  - Types of all expressions are determined
  - Disallowed operations cause compile-time error
    - Cannot run the program

- **Static types are often explicit (aka manifest)**
  - Specified in text (at variable declaration)
    - C, C++, Java, C#
  - But may also be inferred – compiler determines type based on usage
    - OCaml, C# and Go (limited)
Dynamic Type Checking

- **During** program execution
  - Can determine type from run-time value
  - Type is checked before use
  - Disallowed operations cause run-time exception
    - Type errors may be latent in code for a long time
- **Dynamic types are** *not manifest*
  - Variables are just introduced/used without types
  - Examples
    - Ruby, Python, Javascript, Lisp
Static and Dynamic Typing

- **Ruby** is dynamically typed, **C** is statically typed

<table>
<thead>
<tr>
<th># Ruby</th>
<th>/* C */</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 3</code></td>
<td><code>int x;</code></td>
</tr>
<tr>
<td><code>x = &quot;foo&quot;</code> # gives x a new type</td>
<td><code>x = 3;</code></td>
</tr>
<tr>
<td><code>x.foo</code> # NoMethodError # at runtime</td>
<td><code>x = &quot;foo&quot;; /* not allowed */</code></td>
</tr>
<tr>
<td></td>
<td><code>/* program doesn’t compile */</code></td>
</tr>
</tbody>
</table>

- **Notes**
  - Can always run the Ruby program; may fail when run
  - C variables declared, with types
    - Ruby variables declared *implicitly*
    - Implicit declarations most natural with dynamic typing
Tradeoffs?

Static type checking

- More work for programmer (at first)
  - Catches more (and subtle) errors at compile time
- Precludes some correct programs
  - May require a contorted rewrite
- More efficient code (fewer run-time checks)

Dynamic type checking

- Less work for programmer (at first)
  - Delays some errors to run time
- Allows more programs
  - Including ones that will fail
- Less efficient code (more run-time checks)
Java: *Mostly* Static Typing

- In Java, types are mostly checked statically
  ```java
  Object x = new Object();
x.println("hello");  // No such method error at compile time
  ```

- But sometimes checks occur at run-time
  ```java
  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)
  ```
Quiz 1: Get out your clickers!

True or false: This program has a type error

```
# Ruby
b = "foo"
a = 30
a = b
```

A. True  
B. False
Quiz 1: Get out your clickers!

- **True or false:** This program has a type error

  ```ruby
  # Ruby
  b = "foo"
  a = 30
  a = b
  ```

  - A. True
  - B. False

- **True or false:** This program has a type error

  ```c
  /* C */
  void foo() {
      int a = 3;
      char *b = "foo";
      a = b;
  }
  ```

  - A. True
  - B. False
Quiz 1: Get out your clickers!

- **True or false:** This program has a type error
  
  ```ruby
  # Ruby
  b = "foo"
  a = 30
  a = b
  ```

  A. True  
  B. False

- **True or false:** This program has a type error
  
  ```c
  /* C */
  void foo() {
      int a = 3;
      char *b = "foo";
      a = b;
  }
  ```

  A. True  
  B. False
Control Statements in Ruby

- A control statement is one that affects which instruction is executed next
  - While loops
  - Conditionals

```ruby
i = 0
while i < n
  i = i + 1
end
```

```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
```
Conditionals and Loops Must End!

All Ruby conditional and looping statements must be terminated with the `end` keyword.

Examples

- if grade >= 90 then
  puts "You got an A"
end

- if grade >= 90 then
  puts "You got an A"
else
  puts "No A, sorry"
end

- i = 0
  while i < n
    i = i + 1
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!**
Quiz 2: What is the output?

```
x = 0
if x then
  puts "true"
elsif x == 0 then
  puts "== 0"
else
  puts "false"
end
```

A. Nothing – there’s an error
B. “false”
C. “== 0”
D. “true”
Quiz 2: What is the output?

```ruby
x = 0
if x then
  puts "true"
elsif x == 0 then
  puts "== 0"
else
  puts "false"
end
```

A. Nothing – there’s an error
B. “false”
C. “== 0”
D. “true”

x is neither false nor nil so the first guard is satisfied