Introduction

What is Ruby?

Experimenting with Ruby using irb

Executing Ruby code in a file

Everything is an object

Variables have no type

Ruby's philosophy is often "Why not?"

What is Ruby?

"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".

Ruby was invented by Yukihiro Matsumoto ("Matz"), a "Japanese amateur language designer" (his own words). Here is a second-hand summary 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."
> http://www.rubygarden.org/faq/entry/show/5

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 and try Ruby. I'm sure you'll find out for yourself."

What is Ruby?

Ruby is a language in flux.

- Version 1.8.4 is installed on lectura. Version 1.9 is available.
- There is no written standard for Ruby. The language is effectively defined by MRI—Matz' Ruby Implementation.

There is good on-line documentation:

- The first edition of our text, the "pickaxe book", is available for free: www.ruby-doc.org/docs/ProgrammingRuby
- Documentation for the core classes is at www.ruby-doc.org/core.
- The above and more is collected at www.ruby-lang.org/en/documentation.
- Chapter 22 of the text is a fairly concise language reference for Ruby. It will soon be available through the UA library as an E-Reserve. Watch for a link on our website.

What is Ruby?

Ruby is getting a lot of attention and press at the moment. Two popular topics:

- Ruby on Rails, a web application framework.
- JRuby, a 100% pure-Java implementation of Ruby. With JRuby, among other things, you can use Java classes in Ruby programs. (jruby.codehaus.org)

Experimenting with Ruby using irb

There are two common ways to execute Ruby code: (1) Put it in file and execute the file with the "ruby" command. (2) Use irb, the Interactive Ruby Shell. We'll start with irb.

NOTE: The examples on these slides assume a particular configuration for irb, so before you run irb the first time, copy our .irbrc file into your home directory on lectura:

```
cp /home/cs372/fall06/ruby/.irbrc ~
```

irb's default prompt contains a little more information than we need at the moment, so we'll ask for the "simple" prompt:

```
% irb --prompt simple
>>
```

To exit irb, use control-D.

irb, continued

irb evaluates expressions as are they typed.

```
>> 1 + 2
=> 3

>> "testing" + "123"          (NOTE: No trailing semicolon!)
=> "testing123"
```

One of the definitions in our .irbrc allows the last result to be referenced with "it":

```
>> it
=> "testing123"

>> it+it
=> "testing123testing123"
```

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

```
>> 1.23 +
?> 1e5
=> 100001.23
```


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:

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

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

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

Executing Ruby code in a file, continued

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

Here is a program that reads lines from standard input and writes them, with a line number, to standard output:

```
% cat numlines.rb
line_num = 1

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 already seen.

Everything is an object

In Ruby, *every* value is an object.

Methods can be invoked using the form *value.method(parameters...)*.

```
>> "testing".index("i")        # Where's the first "i"?
=> 4

>> "testing".count("t")        # How many times does "t" appear?
=> 2

>> "testing".slice(1,3)
=> "est"

>> "testing".length()
=> 7
```

Repeat: In Ruby, every value is an object.

What are some values in Java that are not objects?

Everything is an object, continued

Parentheses can be omitted from an argument list:

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

>> "testing".slice 1,3
=> "est"

>> puts "number",3
number
3
=> nil

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

If no parameters are required, the parameter list can be omitted.

```
>> "testing".length
=> 7
```


Everything is an object, continued

Of course, "everything" includes numbers:

```
>> 7.class
=> Fixnum

>> 1.2.class
=> Float

>> (3-4).abs
=> 1

>> 17**50
=>33300140732146818380750772381422989832214186835186851059977249

>> it.succ
=>33300140732146818380750772381422989832214186835186851059977250

>> it.class
=> Bignum
```

Everything is an object, continued

The TAB key can be used to show completions:

```
>> 100.<TAB>
                          100.nil?
100.__id__                100.nonzero?
100.__send__              100.numerator
100.abs                   100.object_id
100.between?              100.power!
100.ceil                  100.prec
100.chr                   100.prec_f
100.class                 100.prec_i
100.clone                 100.private_methods
100.coerce                100.protected_methods
100.denominator           100.public_methods
100.display               100.quo
100.div                   100.rdiv
100.divmod                100.remainder
100.downto                100.require
```


Variables have no type

In Java, variables are declared to have a type. When a program is compiled, the compiler ensures that all operations are valid with respect to the types involved.

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

```
>> x = 10
=> 10

>> x = "ten"
=> "ten"

>> x.class
=> String

>> x = x.length
=> 3
```

Here's another way to think about this: Every variable can hold a reference to an object. Because every value is an object, any variable can hold any value.

Variables have no type, continued

It is often said that Java uses *static typing*. Ruby, like most scripting languages, uses *dynamic typing*.

Sometimes the term *strong typing* is used to characterize languages like Java and *weak typing* is used to characterize languages like Ruby but those terms are now often debated and perhaps best avoided.

Another way to describe a language's type-checking mechanism is based on *when* the checking is done. Java uses *compile-time type checking*. Ruby uses *run-time type checking*.

Some statically-typed languages do some type checking at run-time. An example of a run-time type error is Java's ClassCastException. C does absolutely no type-checking at run-time. Ruby does absolutely no type-checking at compile-time.

What is ML's type-checking approach?

Variables have no type, continued

In a statically typed language a number of constraints can be checked at compile time. For example, all of the following can be verified when a Java program is compiled:

x.getValue()	*x must have a getValue method*
x * y	*x and y must be of compatible types for **
x.f(1,2,3)	*x.f must accept three integer parameters*

In contrast, a typical compiler for a dynamically typed language will attempt to verify none of the above when the code is compiled. If x doesn't have a getValue method, or x and y can't be multiplied, or x.f requires three strings instead of three integers, there will be no warning of that until the program is run.

For years it has been widely held in industry that static typing is a must for reliable systems but a shift in thinking is underway. It is increasingly believed that good test coverage can produce equally reliable software.[1]

[1] http://www.artima.com/weblogs/viewpost.jsp?thread=4639

Ruby's philosophy is often "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?"

The instructor sees Ruby's philosophy as often being "Why not?" Here are some examples, involving overloaded operators:

```
>> [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]
=> [15, 7]

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

>> "decimal: %d, octal: %o, hex: %x" % [20, 20, 20]
=> "decimal: 20, octal: 24, hex: 14"
```


Building blocks

- nil
- Strings
- Numbers
- Conversions
- Arrays

The value nil

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

```
>> nil
=> nil

>> nil.class
=> NilClass

>> nil.object_id
=> 4
```

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

Speculate: Do uninitialized variables have the value nil?

Strings

Instances of Ruby's String class are used to represent character strings.

One way to specify a literal string is with double quotes. A variety of "escapes" are recognized:

```
>> "formfeed \f, newline \n, return \r, tab \t"
=> "formfeed \f, newline \n, return \r, tab \t"

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

> puts "newline >\n<, return (\r), tab >\t<"
newline >
), tab >        <
=> nil

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

Page 321 in the text has a full list of escapes.

Strings, continued

A string literal may be constructed using apostrophes instead of double quotes. If so, only \' and \\ are recognized as escapes:

```
>> '\n\t'.length      Four characters: backslash, n, backslash, t
=> 4

>> '\'\\'             Two characters: apostrophe, backslash
=> "'\\"

>> it.length
=> 2
```

A "here document" provides a third way to specify a string:

```
>> s = <<SomethingUnique
just
   testing
SomethingUnique
=> "just    \n    testing\n"
```

There's a fourth way, too: %q{ just testin' this } How many ways to do something is too many? Which are syntactic sugar?

Strings, continued

The public_methods (and methods) method show the public methods that are available for an object. Here are some of the methods for String:

```
>> "abc".public_methods.sort
=> ["%", "*", "+", "<", "<<", "<=", "<=>", "==", "===", "=~", ">", ">=", "[]", "[]=", "__id__",
"__send__", "all?", "any?", "between?", "capitalize", "capitalize!", "casecmp", "center",
"chomp", "chomp!", "chop", "chop!", "class", "clone", "collect", "concat", "count",
"crypt", "delete", "delete!", "detect", "display", "downcase", "downcase!", "dump",
"dup",  "each", "each_byte", "each_line", "each_with_index", "empty?", "entries",
"eql?", "equal?", "extend", "find", "find_all", "freeze", "frozen?", "gem", "grep", "gsub",
"gsub!", "hash", "hex", "id", "include?", "index", "inject", "insert", "inspect",
"instance_eval", "instance_of?", "instance_variable_get", "instance_variable_set",
"instance_variables", "intern", "is_a?", "kind_of?", "length", "ljust", "lstrip", "lstrip!",
"map", "match", "max", "member?", "method", "methods","min", "next", "next!", "nil?",
"object_id", "oct", "partition", "private_methods", "protected_methods",
"public_methods", "reject", "replace", "require", "require_gem", "respond_to?",
"reverse", "reverse!", "rindex", "rjust", "rstrip", "rstrip!", "scan", "select", "send",
...

>> "abc".public_methods.length
=> 145
```


Strings, continued

Unlike Java, ML, 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:

```
>> x = "testing"
=> "testing"

>> y = x              x and y now reference the same instance of String.
=> "testing"

>> x.upcase!          Convention: If there are both applicative and imperative forms of a
                      method, the name of the imperative form ends with an exclamation.
=> "TESTING"

>> y
=> "TESTING"
```

In Java, if s1 and s2 are Strings an assignment such as s1 = s2 produces a shared reference but it's never an issue because instances of String are immutable—no methods change a String.

Strings, continued

The dup method produces a copy of a string.

```
>> y = x.dup
=> "TESTING"

>> y.downcase!
=> "testing"

>> y
=> "testing"

>> x
=> "TESTING"
```

Some objects that hold strings make a copy of the string when the string is added to the object.

Strings, continued

Strings can be compared with a typical set of operators:

```
>> s1 = "apple"
=> "apple"

>> s2 = "testing"
=> "testing"

>> s1 == s2
=> false

>> s1 != s2
=> true

>> s1 < s2
=> true

>> s1 >= s2
=> false
```


Strings, continued

There is also a *comparison operator*. It produces -1, 0, or 1 depending on whether the first operand is less than, equal to, or greater than the second operand.

```
>> "apple" <=> "testing"
=> -1

>> "testing" <=> "apple"
=> 1

>> "x" <=> "x"
=> 0
```


Strings, continued

A individual character can be extracted from a string but note that the result is an integer character code (an instance of Fixnum), **not** a one-character string:

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

>> s[0]
=> 97            # 97 is the ASCII code for 'a'

>> s[1]
=> 98

>> s[-1]         # -1 is the last character, -2 is next to last, etc.
=> 99

>> s[100]        # Why not produce 0 for an out of bounds reference?
=> nil
```

Note that the position is zero-based. A negative value indicates an offset from the end of the string.

What's a good reason that Java provides s.charAt(n) instead of allowing s[n]?

Strings, continued

A subscripted string can be the target of an assignment.

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

>> s[0] = 65
=> 65

>> s[1] = "tomi"
=> "tomi"

>> s
=> "Atomic"
```

The numeric code for a character can be obtained by preceding the character with a question mark:

```
>> s[0] = ?B
=> 66
>> s
=> "Btomic"
```

Strings, continued

A substring can be referenced in several ways.

```
>> s = "replace"
=> "replace"

>> s[2,3]
=> "pla"

>> s[2,1]        Remember that s[n] yields a number, not a string.
=> "p"

>> s[2..-1]      2..-1 creates a Range object.  (More on ranges later.)
=> "place"

>> s[10,10]
=> nil

>> s[-4,3]
=> "lac"
```

Speculate: What does s[1,100] produce? How about s[-1,-3]?

Strings, continued

A substring can be the target of assignment:

```
>> s = "replace"
=> "replace"

>> s[0,2] = ""
=> ""

>> s
=> "place"

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

>> s["aria"] = "kton"      If "aria" appears, replace it (error if not).
=> "kton"

>> s
=> "plankton"
```


Strings, continued

In a string literal enclosed with double quotes, or specified with a here document, the sequence #{*expr*} causes interpolation of *expr*, an arbitrary Ruby expression.

```
>> x = 10
=> 10

>> y = "twenty"
=> "twenty"

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

>> s = "String methods: #{"abc".methods}".length
=> 896
```

The << operator appends to a string and produces the new string. *The string is changed.*

```
>> s = "just"
=> "just"
>> s << "testing" << "this"
=> "justtestingthis"
```


Numbers

On lectura, integers in the range -2^{30} to $2^{30}-1$ 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**30-1      The exponentiation operator is **.
=> 1073741823

>> x.class
=> Fixnum

>> y = x + 1
=> 1073741824

>> y.class
=> Bignum

>> z = y - 1
=> 1073741823

>> z.class
=> Fixnum
```

How can we see what methods are available for instances of Fixnum?

Numbers, continued

The Float class represents floating point numbers that can be represented by a double-precision floating point number on the host architecture.

```
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.1110755554987

>> x * 2e-3
=> 0.246912

>> x = x / 0.0
=>  Infinity

>> (0.0/0.0).nan?
=> true
```


Numbers, continued

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

```
>> 10 / 5.1
=> 1.96078431372549

>> 10 % 4.5
=> 1.0

>> 2**40 / 8.0
=> 137438953472.0

>> it.class
=> Float
```

Other numeric classes in Ruby include BigDecimal, Complex, Rational and Matrix.

Conversions

Unlike many scripting 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"
=> "1020"

>> 10 + "20".to_f
=> 30.0

>> 10 + 20.9.to_i
=> 30

>> 2**100.to_f
=>1.26765060022823e+030
```

Speculate: What does "123xyz".to_i produce?

Arrays

An ordered sequence of values is typically represented in Ruby by an instance of Array.

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

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

>> a2 = ["ten", 20, 30.0, 10**40]
=> ["ten", 20, 30.0, 10000000000000000000000000000000000000000]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 10000000000000000000000000000000000000000],
[[[10, 20, 30]]]]
```

What's a difference between a Ruby array and an ML list?

Arrays, continued

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

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

>> a[0]
=> 1

>> a[1,2]
=> ["two", 3.0]

>> a[-1][-2]
=> "c"

>> a[-1][-2][0]
=> 99
```

Note that %w{ ... } provides a way to avoid the tedium of surrounding each string with quotes. Experiment with it!

Arrays, continued

Elements and subarrays can be assigned to. Ruby accommodates a variety of cases; here are some:

```
>> a = [10, 20, 30, 40, 50, 60]
=> [10, 20, 30, 40, 50, 60]

>> a[1] = "twenty"; a      Note: Semicolon separates expressions; a's value is shown.
=> [10, "twenty", 30, 40, 50, 60]

>> a[2..4] = %w{a b c d e}; a
=> [10, "twenty", "a", "b", "c", "d", "e", 60]

>> a[1..-1] = [ ]; a
=> [10]

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

>> a[10] = [5,6]; a
=> [[1, 2, 3], nil, nil, nil, nil, nil, nil, nil, nil, nil, [5, 6]]
```


Arrays, continued

A variety of operations are provided for arrays. Here's a small sample:

```
>> a = [ ]
=> [ ]

>> a << 1; a
=> [1]

>> a << [2,3,4]; a
=> [1, [2, 3, 4]]

>> a.reverse!; a
=> [[2, 3, 4], 1]

>> a[0].shift
=> 2

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

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

>> a + b
=> [1, 2, 3, 4, 1, 3, 5]

>> a - b
=> [2, 4]

>> a & b
=> [1, 3]

>> a | b
=> [1, 2, 3, 4, 5]

>> a == (a | b)[0..3]
=> true
```

Control structures

The while loop

Sidebar: Source code layout

Expressions or statements?

Logical operators

if-then-else

if and unless as modifiers

break and next

The for loop

The while loop

Here is a loop to print the numbers from 1 through 10, one per line.

```
i = 1
while i <= 10
   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.

The control expression can be optionally followed by do or a colon. Example:

```
while i <= 10 do          # Or, while i <= 10 :
   puts i
   i += 1
end
```

What's a minor problem with Ruby's syntax versus Java's use of braces to bracket multi-line loop bodies?

while, continued

In Java, control structures like 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 an integer value that is non-zero to be "true".

In Ruby, any value that is not false or nil is considered to be "true".

Consider this loop, which reads lines from standard input using gets.

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

Each call to gets returns a string that is the next line of the file. The string is assigned to line and like Java, assignment produces the value assigned. If the first line of the file 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 and "one" is printed on standard output.

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

while, continued

On UNIX machines the string returned by gets has a trailing newline. The chomp method of String can be used to remove it.

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

```
result = ""

while line = gets.chomp
   result += line
end

puts result
```

It doesn't work. What's wrong with it?

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.[1]

[1] I bet some of you get this wrong the first time, like I did!

Sidebar: Source code layout

Unlike Java, Ruby does pay some attention to the presence of newlines in source code. For example, a while loop cannot be naively compressed to a single line. This does not work:

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

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

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

There is some middle ground, too:

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


Source code layout, continued

Ruby considers a newline to terminate an expression, unless the expression is definitely incomplete. Examples:

```
while i <=                          # OK because "i <=" is definitely incomplete
10 do puts i; i += 1 end

while i                          # NOT OK.  "while i" is complete, but then "<= 10"
<= 10 do puts i; i += 1 end      # is flagged as a syntax error.
```

There is a pitfall related to this rule. For 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, put an operator at the end of the line:

```
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 an operation but does not produce a value. The term "expression" is consistently used to describe an operation that produces a value.

Ruby literature, including the text, sometimes talks about the "while statement" even though while produces a value:

```
>> i = 1
=> 1

>> a = (while i <= 3 do i += 1 end)
=> nil
```

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

The text sometimes uses the term "while loop" instead.

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, but with somewhat different semantics.

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

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

>> 1 && 2
=> 2

>> true && "abc"
=> "abc"

>> true && false
=> false

>> true && nil
=> nil
```

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

Logical operators, continued

Disjunction is ||, just like Java. As with conjunction, the values produced are interesting:

```
>> 1 || nil
=> 1

>> false || 2
=> 2

>> "abc" || "xyz"
=> "abc"

>> s = "abc"
=> "abc"

>> s[0] || s[3]
=> 97

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


Logical operators, continued

Just like Java, an exclamation mark inverts a logical value. The resulting value is true or false.

```
>> ! true
=> false

>> ! 1
=> false

>> ! nil
=> true

>> ! (1 || 2)
=> false

>> ! ("abc"[5] || [1,2,3][10])
=> true

>> ![nil]
=> false
```

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

The if-then-else construct

Ruby's if-then-else looks familiar:

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

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

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

What can we say about it?

Speculate: What will 'if 1 > 2 then 3 end' produce?

if-then-else, continued

If there's no else clause and the control expression is false, nil is produced:

```
>> if 1 > 2 then 3 end
=> nil
```

If a language provides for if-then-else to return a value it raises the issue of what if-then means.

- In the C family, if-then-else doesn't return a value.
- ML simply doesn't allow an else-less if.
- In Icon, an expression like if > 2 then 3 is said to *fail*. No value is produced and that failure propagates to any enclosing expression, which in turn fails.

Ruby also provides 1 > 2 ? 3 : 4, a ternary conditional operator, just like the C family. Is that a good thing or bad thing?

if-then-else, continued

The most common Ruby coding style puts the if, the else, the end, and the expressions of the clauses on separate lines:

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

Speculate: Ruby has both || and or for disjunction. Why was or used above?

The elsif clause

Ruby provides an elsif clause for "else-if" situations.

```
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.

How could the code above be improved?

Is elsif syntactic sugar?

if and unless as modifiers

Conditional execution can be indicated by using if and unless as modifiers.

```
>> total, count = 123.4, 5
=> [123.4, 5]

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

>> total, count = 123.4, 0
=> [123.4, 0]

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

The general forms are:

expression1 if *expression2*
expression1 unless *expression2*

What does 'x.f if x' mean?

break and next

The break and next expressions are similar to break and continue in Java.

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.

```
while line = gets

   if line[0] == ?. then
      break
   end

   if line[0] == ?# then next end

   puts line
end
```

Recall: (1) If s is a string, s[0] produces an integer. (2) The construct ?c produces the integer code of the character c.

Problem: Rewrite it to use if as a modifier.

break and next, continued

If an expression is specified with break, the value of the expression becomes the value of the while:

```
% cat break2.rb
s = "x"

puts (while true do
   break s if s.size > 30
   s += s
end)

% ruby break2.rb
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
%
```

Two other control flow modifiers used with loops are redo and retry.

The for loop

Here are three simple examples of Ruby's for loop:

```
for i in 1..100 do
   sum += i
end

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

for method_name in "x".methods do
   puts method_name if method_name.include? "!"
end
```

The "in" expression must be an object that has an each method. In the first case, the "in" expression is a Range. In the latter two it is an Array.

break and next have the same meaning as in a while loop.

The for loop, continued

The for loop supports parallel assignment:

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

Output:

```
1-1-1-1-1
sos
 <-> <-> <-> <-> <-> <-> <-> <-> <-> <->
```

Is it good or bad that the for loop specifically supports parallel assignment? How inconvenient would it be to do without it?

Of course, while, for, if-then-else and other statements can be arbitrarily interleaved and nested, just like in most languages.