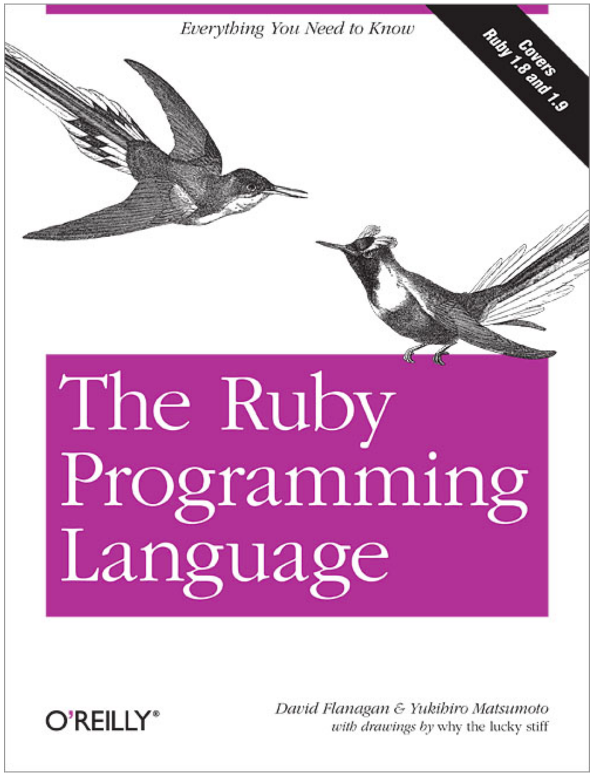
CMSC 330:
Organization of Programming Languages

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


Everything You Need to Know
Covers Ruby 1.8 and 1.9
The Ruby Programming Language
O'REILLY®
David Flanagan & Yukihiro Matsumoto
with drawings by why the lucky stiff

- 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

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

...
```

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

ruby1.rb:

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

```
% 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

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

line break separates expressions (can also use ";")

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

```
# Ruby
x = 3
x = "foo"   # gives x a
            # new type
x.foo       # NoMethodError
            # at runtime
```

```
/* C */
int x;
x = 3;
x = "foo"; /* not allowed */
/* program doesn’t compile */
```

- 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

```
Object x = new Object();
x.println(“hello”);   // No such method error at compile time
```

- But sometimes checks occur at run-time

```
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
b = “foo”
a = 30
a = b
```

A. True
B. False

- True or false: This program has a type error

```
/* 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
b = “foo”
a = 30
a = b
```

A. True
B. False

- True or false: This program has a type error

```
/* 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

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

```
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

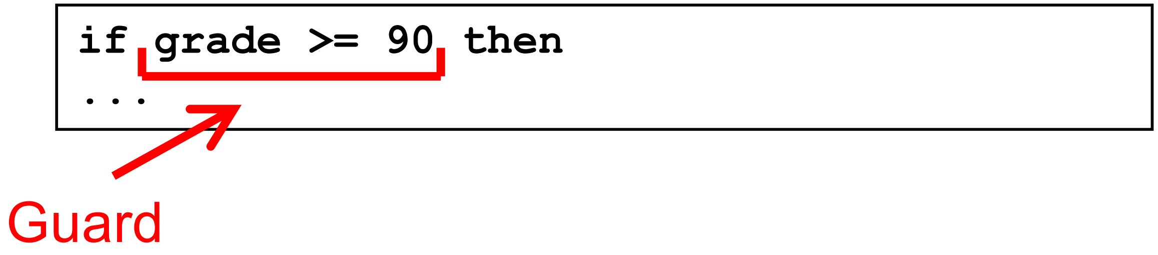
  -
    ```
    if grade >= 90 then
      puts "You got an A"
    end
    ```

  -
    ```
    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?

```
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