Where are we?

In 8 weeks, we've picked up enough Haskell, Scheme, and Ruby to talk intelligently about modern, general-purpose PLs; and as a bonus, enough CLP($\mathcal{R}$) to talk about logic and constraint programming.

- Congratulations to us!

Now we need to:

- Consider OO semantics carefully
- Consider various OO extensions and design decisions
- Consider OO type systems as carefully as we did FP type systems
- Compare OO and FP, specifically extensibility and polymorphism

Next up: Ruby look-up rules, a lower-level view of dynamic dispatch

Look-up rules

How we *resolve* various "symbols" is a key part of language definition.

- In many ways, FP boils down to first-class functions, *lexical scope*, and immutability.

In Ruby, we *syntactically distinguish* instance fields (`@x`) and class fields (`@@x`) from method/block variables (`x`) and method names (`x`).

- Unlike Java, local variables can shadow method names (does `m+2` read a variable or call a method)
  - Rather clumsy since variables aren't declared.
  - Will ignore confusion today; see book for the rules.

“First-class”

If something can be computed, stored in fields, passed as arguments, returned as results, etc., we say it is “first-class”.

All objects in Ruby are first-class (and most things are objects).

These things are not:

- Message names: Must write `if b then x.m else x.n end`, not `x.(if b then m else n end)`
- Blocks (hence conversion to `Proc` instances)
- Argument lists

Variable lookup

To *resolve* a variable (e.g., `x`):

- Inside a code block that defines `x` (`{|x| e}`), `x` resolves to the local variable of the block (i.e., the argument).
- Else inside a code block, `x` resolves to an `x` that is defined in the enclosing method.
  - Lexical scope, just like in Haskell and Scheme
  - Ruby implementation must build closures (those pairs of code and environment you built for the Scheme interpreter)

Nothing really new here

Message lookup

To resolve a message (e.g., `m`):

- A message is sent to an object (e.g., `e.m`), so first evalaute expression `e` to an object `obj`.
- Get the class of `obj` (e.g., `A`) (every object has a class).
- If `m` is defined in `A` (check instance methods first, then class methods), invoke that method, else recur with superclass of `A`.
  - Will slightly complicate this story later with mixins

What about `self`?

As always, evaluation takes place in an environment.

In every environment, `self` is always bound to some object. (This determines message resolution for `self` and `super`.)

Key principles of OOP:

- Inheritance and override (last slide)
- Private fields
- *The semantics of message send*

To send `m` to `obj` means evaluate the body of the method `m` in an environment with argument names mapped to actual arguments *and `self` bound to `obj`*.

That last phrase is exactly what "late-binding", "dynamic dispatch", and "virtual function call" mean. It is why code defined in superclasses can invoke code defined in subclasses.

A Simple Example, part 1

```
ev m = even m
  where
    even n =
      if n==0
        then True
        else odd (n-1)
    odd n =
      if n==0
        then False
        else even (n-1)

-- neither of these changes the behavior of odd
even x = (x mod 2) == 0
even x = False
```

A Simple Example, part 2

```
class A
  def even x
     if x=0 then true else odd(x-1) end
  end
  def odd x
     if x=0 then false else even(x-1) end
  end
end
class B < A
  def even x # changes B's odd too!
     x % 2 = 0
  end
end
```

Some Perspective on Late-Binding

Some opinions:

- Late-binding makes a more complicated semantics
  - Ruby without `self` is easier to define and reason about
  - It takes months in 142 to get to where we can explain it
  - It makes it harder to reason about programs
- But late-binding is often an elegant pattern for reuse
  - OO without `self` is not OO
  - Late-binding fits well with the "object analogy"
  - Late-binding can make it easier to localize specialized code even when other code wasn't expecting specialization
    - More reuse/abuse.

A Lower-Level View

Ruby clearly encourages late-binding with its message-send semantics.

But a definition in one language is often a pattern in another...

We can simulate late-binding in Scheme easily enough

And sketch how compilers/interpreters implement objects

- A naive but *accurate* view of implementation can give an alternate way to reason about programs

The Key Idea

The key to implementing late-binding is extending all the methods to take an extra argument (for `self`).

So an object is implemented as a record holding methods and fields, where methods are passed self explicitly.

And message-resolution always uses `self`.

What about classes and performance?

This approach, while a fine pattern, has some problems:

- It doesn't model Ruby, where methods can be added/removed from classes dynamically and an object's class determines behavior.
- It is space-inefficient: all objects of a class have the same methods.
- It is time-inefficient: message-send should be constant-time, not list traversals.

We fix the first two by adding a level of indirection: all instances made from same "constructor" share list of methods, and we can mutate the list.

We fix the third with better data structures (array or hash) and various tricks (so subclassing works).

Static typing gets in the way

- We have seen late-binding as a Scheme pattern
- In reality, we have learned roughly how OO implementations do it, without appealing to assembly code (where it really happens)
- Using Haskell instead of Scheme would have been a pain:
  - The Haskell type system is “unfriendly” for `self`.
  - We would have roughly taken the “embed Scheme in Haskell” approach, giving *every* object the same Haskell type.
  - But to be fair, basic OO typing (coming soon) is “unfriendly” to Haskell datatypes, first-class functions, and parametric polymorphism.

Multiple Inheritance and Related Ideas

Have seen OO's essence: inheritance, overriding, dynamic-dispatch.

What if we want these things from more than "exactly 1 superclass"?

- *Multiple inheritance*: allow > 1 superclasses
  - Useful but has some problems (see Python, C++)
- Java-style *interfaces*: allow > 1 *types*
  - "Irrelevant" in a dynamically typed language, but fewer problems
- *Mixins*: allow > 1 "sources of methods"
  - Close to multiple inheritance; almost as useful with fewer (?) problems
  - In Ruby

Multiple Inheritance

If code reuse via inheritance is so useful, why not allow multiple superclasses?

- Because it causes some semantic awkwardness and implementation awkwardness (we'll discuss only the former)
- (With static typing, there are some more issues)

Is it useful? Sure: A simple example is "3DColorPoint" assuming we already have "3DPoint" and "ColorPoint".

Naive view: Subclass has all fields and methods of all superclasses

Trees, dags, and diamonds

The “class hierarchy” is a (conceptual) graph with edges from subclasses to superclasses.

Ambiguous phrase: *subclass*, let’s use *immediate-subclass* or *transitive-subclass* when we need to be clear.

- With single inheritance, the class hierarchy is a tree.
- With multiple inheritance, the class hierarchy is a dag.
  - Semantic problems arise from *diamonds*: Multiple ways to show that class `A` is a transitive-subclass of some class `B`.
  - If all classes are transitive-subclasses of something like `Object`, then multiple inheritance always leads to diamonds.

Multiple Inheritance Semantic Problems

What if multiple superclasses define the same message $\boldsymbol{m}$ or field $\boldsymbol{f}$?

- Classic example: Artists, Cowboys, and ArtistCowboys

Options for $\boldsymbol{m}$:

- Reject subclass—too restrictive (especially due to diamonds)
- "Left-most superclass wins" (leads to silent weirdness and really want per-method flexiblity)
- Require subclass to override $\boldsymbol{m}$ (can use *directed resends*)

Options for $\boldsymbol{f}$: one copy or two copies?

C++ provides two forms of inheritance:

- One always makes two copies
- One makes one copy *if* fields were declared by same class

Java-style interfaces

(Recall?) in Java, we can define *interfaces* and classes can *implement* them.

- Interface describes methods and their types

```
interface Example {
    void m1(int x, int y);
    Foo m2(Example e, String s);
}
```

- `Example` is a type (can be used for a field, method argument, local variable, etc.)
- If class `C` implements interface `I`, then instances of `C` can have type `I` but `C` must define everything in `I` (directly or via inheritance).
- Given an expression of type `I`, it type-checks to send it any message `I` promises.

Interfaces are a typing thing

In Java, you have 1 immediate-superclass and any number of interfaces you implement.

Because interfaces provide no methods or fields (only types of methods), no duplication problems result!

- No problem if `I1` and `I2` both "promise" some method `m` and `C` implements `I1` and `I2`.

But interfaces do not give us the power we want for making colored 3D points or artist-cowboys.

They're *totally irrelevant* in a dynamically typed language like Ruby:

- We are already allowed to send any message to any object
- It is up to us to get it right ("interfaces" more in comments or *reflection*, e.g., the `methods` method of `Object`)

Interfaces vs. Abstract Classes

If you had multiple inheritance, you could replace interfaces with abstract classes containing only abstract methods.

- Called pure virtual methods in C++
- But the whole point is multiple inheritance is more powerful because it doesn't require $n - 1$ superclasses to have only abstract methods.

Mixins

A mixin is a collection of methods

- no fields, constructors, instances, etc.

Languages with mixins (e.g., Ruby) typically allow a class to have 1 superclass but any number of mixins.

Bad news: Less powerful than multiple inheritance; have to decide upfront what is a class and what is a mixin.

Good news: Clear semantics on methods/fields and works great for certain idioms.

Ruby mixin basics

A *module*'s instance methods are mixed into a class by *including* the module in the class definition.

Method-lookup rules: First class's methods, then its mixins' methods, (later includes shadow), then immediate-superclass, then immediate-superclass's mixins, . . .

Field rules: It is all one object.

What mixins are good for

We could make `Color` a mixin and then use it for coloring 2D and 3D points.

- Works fine but often bad style to have mixin methods define fields (could conflict with other fields)

For artist-cowboys, what should the mixin be?

But mixins are elegant for letting classes "get a bunch of methods while defining only a few".

- All thanks to late-binding!
- Cool examples in Ruby library: `Comparable` and `Enumerable`