Our topic sequence:

- Functional programming with Haskell (Done!)
- Imperative and object-oriented programming using dynamic typing with Ruby
- Logic programming with Prolog
- Whatever else in the realm of programming languages that we find interesting and have time for.
Introduction
whm wrote:
> I ran into John Cropper in the mailroom a few minutes ago. He said
> he was out at your place today and that you're doing well. I
> understand you've got a meeting coming up regarding math in your
> weaving book -- sounds like fun!?

Hi, William

I'm doing well in the sense of surviving longer than expected. But I'm still a sick person without much energy and with a lot of pain.

> My first lecture on Ruby is tomorrow. Ruby was cooked up by a
> Japanese fellow. Judging by the number of different ways to do the
> same thing, I wonder if Japanese has a word like "no".

Interesting. I know nothing about Ruby, but I've noticed it's getting a lot of press, so there must be something to it.
"A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write." — ruby-lang.org

Ruby is commonly described as an "object-oriented scripting language".

- I don't like the term "scripting language"!
- I describe Ruby as a dynamically typed object-oriented language.

Ruby on Rails, a web application framework, has largely driven Ruby's popularity.

Ruby was invented by Yukihiro Matsumoto ("Matz"), a "Japanese amateur language designer", in his own words.
Here is a second-hand excerpt of a posting by Matz:
"Well, Ruby was born on February 24, 1993. I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising."

Another quote from Matz:
"I believe that the purpose of life is, at least in part, to be happy. Based on this belief, Ruby is designed to make programming not only easy but also fun. It allows you to concentrate on the creative side of programming, with less stress. If you don’t believe me, read this book [the "pickaxe" book] and try Ruby. I’m sure you’ll find out for yourself."
ISO/IEC 30170:2012 is an international standard for Ruby but the language is effectively defined by MRI—Matz' Ruby Implementation.

The most recent stable version of MRI is 2.5.0.

On lectura we'll use rvm (the Ruby Version Manager) to run version 2.2.4.

macOS from Mavericks through Sierra has Ruby 2.0.0. High Sierra has Ruby 2.3.3.

The last major upheaval in Ruby occurred between 1.8 and 1.9. (2007)

In general, there are few incompatibilities between 1.9.3 (2011) and the latest version.

Notable: In 2.4, Fixnum and Bignum were unified to Integer.
Resources

*The Ruby Programming Language* by David Flanagan and Matz
   – Perhaps the best book on Safari that covers 1.9 (along with 1.8)
   – I'll call it "RPL".

*Programming Ruby 1.9 & 2.0 (4th edition): The Pragmatic Programmers' Guide* by Dave Thomas, with Chad Fowler and Andy Hunt
   – Known as the "Pickaxe book"
   – $28 for a DRM-free PDF at [pragprog.com](http://pragprog.com).
   – I'll call it "PA".

O'Reilly Safari has:
   • Many relatively new Ruby books
   One recommendation: *The Ruby Way, 3rd edition* by Hal Fulton
   • Lots of books on Ruby on Rails
   • Lots of pre-1.9 Ruby books
ruby-lang.org
  • Ruby's home page

ruby-doc.org
  • Documentation

  • Here's a sample URL, for the String class in 2.2.4:  
    http://ruby-doc.org/core-2.2.4/String.html

  • Suggestion: Create a Chrome "search engine" named rc ("Ruby class") with the following URL template:
    http://www.ruby-doc.org/core-2.2.4/%s.html
    (See http://cs.arizona.edu/~whm/o1nav.pdf)
Getting and running Ruby

Ruby as supplied by Apple with recent versions of macOS, should be fine for our purposes.

I installed Ruby 2.2.0 on my Mac using MacPorts. The "port" is ruby22.

Lots of people install Ruby versions using the Homebrew package manager.
Getting Ruby for Windows

Go to http://rubyinstaller.org/downloads and get "Ruby 2.2.6" (not x64)

When installing, I recommend these selections:
   Add Ruby executables to your PATH
   Associate .rb and .rbw files with this Ruby installation
**Running Ruby on lectura**

**rvm** is the Ruby Version Manager. It lets one easily select a particular version of Ruby to work with.

On lectura, we can select Ruby 2.2.4 and then check the version like this:

```
% rvm 2.2.4
% ruby --version
ruby 2.2.4p230 (2015-12-16 revision 53155) [x86_64-linux]
```

Depending on your Bash configuration, **rvm** may produce a message like "Warning! PATH is not properly set up..." but if **ruby --version** shows 2.2.4, all is well.

Note: **rvm** does not work with **ksh**. If you're running **ksh**, let us know.
IMPORTANT: you must either

1. Do `rvm 2.2.4` each time you login on lectura.
   —OR—

2. Add the command `rvm 2.2.4` to one of your Bash start-up files.

There are a variety of ways in which Bash start-up files can be configured.

- With the default configuration for CS accounts, add the line
  `rvm 2.2.4 >& /dev/null`
  at the end of your `~/.profile`.

- If you're using the configuration suggested in my Fall 2015 352 slides, put
  `rvm 2.2.4 >& /dev/null`
  at the end of your `~/.bashrc`.

- Let us know if you have trouble with this.
irb—Interactive Ruby Shell

irb, *Interactive Ruby Shell*, provides a REPL for Ruby.

irb evaluates expressions as they are typed.

$ irb
   > "abc" + "12"
   => "abc12"

If an expression is definitely incomplete, irb displays an alternate prompt:

   >> 1.23 +
   ?> 2e3
   => 2001.23

Note: To save space on the slides, the result line (=> ...) won't be shown when it's uninteresting.

Control-D terminates irb.
Collaborative Learning Exercise

[cs.arizona.edu/classes/cs372/spring18/cle-7.html]

Note to self: push-cle 7
I use a Bash alias for `irb` that requests a simple prompt and activates auto-completion:

```
alias irb="irb --prompt simple -r irb/completion"
```

When `irb` starts up, it first processes `~/irbrc`, if present.

`spring18/ruby/dotirbrc` is a recommended starter `~/.irbrc` file.

```
% cp /cs/www/classes/cs372/spring18/ruby/dotirbrc ~/.irbrc
```

On Windows you might use a batch file named `irbs.bat` to start with those options. Example:

```
W:\372\ruby> type irbs.bat
irb --prompt simple -r irb/completion
```

Run it by typing `irbs` (not just `irb`).
I like "it" better than underscore for referencing the previous result.

With the ~/.irbrc suggested on the previous slide, I can use "it" to reference the last result:

```ruby
>>> 3 + 4
=> 7

>>> it * 3
=> 21

>>> it.to_s
=> "21"
```

Observation: Ruby's flexibility lets me use a convention I like.
Ruby basics
Every value is an object

In Ruby, every value is an object.

Methods can be invoked using receiver.method(parameters...)

```ruby
>> "testing".count("t")    # How many "t"s are there?
  => 2

>> "ostentatious".tr("aeiou","12345")

>> "testing".length()
```

Repeat: In Ruby, every value is an object.

What are some values in Java that are not objects?
Everything is an object, continued

Of course, "everything" includes numbers:

```
>> 1.2.class()
=> Float

>> (10-20).class()
=> Fixnum

>> 17**25
=> 5770627412348402378939569991057

>> it.succ()  # Remember: the custom .irbrc is needed to use "it"
=> 5770627412348402378939569991058

>> it.class()
=> Bignum
```
Everything is an object, continued

The TAB key can be used to show completions in `irb`:

```ruby
>> 100.<TAB><TAB>
Display all 107 possibilities? (y or n)
100.__id__         100.display
100.__send__       100.div
100.abs            100.divmod
100.abs2           100.downto
100.angle          100.dup
100.arg            100.enum_for
100.between?       100.eql?
100.ceil           100.equal?
100.chr            100.even?
100.class          100.extend
100.clone          100.fdiv
100.coerce         100.floor
100.conj           100.freeze
100.conjugate      100.frozen?
100.define_singleton_method 100.gcd
100.denominator    100.gcdlcm
```
Parentheses are optional, sometimes

Parentheses are **often** optional in method invocations:

```ruby
>> 1.2.class
=> Float

>> "testing".count "aeiou"
=> 2
```

But, the following case fails. (Why?)

```ruby
>> "testing".count "aeiou".class
TypeError: no implicit conversion of Class into String
  from (irb):17:in `count'
```

Solution:

```ruby
>> "testing".count("aeiou").class
=> Fixnum
```

I usually omit parentheses in simple method invocations.
A post-Haskell hazard!

Don't let the optional parentheses make you have a Haskell moment and leave out a comma between arguments:

```ruby
>> "testing".slice 2 3
SyntaxError: (irb):20: syntax error, unexpected tINTEGER,
expecting end-of-input
```

Commas are required between arguments!

```ruby
>> "testing".slice 2,3
=> "sti"
```

I almost always use parentheses when there's more than one argument:

```ruby
>> "testing".slice(2,3)
=> "sti"
```
Operators are methods, too

Ruby operators are methods with symbolic names.

In general,

```
expr1 op expr2
```

means

```
expr1.op(expr2)
```

Example:

```
>> 3 + 4
=> 7
```

```
>> 3.+(4)
=> 7
```

```
>> "abc".==(97.chr.+("bc"))
```

CSC 372 Spring 2018, Ruby Slide 25
The **Kernel** module has methods for I/O and more. Methods in **Kernel** can be invoked with only the method name.

```ruby
>> puts "hello"
hello
=> nil

>> printf("sum = %d, product = %d\n", 3+4, 3 * 4)
sum = 7, product = 12
=> nil

>> puts gets.inspect
What can say about value, type and side-effect for `puts` and `printf`?
testing
"testing\n"
=> nil
```

See [http://ruby-doc.org/core-2.2.4/Kernel.html](http://ruby-doc.org/core-2.2.4/Kernel.html)
Extra Credit Assignment 2

For two assignment points of extra credit:

1. Run `irb` somewhere and try ten Ruby expressions with some degree of variety.

2. Capture the interaction (both expressions and results) and put it in a plain text file, `eca2.txt`. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in `eca2.txt` with the following command:
   
   \[
   \text{% turnin 372-eca2 eca2.txt}
   \]

Due: At the start of the next lecture after we hit this slide.

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.
A LHtLaL suggestion:
Start accumulating a file of brief notes on Ruby. Example:

$ cat ~/notes/ruby.txt
#running
rvm 2.2.4 to select version on lectura
irb is REPL, reads ~/.irbrc

#irb
_ is last value

#misc
Every value is an object
Can often omit parens on methods:
   3.class, "testing".count "t"
Operators are methods: 3+4 is really 3.+(4)

#i/o
gets, puts, printf (in Kernel module)
Executing Ruby code in a file

The **ruby** command can be used to execute Ruby source code contained in a file.

By convention, Ruby files have the suffix `.rb`.

Here is "Hello" in Ruby:

```ruby
% cat hello.rb
puts "Hello, world!"
```

```bash
% ruby hello.rb
Hello, world!
```

Note that the code does not need to be enclosed in a method—"top level" expressions are evaluated when encountered.

There is no evident compilation step or artifact produced. It just runs!
Alternatively, code can be placed in a method that is invoked by an expression at the top level:

```
% cat hello2.rb
def say_hello
  puts "Hello, world!"
end

say_hello

% ruby hello2.rb
Hello, world!
```

The definition of `say_hello` must precede the call.

We'll see later that Ruby is somewhat sensitive to newlines.
A line-numbering program

The program below reads lines from standard input and writes each, with a line number, to standard output:

```ruby
line_num = 1     # numlines.rb

while line = gets
    printf("%3d: %s", line_num, line)
    line_num += 1  # Ruby does not have ++ and --
end
```

Execution:

```
% ruby numlines.rb < hello2.rb
 1: def say_hello
 2:   puts "Hello, world!"
 3: end
 4: 
 5: say_hello
```
Problem: Write a program that reads lines from standard input and writes them in reverse order to standard output. Use only the Ruby you've seen.

For reference, here's the line-numbering program:

```ruby
line_num = 1
while line = gets
  printf("%3d: %s", line_num, line)
  line_num += 1
end
```

Solution:
Some basic types
The value `nil`

`nil` is Ruby's "no value" value. The name `nil` references the only instance of the class.

```ruby
>> nil
=> nil
```

```ruby
>> nil.class
=> NilClass

>> nil.object_id
=> 4
```

We'll see that Ruby uses `nil` in a variety of ways.

Speculate: What happens if we use a variable that hasn't been assigned to?
Instances of Ruby's **String** class represent character strings.

A variety of "escapes" are recognized in double-quoted string literals:
```
>> puts "newline >\n< and tab >\t"
newline >
< and tab > <

>> "\n\t".length
=> 3

>> "Newlines: octal \012, hex \xa, control-j \cj"
=> "Newlines: octal \n, hex \n, control-j \n"
```

Section 3.2, page 49 in RPL has the full list of escapes.
In single-quoted string literals only \' and \" are recognized as escapes:

```ruby
>> puts '\n\t'
\n\t
=> nil

>> '\n\t'.length  # Four chars: backslash, n, backslash, t
=> 4

>> puts '\\\\'
\ 
=> nil

>> '\\\\'.length  # Two characters: apostrophe, backslash
=> 2
```
The `public_methods` method shows the public methods that are available for an object. Here are some of the methods for `String`:

```ruby
>> "abc".public_methods.sort
=> [:!, :!=, :!~, :%, :*, :+, :<, :<<, :<=, :<=>, :==, ===, :=~, :
:>, :>=, [:], [[]]=, __id__, __send__, ascii_only?,
:between?, :bytes, bytesize, byteslice, capitalize, 
capitalize!, :casecmp, center, chars, chomp, chomp!, :chop, 
:chop!, :chr, :class, :clear, :clone, :codepoints, :concat, :count, 
crypt, :define_singleton_method, :delete, :delete!, :display, 
downcase, :downcase!, :dump, :dup, :each_byte, :each_char, 
each_codepoint, :each_line, :empty?, ...

>> "abc".public_methods.length
=> 169
```
Unlike Java, Haskell, and many other languages, strings in Ruby are mutable.

If two variables reference a string and the string is changed, the change is reflected by both variables:

```ruby
>> x = "testing"

>> y = x       # x and y now reference the same instance of String

>> y << " this"       # the << operator appends a string

>> y

>> x

Is it a good idea to have mutable strings?
Strings are mutable, continued

The `dup` method produces a copy of a string.

```ruby
>> x = "testing"
>> y = x.dup
=> "testing"
```

```ruby
>> y << "...more"

>> y

>> x
```

Some objects that hold strings `dup` the string when the string is added to the object.
Sidebar: Applicative vs. imperative methods

Some methods have both an *applicative* and an *imperative* form.

**String**'s `upcase` method is *applicative*—it produces a new **String** but doesn't change its *receiver*, the instance of **String** on which it's called:

```ruby
>> s = "testing"
=> "testing"

>> s.upcase
=> "TESTING"

>> s
```

String's `upcase` method is applicative—it produces a new String but doesn't change its receiver, the instance of String on which it's called:
In contrast, an *imperative* method potentially changes its receiver.

String's `upcase!` method is the imperative counterpart to `upcase`:

```ruby
>> s.upcase!
=> "TESTING"
```

A Ruby convention:

When methods have both an applicative and an imperative form, the imperative form ends with an exclamation mark.
Strings can be compared with a typical set of operators:

```ruby
>> s1 = "apple"

>> s2 = "testing"

>> s1 == s2
=> false

>> s1 != s2
=> true

>> s1 < s2
=> true
```

We'll talk about details of `true` and `false` later.

These operators work with most other types, too!
There is also a comparison operator: `<=>`

Behavior:

```
>> "apple" <=> "testing"
```

```
>> "testing" <=> "apple"
```

```
>> "x" <=> "x"
```

Speculate: How is the operator `<=>` read aloud by some programmers?

What are the Java and C analogs for `<=>` when applied to strings?
Substrings

Subscripting a string with a number produces a one-character string.

```ruby
>> s="abcd"

>> s[0]  # Positions are zero-based

>> s[1]

>> s[-1]  # Negative positions are counted from the right

>> s[100]
=> nil  # An out-of-bounds reference produces nil

Historical note: With Ruby versions prior to 1.9, "abc"[0] is 97.

Why doesn't Java provide s[n] instead of s.charAt(n)?
A subscripted string can be the target of an assignment. A string of any length can be assigned.

```ruby
>> s = "abc"
=> "abc"

>> s[0] = 65.chr

>> s[1] = "tomi"

>> s

>> s[-3] = ""

>> s
```
A substring can be referenced with $s[start, length]$

```ruby
>> s = "replace"

>> s[2,3]

>> s[3,100]

>> s[-4,3]

>> s[10,10]
```

Substrings, continued
Instances of Ruby's `Range` class represent a range of values. A `Range` can be used to reference a substring.

```ruby
>> r = 2..-2
=> 2..-2

>> r.class
=> Range

>> s = "replaced"

>> s[r]

>> s[r] = ""

>> s
```
Substrings with ranges, continued

It's more common to use literal ranges with strings:

```ruby
>> s = "rebuilding"
>> s[2..-1]
=> "build"
>> s[2..-4]
=> "build" @a
>> s[2...-4]
=> "buil"  # three dots is "up to" @a
>> s[-8..-4]
>> s[-4..-8]
```

CSC 372 Spring 2018, Ruby Slide 48
A substring can be the target of an assignment:

```ruby
>> s = "replace"
>> s[0,2] = ""

> s

>> s[3..-1] = "naria"
>> s

>> s["aria"] = "kton"

>> s
```

Changing substrings:

```
place
0 1 2 3 4 5 6
7 6 5 4 3 2 1 (negative)
```

```
place
0 1 2 3 4
5 4 3 2 1 (negative)
```
Interpolation in string literals

In a string literal enclosed with double quotes the sequence \texttt{#{expr}} causes interpolation of \texttt{expr}, an arbitrary Ruby expression.

\begin{verbatim}
>> x = 10

>> y = "twenty"

>> s = "x = #{x}, y + y = #{y + y}"
=> "x = 10, y + y = twentytwenty"

>> puts "There are #{"".public_methods.length} string methods"
There are 169 string methods

>> "test #{"#{"abc".length*4}"}"    # Arbitrary nesting works
\end{verbatim}

It's idiomatic to use interpolation rather than concatenation to build a string from multiple values.
With 2.2.4 on lectura, integers in the range \(-2^{62}\) to \(2^{62}-1\) are represented by instances of **Fixnum**. If an operation produces a number outside of that range, the value is represented with a **Bignum**.

```
>> x = 2**62-1
=> 4611686018427387903

>> x.class
=> Fixnum

>> x += 1
=> 4611686018427387904

>> x.class
=> Bignum

>> x -= 1
=> 4611686018427387903

>> x.class
=> Fixnum
```

Is this automatic transitioning between **Fixnum** and **Bignum** a good idea? How do other languages handle this?
The **Float** class represents floating point numbers that can be represented by a double-precision floating point number on the host architecture.

```ruby
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.111075555498667

>> x = x / 0.0

>> (0.0/0.0).nan?
```
Arithmetic on two Fixnums produces a Fixnum.

```ruby
>> 2/3
=> 0

>> it.class
=> Fixnum
```

Fixnums and Floats can be mixed. The result is a Float.

```ruby
>> 10 / 5.1
=> 1.9607843137254903

>> 10 % 4.5
=> 1.0

>> it.class
=> Float
```
Ruby has a `Complex` type.

```ruby
>> x = Complex(2,3)
=> (2+3i)

>> x * 2 + 7
=> (11+6i)

>> Complex 'i'

>> it ** 2
```
There's `Rational`, too.

```ruby
>> Rational(1,3)
=> (1/3)

>> it * 300

>> Rational 0.5

>> Rational 0.6

>> Rational 0.015625
```
Unlike some languages, Ruby does not automatically convert strings to numbers and numbers to strings as needed.

```
>> 10 + "20"
TypeError: String can't be coerced into Fixnum
```

The methods `to_i`, `to_f`, and `to_s` are used to convert values to **Fixnums**, **Floats** and **Strings**, respectively.

```
>> 10.to_s + "20"
>> 10 + "20".to_f
>> 10 + 20.9.to_i
```

```
>> 33.to_<TAB><TAB>
33.to_c     33.to_int
33.to_enum  33.to_r
33.to_f     33.to_s
33.to_i
```
A sequence of values is typically represented in Ruby by an instance of the **Array** class.

An array can be created by enclosing a comma-separated sequence of values in square brackets:

```ruby
>> a1 = [10, 20, 30]
=> [10, 20, 30]

>> a2 = ["ten", 20, 30.0, 2**40]
=> ["ten", 20, 30.0, 1099511627776]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 1099511627776], [[[10, 20, 30]]]]
```

What's a difference between Ruby arrays and Haskell lists?
Arrays, continued

Array elements and subarrays (sometimes called slices) are specified with a notation like that used for strings.

```ruby
>> a = [1, "two", 3.0, %w{a b c d}]
=> [1, "two", 3.0, ["a", "b", "c", "d"]]
```

```ruby
>> a[0]
```

```ruby
>> a[1,2]  # a[start, length]
```

```ruby
>> a[-1]
```

```ruby
>> a[-1][-2]
```
Elements and subarrays can be assigned to. Ruby accommodates a variety of cases; here are some:

```ruby
>> a = [10, 20, 30, 40, 50, 60]
>> a[1] = "twenty"; a
>> a[2..4] = %w{a b c d e}; a
>> a[1..-1] = []; a
```

Semicolon separates expressions. We make a change and show new value.
A few more:

```
>> a
=> [10]

>> a[0] = [1,2,3]; a

>> a[4] = [5,6]; a

>> a[0,2] = %w{a bb ccc}; a
```

What's important to retain from the examples above?

- Elements of arrays and subarrays can be assigned to.
- Lots of rules; some complex.
A variety of other operations are provided for arrays. Here's a sampling:

```ruby
>>> a = []

>>> a << 1; a

>>> a << [2,3,4]; a

>>> a.reverse; a

>>> a.reverse!; a
```
A few more:

```ruby
>> a
=> [[2, 3, 4], 1]

>> a[0].shift

>> a

>> a.unshift "a","b","c"

>> a.shuffle.shuffle
```
Even more!

```ruby
>> a = [1,2,3,4]; b = [1,3,5]

>> a + b

>> a - b

>> a & b

>> a | b

>> ('a..'..'zzz').to_a.size
```
Comparing arrays

We can compare arrays with `==` and `!=`. Elements are compared in turn, possibly recursively.

```ruby
>> [1,2,3] == [1,2]  

>> [1,2,[3,"bcd"]]== [1,2] + [[3, "abcde"]]

>> [1,2,[3,"bcd"]]== [1,2] + [[3, "abcde"[1..-2]]]
```
Comparison of arrays with $\leqslant$ is lexicographic.

$$\gg [1,2,3,4] \leqslant [1,2,10]$$
$$=> -1$$

$$\gg [[10,20], [2,30], [5,"x"]].sort$$
$$=> [[2, 30], [5, "x"], [10, 20]]$$
Comparing arrays

Comparison with $\langle=\rangle$ produces `nil` if differing types are encountered.

$$
\begin{align*}
\text{\triangleright} \ [1,2,3,4] & \langle=\rangle [1,2,3,"four"] \\
& \Rightarrow \text{nil}
\end{align*}
$$

$$
\begin{align*}
\text{\triangleright} \ [[10,20],[5,30],[5,"x"]].\text{sort} \\
& \text{ArgumentError: comparison of Array with Array failed}
\end{align*}
$$

Here's a simpler failing case. Should it be allowed?

$$
\begin{align*}
\text{\triangleright} \ ["sixty",20,"two"].\text{sort} \\
& \text{ArgumentError: comparison of String with 20 failed}
\end{align*}
$$
Comparing arrays, continued

At hand:

```ruby
>> ["sixty",20,"two"].sort
ArgumentError: comparison of String with 20 failed
```

Contrast with Icon:

```icon
]  
  [ sort(["sixty",20,"two"]) 
    r := [20,"sixty","two"] (list)

]  
  [ sort([3.0, 7, 2, "a", "A", ":", [2], [1], -1.0]) 
    r := [2, 7, -1.0, 3.0, ":", "A", "a", [2], [1]] (list)
```

What does Icon do better? What does Icon do worse?

Here's Python 2:

```python
>>> sorted([3.0, 7, 2, "a", "A", ":", [2], [1], -1.0])
[-1.0, 2, 3.0, 7, [1], [2], ":", 'A', 'a']
```
An array can hold a reference to itself:

```ruby
>> a = [1,2,3]

>> a.push a

>> a.size

>> a[-1]

>> a[-1][-1][-1]
```

Arrays can be cyclic
Type Checking
"The Java programming language is a \textit{statically typed} language, which means that every variable and every expression has a type that is known at compile time."

\begin{quote}
\textit{The Java Language Specification, Java SE 9 Edition}
\end{quote}

Example: assume the following...

\begin{verbatim}
int i = ...;  String s = ...;  Object o = ...;  static long f(int n);
\end{verbatim}

What are the types of the following expressions?

\begin{verbatim}
i + 5
i + s
s + o
o + o
o.hashCode()
f(i.hashCode()) + 3F
i = i + s
\end{verbatim}

Did we need to know any values or execute any code to know those types?
If there's a type error in a Java program, `javac` doesn't produce a `.class` file—there's nothing to run.

```java
% cat X.java
public class X {
    public static void main(String args[]) {
        System.out.println("Running!");
        System.out.println(3 + new int[5]);
    }
}
```

```bash
% javac X.java
X.java:4: error: bad operand types for binary operator '+'
    System.out.println(3 + new int[5]);
    ^
```

```bash
% java X
Error: Could not find or load main class X
- The file `X.class` wasn't created.
- Hazard: If we don't notice the `javac` error, we might run a copy of `X.class` created by a previous `javac X.java` invocation.
```
Java does type checking based on the declared types of variables and the intrinsic types of literals.

Haskell supports type declarations but we know that it also provides type inferencing:

> :set +t
> f x y z = (isLetter $ head $ [x] ++ y) && z
f :: Char -> [Char] -> Bool -> Bool

Haskell, too, is a statically typed language—the type of every expression can be determined by analyzing the code.
Static typing, continued

With a statically typed language:

• The type for all expressions is determined when a body of code is compiled/loaded/etc.

• All* type inconsistencies that exist are discovered at that time.

• Execution doesn't begin if errors exist.
  o Sometimes manifested by no "executable" file being produced.
  o With ghci, functions in a file with an error aren't loaded.
Important:

A statically typed language lets us guarantee that certain kinds of errors
don't exist without having to run any code.

If a Java program compiles, we can be absolutely sure no errors of the
following sort (and more) exist:

- Dividing a string by a float
- Concatenating a number with a list
- Subscripting a value that's not an array
- Misspelling the name of a method (true?)
- Invoking a method with the wrong number of arguments
- Forgetting a `return` at the end of a method that returns a value

Exception: Errors due to casts and boxing/unboxing can still exist!
How often did your Haskell code run correctly as soon as the type errors were fixed?

How does that compare with your experience with Java?
   With C?
   With Python?
   With others?

Paul Hudak of Yale wrote,
"The best news is that Haskell's type system will tell you if your program is well-typed before you run it. This is a big advantage because most programming errors are manifested as typing errors."

Do you agree with Hudak?
Sidebar: Interpreted or compiled?

A common misunderstanding:
Python and Ruby are interpreted languages.
Java and C are compiled languages.

The fact:
Interpretation and compilation are attributes of an implementation of a language, not the language itself!

A simple, polarized viewpoint:
• Interpreters execute source code as-is.
• Compilers translate source code into machine code.

Reality:
• Language implementations use a variety of techniques that each fall along a broad spectrum from interpretation to compilation.
• A particular implementation of any language can be made to fall at either end of that spectrum, or anywhere in the middle.
In Java, variables are declared to have a type.

Variables in Ruby do not have a type. Instead, type is associated with values.

```ruby
>> x = 10
>> x.class # What's the class of the object held by x?
=> Fixnum

>> x = "ten"
>> x.class
=> String

>> x = 2**100
>> x.class
=> Bignum
```
Array's `sample` method returns a random element of the receiver.

```ruby
>> a = ["one", 2, [3], 4.0]
>> a.sample  #=> [3]
>> a.sample  #=> "one"
```

What's the type of `a.sample + a.sample`?

```ruby
>> (a.sample + a.sample).class
=> String
```

```ruby
>> (a.sample + a.sample).class
=> Float
```

The type of `a.sample`, `a.sample + a.sample`, `a.sample[0]`, etc. is unknown until the expression is evaluated!
Ruby is a dynamically typed language.

Consider this Ruby method:
```ruby
def f x, y, z
  return x[y + z] * x.foo
end
```

For some combinations of types it will produce a value. For others it will produce a **TypeError**.

With a dynamically typed language, types are not checked until an expression is evaluated.

Another way to say it:
- There is no static analysis of the types in Ruby expressions.
Dynamic typing, continued

With dynamic typing, no type checking is done when code is loaded. Instead, types of values are checked during execution, as each operation is performed.

Consider this Ruby code:

```ruby
while line = gets
    puts(line[-x] + y)
end
```

What types must be checked each time through that loop?

• Is `gets` a method or a value?
• Can `x` be negated?
• Is `line` subscriptable with `-x`?
• Can `line[-x]` and `y` be added?
• Is `puts` a method?
Performance implications with dynamic typing

One performance cost with a dynamically typed language is execution-time type checking.

Evaluating the expression \( x + y \) might require decision-making like this:

- if both \( x \) and \( y \) are integers
  - add them
- else if both \( x \) and \( y \) are floats
  - add them
- else if one is a float and the other an integer
  - convert the integer to a float and add them
- else if both \( x \) and \( y \) are strings
  - concatenate them
...and more...

If that \( x + y \) is in a loop, that decision making is done every time around.

Note: The above "implementation" can be improved upon in many ways.
In contrast, consider this Java method:

```java
int count(int wanted, int[] values) {
    int result = 0;
    for (int value : values)
        if (value == wanted)
            result += 1;
    return result;
}
```

Generated virtual machine code:

```
0: iconst_0 18: aload_3
1: istore_2 19: iload 5
2: aload_1 21: iaload
3: astore_3 22: astore_6
4: aload_3 24: iinc 2, 1
5: arraylength 26: iload_0
6: astore_4 27: if_icmpne 33
8: iconst_0 30: iinc 2, 1
9: astore_5 33: iinc 5, 1
11: iload 5 36: goto 11
13: iload 4 39: iload_2
15: if_icmpge 39 40: ireturn
```

See also [https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings](https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings)
Static vs. dynamic typing

With respect to static typing, what are the implications of dynamic typing for...

- Loading ("compilation") speed?
- Execution speed?
- The likelihood of code to be correct?
A long-standing question in industry: Can a good test suite find type errors in dynamically typed code as effectively as static type checking?

What's a "good" test suite?
- Full code coverage? (every line executed by some test)
- Full path coverage? (all combinations of paths exercised)
- How about functions whose return type varies?

But wouldn't we want a good test suite no matter what language we're using?

"Why have to write tests for things a compiler can catch?"
—Brendan Jennings, SigFig
What ultimately matters?

What does the user of software care about?
• Dynamic vs. static typing? Ruby vs. Java?
  ∘ No!

• Software that works
  ∘ Facebook game vs. radiation therapy system

• Fast enough
  ∘ When does 10ms vs. 50ms matter?

• Better sooner than later
  ∘ "First to volume" can be the key to success for a company.
  ∘ A demo that's a day late for a trade show isn't worth much.

• Affordable
  ∘ "People will pay to stop the pain." – Doug Higgins
  ∘ I'd pay a LOT for a great system for writing slides.
Java is statically typed but casts introduce the possibility of a type error not being detected until execution.

C is statically typed but has casts that allow type errors during execution that are never detected.

Ruby, Python, and Icon have no static type checking whatsoever, but type errors during execution are always detected.

An example of a typing-related trade-off in execution time:
- C spends zero time during execution checking types.
- Java checks types during execution only in certain cases.
- Languages with dynamic typing check types on every operation, at least conceptually.
"Why?" vs. "Why Not?"
"Why?" or "Why not?"

When designing a language some designers ask, "Why should feature X be included?"

Some designers ask the opposite: "Why should feature X not be included?"

Let's explore that question with Ruby.
A "here document" is a third way to literally specify a string in Ruby:

```
>> s = <<-XYZZY
   +-------+
   |      **|
   |      */ |
   |     ' ' |
   +-------+

XYZZY
=> " +-------+
+\n |      **| \n |      */ | \n |     ' ' | \n+-------+\n"
```

The string following `<<` specifies a delimiter that ends the literal. (The ending occurrence must be at the start of a line.)

"There's more than one way to do it!"—a Perl motto
Here's another way to specify string literals. See if you can discern some rules from these examples:

```ruby
>> %q{ just testin' this... }
=> " just testin' this... "
```

```ruby
>> %Q|\n\t|
=> "\n\t"
```

```ruby
>> %q(\u0041 is Unicode for A)
=> "\u0041 is Unicode for A"
```

```ruby
>> %q.test.
=> "test"
```
Partial summary of string literal syntax in Ruby:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 5; s = &quot;x is #{x}&quot;</code></td>
<td>&quot;x is 5&quot;</td>
</tr>
<tr>
<td><code>\&quot;\\n\t\&quot;.length</code></td>
<td>6</td>
</tr>
</tbody>
</table>
| `hd = <<X
just
testing
X`                                          | "just\ntesting\n"                                                     |
| `%q{ \n \t } + %Q|\n \t | + %Q(u0021 \u{23})`                                              | " \n \t \n \t ! #"                                                  |

How much is enough?

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many ways does Haskell have to make a string literal?</td>
</tr>
<tr>
<td>How many ways should there be to make a string literal?</td>
</tr>
<tr>
<td>What's the minimum functionality needed?</td>
</tr>
<tr>
<td>Which of Ruby's would you remove?</td>
</tr>
</tbody>
</table>
"Why" or "Why not?" as applied to operator overloading

Here are some examples of operator overloading:

```ruby
>> [1,2,3] + [4,5,6] + [] + [7]
=> [1, 2, 3, 4, 5, 6, 7]

>> "abc" * 5
=> "abcabcabcabcabc"

>> [1, 3, 15, 1, 2, 1, 3, 7] - [3, 2, 1, 3]
=> [15, 7]

>> [10, 20, 30] * "..."

>> "decimal: %d, octal: %o, hex: %x" % [20, 20, 20]
```
What are some ways in which inclusion of a feature impacts a language?

• Increases the "mental footprint" of the language.
  o There are separate mental footprints for reading code and writing code.

• Maybe makes the language more expressive.

• Maybe makes the language useful for new applications.

• Probably increases size of implementation and documentation.

• Might impact performance.
Features come in all sizes!

Small: A new string literal escape sequence ("\U{65}" for "A")

Small: Supporting an operator on a new pair of types

Medium: Support for arbitrary precision integers

Large or small?

Support for object-oriented programming

Support for garbage collection
At one of my first meetings with Ralph Griswold I put forth a number of ideas I had for new features for Icon.

He listened patiently. When I was done he said, "Go ahead. Add all of those you want to."

And then he added, "But for every feature you add, first find one to remove."
The art of language design

There's a lot of science in programming language design but there's art, too.

Excerpt from interview with Perl Guru Damian Conway:

Q: "What languages other than Perl do you enjoy programming in?"
A: "I'm very partial to Icon. It's so beautifully put together, so elegantly proportioned, almost like a Renaissance painting."

http://www.pair.com/pair/current/insider/1201/damianconway.html (404 now!)

"Icon: A general purpose language known for its elegance and grace. Designed by Ralph Griswold to be successor to SNOBOL4."

—Digibarn "Mother Tongues" chart (see Intro slides)
The art of language design, continued

- Between SNOBOL4 and Icon there was there SL5 (SNOBOL Language 5).

- I think of SL5 as an example of the "Second System Effect". It was never released.

- Ralph once said, "I was laying in the hospital thinking about SL5. I felt there must be something simpler."

- That simpler thing turned out to be Icon.
  - SL5 was an expansion
  - Icon was a contraction
Design example: invocation in Icon

Procedure call in Icon:

```icon
][ reverse("programming")
  r := "gnimmargorp" (string)

][ p := reverse
  r := function reverse (procedure)

][ p("foo")
  r := "oof" (string)
```

Doctoral student Steve Wampler added mutual goal directed evaluation (MGDE). A trivial example:

```icon
][ 3("one", 2, "III")

][ (?3)("one", 2, "III")
```
After a CSC 550A lecture where Ralph introduced MGDE, I asked, "How about 'string invocation', so that "+(3,4) would be 7?"

What do you suppose Ralph said?
"How would we distinguish between unary and binary operators?"

Solution: Discriminate based on the operand count!

```
)] -(5,3)
  r := 2  (integer)
)] -(5)
  r := -5  (integer)
)[ (?"+-")3,4)
  r := -1  (integer)
```

Within a day or two I added string invocation to Icon.

Why did Ralph choose to allow this feature?
He felt it would increase the research potential of Icon.
Design example: Parallel assignment

An interesting language design example in Ruby is *parallel assignment*:

```ruby
>> a, b = 10, [20, 30]

>> a

>> b

>> c, d = b

>> a, b, c = "800-555-1211".split "-"

>> [a,b,c]
```
Parallel assignment, continued

How could we do a swap with parallel assignment?

```ruby
>> x, y = 10, 20
```

Another way?

Contrast:

Icon has a swap operator: `x :=: y`
Parallel assignment, continued

Speculate: Does the following work?

```plaintext
>> a, b, c = [10, 20, 30, 40, 50]
```

Speculate again:

```plaintext
>> a, *b, c = [10, 20, 30, 40, 50]
```

```plaintext
>> a, *b, *c = [10, 20, 30, 40, 50]
```

Section 4.5.5 in RPL has full details on parallel assignment. It is both more complicated and less general than pattern matching in Haskell. (!)
Control Structures
The **while** loop

Here's a loop to print the integers from 1 through 10, one per line.

```ruby
i=1
while i <= 10 do      # "do" is optional
  puts i
  i += 1
end
```

When `i <= 10` produces `false`, control branches to the code following `end`, if any.

The body of the **while** is always terminated with `end`, even if there's only one expression in the body.
Java control structures such as if, while, and for are driven by the result of expressions that produce a value whose type is boolean.

C has a more flexible view: control structures consider a scalar value that is non-zero to be "true".

PHP considers zeroes, the empty string, the string "0", empty arrays, and more to be false.

Python and JavaScript, too, have sets of "truthy" and "falsy/falsey" values.

Here's the Ruby rule:

Any value that is not false or nil is considered to be "true".
Remember: Any value that is not `false` or `nil` is considered to be "true".

Let's analyze this loop, which reads lines from standard input using `gets`.

```ruby
while line = gets
  puts line
end
```

`gets` returns a string that is the next line of the input, or `nil`, on end of file.

The expression `line = gets` has two side effects but also produces a value.

- Side effects: (1) a line is read from standard input and (2) is assigned to `line`.
- Value: The string assigned to `line`.

If the first line from standard input is "one", then the first time through the loop what's evaluated is `while "one"`.

The value "one" is not `false` or `nil`, so the body of the loop is executed, causing "one" to be printed on standard output.

At end of file, `gets` returns `nil`. `nil` is assigned to `line` and produced as the value of the assignment, in turn terminating the loop.
From the previous slide:

```ruby
while line = gets
  puts line
end
```

Partial understanding:
The loop reads and prints every line from standard input.

Full understanding:
What we worked through on the previous slide.

I think there's merit in full understanding.

More examples of full understanding:
• Knowing exactly how `*p++ = *q++` works in C.
• Knowing the rules for field initialization in Java.
• Knowing the full set of truthy/falsy rules for a language.
String's `chomp` method removes a carriage return and/or newline from the end of a string, if present.

Here's a program that's intended to flatten all input lines to a single line:

```ruby
result = ""
while line = gets.chomp
    result += line
end
puts result
```

Will it work?
At hand:

```ruby
result = ""
while line = gets.chomp
  result += line
end
puts result
```

At end of file, `gets` returns `nil`, producing an error on `gets.chomp`.

Which of the two alternatives below is better? What's a third alternative?

| result = ""
while line = gets
  line.chomp!
  result += line
end
puts result |
| result = ""
while line = gets
  result += line.chomp
end
puts result |
| result = ""
while line = gets
  result += line.chomp!
end
puts result |
| result = ""
while line = gets
  result += line
end
puts result |
Problem: Write a `while` loop that prints the characters in the string `s`, one per line. Don't use the `length` or `size` methods of `String`.

Extra credit: Don't use any variables other than `s`.

Solution: `(while5.rb)`

Solution with only `s`: `(while5a.rb)`
Unlike Java, Ruby does pay some attention to the presence of newlines in source code.

For example, a while loop cannot be trivially squashed onto a single line.

```ruby
while i <= 10 puts i; i += 1; end  # Syntax error
```

If we add semicolons where newlines originally were, it works:

```ruby
while i <= 10; puts i; i += 1; end  # OK
```

There is some middle ground, too:

```ruby
while i <= 10 do puts i; i+=1 end  # OK. Note added "do"
```

Unlike Haskell and Python, indentation is never significant in Ruby.
Ruby considers a newline to terminate an expression, unless the expression is definitely incomplete.

For example, the following is ok because "i <=" is definitely incomplete.

```ruby
while i <=
  10 do puts i; i += 1 end
```

Is the following ok?

```ruby
while i
  <= 10 do puts i; i += 1 end
```
The incomplete expression rule does have some pitfalls.

Example: Ruby considers

\[ x = a + b + c \]

to be two expressions: \( x = a + b \) and \( + c \).

Rule of thumb: If breaking an expression across lines, end lines with an operator:

\[ x = a + b + c \]

Alternative: Indicate continuation with a backslash at the end of the line.
Expression or statement?

Academic writing on programming languages commonly uses the term "statement" to denote a syntactic element that performs operation(s) but does not produce a value.

The term "expression" is consistently used to describe a construct that produces a value.

Ruby literature sometimes talks about the "while statement" even though `while` produces a value:

```ruby
i = 1
while i <= 3 do i += 1 end
=> nil
```

Dilemma: Do we call it the "while statement" or the "while expression"?

We'll see later that the `break` construct can cause a `while` loop to produce a value other than `nil`. 
Logical operators

Ruby has operators for conjunction, disjunction, and "not" with the same symbols as Java and C, but with somewhat different semantics.

Conjunction is &&, just like Java, but note the values produced:

```ruby
>> true && false
=> false

>> 1 && 2

>> true && "abc"

>> nil && 1
```

Challenge: Concisely describe the rule that Ruby uses to determine the value of a conjunction operation.

Remember:
Any value that is not false or nil is considered to be "true".

CSC 372 Spring 2018, Ruby Slide 115
Disjunction is `||`, also like Java. As with conjunction, the values produced are interesting:

```ruby
>> 1 || nil
```

```ruby
>> false || 2
```

```ruby
>> "abc" || "xyz"
```

```ruby
>> s = "abc"
>> s[0] || s[3]
```

```ruby
>> s[4] || false
```

Remember:
Any value that is not `false` or `nil` is considered to be "true".
Logical operators, continued

An exclamation mark inverts a logical value. The resulting value is always true or false.

```ruby
>> ! true
```

```ruby
>> ! 1
```

```ruby
>> ! nil
```

```ruby
>> !(1 || 2)
```

```ruby
>> !("abc"[5] || [1,2,3][10])
```

```ruby
>> ![nil]
```

Remember:
Any value that is not false or nil is considered to be "true". 
Logical operators, continued

There are also **and**, **or**, and **not** operators, but with very low precedence.

Why?

We can write this,

\[ x < 2 \text{ \&\& } y > 3 \text{ or } x * y < 10 \text{ || } z > 20 \]

instead of this:

\[ (x < 2 \text{ \&\& } y > 3) \text{ || } (x * y < 10 \text{ || } z > 20) \]

LHtLaL problem: Devise an example for ! vs. **not**.
Here is Ruby's `if-then-else`:

```ruby
>> if 1 < 2 then "three" else [4] end

>> if 10 < 2 then "three" else [4] end

>> if 0 then "three" else [4] end * 3
```

Observations?

Speculate: Is the following valid? If so, what will it produce?
```
if 1 > 2 then 3 end
```
If a language's **if-then-else** returns a value, it creates an issue about the meaning of an **if-then** with no **else**.

In the C family, **if-else** doesn't return a value.

Haskell and ML simply don't allow an **else-less if**.

In Icon, an expression like **if 2 > 3 then 4** is said to **fail**. No value is produced, and failure propagates to any enclosing expression, which in turn fails.

Speculate: How does Ruby handle it?

```ruby
>> if 1 > 2 then 3 end
```

Ruby also provides **1 > 2 ? 3 : 4**, a ternary conditional operator, just like the C family. Is that a good thing or bad thing?
The most common Ruby coding style puts the if, the else, the end, and the expressions of the clauses on separate lines:

```ruby
if lower <= x && x <= higher or inExRange(x, rangeList) then
  puts "x is in range"
  history.add x
else
  outliers.add x
end
```

Note the use of the low-precedence or instead of | |.

The "then" at the end of the first line above is optional.

then is not optional in this one-line expression:

```ruby
if 1 then 2 else 3 end
```
Ruby provides an `elsif` clause for "else-if" situations.

```ruby
if average >= 90 then
  grade = "A"
elsif average >= 80 then
  grade = "B"
elsif average >= 70 then
  grade = "C"
else
  grade = "F"
end
```

Note that there is no "end" to terminate the `then` clauses. `elsif` both closes the current `then` and starts a new clause.

It is not required to have a final `else`.

Is `elsif` syntactic sugar?
At hand:

```ruby
if average >= 90 then
  grade = "A"
elsif average >= 80 then
  grade = "B"
elsif average >= 70 then
  grade = "C"
else
  grade = "F"
end
```

Can we shorten it by thinking less imperatively and more about values?

See 5.1.4 in RPL for Ruby's `case` (a.k.a. "switch") expression.

```ruby
grade =
  if average >= 90 then "A"
  elsif average >= 80 then "B"
  elsif average >= 70 then "C"
  else "F"
  end
```
if and unless as modifiers

if and unless can be used as modifiers to indicate conditional execution.

```ruby
>> total, count = 123.4, 5      # Note: parallel assignment

>> printf("average = %g\n", total / count) if count != 0
average = 24.68
=> nil

>> total, count = 123.4, 0
>> printf("average = %g\n", total / count) unless count == 0
=> nil
```

The general forms are:
- `expr1 if expr2`
- `expr1 unless expr2`

What does 'x.f if x' mean?
Ruby's `break` and `next` are similar to Java's `break` and `continue`.

Below is a loop that reads lines from standard input, terminating on end of file or when a line beginning with a period is read. Each line is printed unless the line begins with a pound sign.

```ruby
while line = gets
  if line[0] == "." then
    break
  end
  if line[0] == "#" then
    next
  end
  puts line
end
```

Problem: Rewrite the above loop to use `if` as a modifier.
Remember that **while** is an expression that by default produces the value **nil** when the loop terminates.

If a while loop is exited with **break expr**, the value of **expr** is the value of the **while**.

Here's a contrived example to show the mechanics of it:

```ruby
% cat break2.rb
s = "x"
puts (while true do
  break s if s.size > 30
  s += s
end)

% ruby break2.rb
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```
Here are three examples of Ruby's `for` loop:

```ruby
for i in 1..100 do # as with while, the do is optional
  sum += i
end

for i in [10,20,30]
  sum += i
end

for msymbol in "x".methods
  puts msymbol if msymbol.to_s.include? "!
end
```

The "in" expression must produce a value that has an `each` method.
- In the first case, the "in" expression is a `Range`.
- In the latter two it is an `Array`. 
The **for** loop supports parallel assignment:

```ruby
for s,n,sep in ["1",5,"-"], ["s",2,"o"], [" <-> ",10,"" ]
  puts [s] * n * sep
end
```

What's the output?

Consider the design choice of supporting parallel assignment in the **for**.
- How would we write the above without it?
- What's the mental footprint of this feature?
- What's the big deal since there's already parallel assignment?
- Is this creeping featurism?
Methods and more
Here is a simple Ruby method:

```ruby
def add x, y
  return x + y
end
```

The keyword `def` indicates that this is a method definition.

Following `def` is the method name.

The parameter list follows, optionally enclosed in parentheses. 
No types can be specified.

Zero or more expressions follow

`end` terminates the definition.
If the end of a method is reached without encountering a `return`, the value of the last expression becomes the return value.

Here is a more idiomatic definition for `add`:

```ruby
def add x, y
  x + y
end
```

Do you prefer the above or the below?

```ruby
def add x, y
  return x + y
end
```

Which is more like Haskell?
As we saw in an early example, if no arguments are required, the parameter list can be omitted:

```ruby
def hello
  puts "Hello!"
end
```

What does `hello` return?
Ruby methods in a file can be tested by hand using **irb**.

```ruby
% cat simple.rb
def add x, y
  x + y
end

% irb -r ./simple.rb # .rb is not required
>> add 3,4
=> 7
```

If the environment variable **RUBYLIB** is set to ".", the "./" can be omitted:

```bash
% echo $RUBYLIB
.
% irb -r simple
...
```

Set it with "**export RUBYLIB=."** in your "/.bashrc".
A source file can be loaded with `Kernel.load`:

```ruby
>> load "simple.rb"
=> true
>> add 3, 4
=> 7
```

How does `load` in Ruby differ from `:load` in `ghci`?

`load "simple.rb"` is simply a Ruby expression that's evaluated by `irb`. Its side-effect is that the specified file is loaded.

How can we take advantage of `load` being a Ruby method?
Because `load` is a method, we can write code that `loads` code.

Here's a method to load `simple.rb`:

```ruby
def r; load "simple.rb"; end  # could be in .irbrc
```

Usage:

```
% irb
>> r

>> add 3,4
=> 7

[...edit `simple.rb` in another window and put a `puts` in `add`...]
>> r

>> add 3,4
add called...
=> 7
```
I claim to be defining methods \texttt{add} and \texttt{hello} but there's no class in sight!

Methods can be added to a class at run-time in Ruby!

A freestanding method found in a file is associated with an object referred to as "\texttt{main}", an instance of \texttt{Object}.

At the top level, the name \texttt{self} references that object.

\begin{verbatim}
>> [self.class, self.to_s] => [Object, "main"]
\end{verbatim}

\begin{verbatim}
>> methods_b4 = self.private_methods
>> load "simple.rb"

>> self.private_methods - methods_b4
=> [:add]
\end{verbatim}

We see that loading \texttt{simple.rb} added the method \texttt{add} to \texttt{main}. 
Where's the class, continued?

We'll later see how to define classes but our initial "mode" on the Ruby assignments will be writing programs in terms of top-level methods.

This is essentially procedural programming with an object-oriented library.
Default values for arguments

Ruby allows default values to be specified for a method's arguments:

```ruby
def wrap s, wrapper = "()"  # wrap3.rb
  wrapper[0] + s + wrapper[-1]  # Why -1?
end

>> wrap "abc", "<>"
=> "<abc>"

>> wrap "abc"
=> "(abc)"

>> wrap it, "|"
=> "|(|abc)|"
```

Lots of library methods use default arguments.

```ruby
>> "a-b c-d".split
=> ["a-b", "c-d"]

>> "a-b c-d".split "-"
=> ["a", "b c", "d"]
```
Ruby does not allow the methods of a class to be overloaded. Here's a Java-like approach that does not work:

```ruby
def wrap s
  wrap(s, "()")
end

def wrap s, wrapper
  wrapper[0] + s + wrapper[-1]
end
```

The imagined behavior is that if `wrap` is called with one argument it will call the two-argument `wrap` with "()" as a second argument. In fact, the second definition of `wrap` simply replaces the first. (Last `def` wins!)

```
>> wrap "x"
ArgumentError: wrong number of arguments (1 for 2)

>> wrap("testing", "[ "]")   => "[testing]"
```
Different languages approach overloading and default arguments in various ways. Here's a sampling:

- **Java**: Overloading; no default arguments
- **Ruby**: No overloading; default arguments
- **C++**: Overloading and default arguments
- **Icon**: No overloading; no default arguments; use an idiom

How does the mental footprint of the four approaches vary? What's the impact on the language's written specification?

Here is `wrap` in Icon:

```icon
procedure wrap(s, wrapper)
    /wrapper := "()" # if wrapper is &null, assign "()" to wrapper
    return wrapper[1] || s || wrapper[-1]
end
```
**Arbitrary number of arguments**

Java's `String.format` and C's `printf` can accept any number of arguments.

This Ruby method accepts any number of arguments and prints them:

```ruby
def showargs(*args)
  puts "#{args.size} arguments"
  for i in 0...args.size do       # Recall a...b is a to b-1
    puts "##{i}: #{args[i]}"
  end
end
```

The rule: If a parameter is prefixed with an asterisk, an array is made of all following arguments.

```
>> showargs(1, "two", 3.0)
3 arguments:
#0: 1
#1: two
#2: 3.0
```
Problem: Write a method `format` that interpolates argument values into a string where percent signs are found.

```ruby
>> format("x = %, y = %, z = %\n", 7, "ten", "zoo")
=> "x = 7, y = ten, z = zoo\n"

>> format "testing\n"
=> "testing\n"
```

Use `to_s` for conversion to `String`.

A common term for this sort of facility is "varargs"—variable number of arguments.
Whole programs
Here's an example of source file layout for a program with several methods:

```ruby
def main
  puts "in main"; f; g
end

def f; puts "in f" end
def g; puts "in g" end

main  # This runs the program
```

A rule: the definition for a method must be seen before it is executed.

The definitions for `f` and `g` can follow the definition of `main` because they aren't executed until `main` is executed.

Could the line "main" appear before the definition of `f`?

Try it: Shuffle around the three definitions and "main" to see what works and what doesn't.
I'd like to load the following file and then test `showline`, but loading it in `irb` seems to hang. Why?

```ruby
% cat main3.rb
def showline s
  puts "Line: #{s.inspect} (#{s.size} chars)"
end
def main
  while line = gets; showline line; end
end
main
```

```bash
% irb
>> load "main3.rb"
...no output or >> prompt after the load...
```
Testing methods when there's a "main", cont.

Here's a technique that lets a program run normally with **ruby** but **not** run **main** when loaded with **irb**:

```ruby
% cat main3a.rb
def showline s
  puts "Line: #{s.inspect} (#{s.size} chars)"
end
def main
  while line = gets; showline line; end
end
main unless $0 == "irb"
```

% **irb**
```
>> load "main3a.rb"
>> showline "testing"
Line: "testing" (7 chars)
>> main
(waits for input)
```

Call **main** unless the name of the program being run is "**irb**".

Now I can test methods by hand in **irb** but still do **ruby** **main3.rb** ...
Variables prefixed with a $ are global, and can be referenced in any method in any file, including top-level code.

```ruby
def f
  puts "f: $x = #{x}" end

def g
  $x = 100
end

$x = 10
f
g

puts "top-level: $x = #{x}"```

The code at left...
1. Sets $x at the top-level.
2. Prints $x in f.
3. Changes $x in g.
4. Prints the final value of $x at the top-level.

Output:
```
f: $x = 10
f: $x = 10
```

`top-level: $x = 100`
Predefined global variables

Ruby has a number of *predefined global variables*:

```ruby
>> puts global_variables.sort * "","
$!, $", $&, $', $*, $+, $., $-0, $-F, $-I, $-K, $-W, $-a, $-d, $-i, $-l, $-p, $-v, $-w, $/, $0, $1, ..., $9, $:, $; $<, $=, $>, $?,$@,$DEBUG,$FILENAME,$KCODE,$LOADED_FEATURES,$LOAD_PATH,$PROGRAM_NAME,$SAFE,$VERBOSE,$\$, $`, $binding,$stderr,$stdin,$stdout,$~
```

A few:

- `$0`  Program name  (>> $0 => "irb2.2")
- `$,` Used by `print` and `Array.join`.  (>> $, = ";"; [1,2].join=> "1-2")
- `$:`  Directories where `load` and `require` look for files.  (Try it!)
- `$_`  Last string read by `Kernel.gets` and `Kernel.readline`.  (Try it!)

More: The *Global Variables* section of *The Top-Level Environment* in RPL, chapter 10.
Wikipedia says,

"In computer programming, a *sigil* is a symbol attached to a variable name, showing the variable's datatype or scope, usually a prefix, as in $foo, where $ is the sigil.

"Sigil, from the Latin sigillum, meaning a 'little sign', means a sign or image supposedly having magical power."

"In 1999 Philip Gwyn adopted the term 'to mean the funny character at the front of a Perl variable'."

Are *reverse!* and *nan*? examples of sigils?

Lots more: [https://en.wikipedia.org/wiki/Sigil_(computer_programming)]
Ordinary variables are local to the method in which they're created.

```ruby
% cat scope1.rb
x = 10

puts "top-level: x = #{x}"

def f
    puts "in f: x = #{x}"
end

f

% ruby scope1.rb
top-level: x = 10
scope1.rb:6:in 'f': undefined local variable or method 'x'
```
A rule in Ruby is that if an identifier begins with a capital letter, it represents a constant.

The first assignment to a constant is considered initialization.

```ruby
>> MAX_ITEMS = 100

Assigning to an already initialized constant is permitted but a warning is generated.

```ruby
>> MAX_ITEMS = 200
(irb):4: warning: already initialized constant MAX_ITEMS
=> 200
```ruby

Modifying an object referenced by a constant does not produce a warning:

```ruby
>> L = [10,20]
=> [10, 20]

>> L.push 30
=> [10, 20, 30]
```
Like globals, constants can be accessed in all methods.

```ruby
% cat constant1.rb
MAX_LEVELS = 1000

def f
  puts "f: max levels = '#{MAX_LEVELS}'"
end

f

puts "top-level: max levels = '#{MAX_LEVELS}'"

% ruby constant1.rb
f: max levels = 1000
top-level: max levels = 1000
```

Find out: Can a constant be created in a method?
Pitfall: If a method is given a name that begins with a capital letter, it compiles ok but it can't be run!

```ruby
>> def Hello; puts "hello!" end

>> Hello
NameError: uninitialized constant Hello
```
There are a number of predefined constants. Here are a few:

**RUBY_VERSION**

The version of Ruby that's running.

**ARGV**

An array holding the command line arguments, like the argument to `main` in a Java program.

**ENV**

An object holding the "environment variables" (shown with `env` on UNIX machines and `set` on Windows machines.)

**STDIN, STDOUT**

Instances of the `IO` class representing standard input and standard output (the keyboard and screen, by default).