CSE 341
Programming Languages

Dynamic Dispatch vs. Closures
OOP vs. Functional Decomposition
Wrapping Up

Zach Tatlock

Spring 2014

Dynamic dispatch

Dynamic dispatch

- Also known as *late binding* or *virtual methods*
- Call `self.m2()` in method `m1` defined in class `C` can *resolve to* a method `m2` defined in a subclass of `C`
- Most unique characteristic of OOP

Need to define the semantics of *method lookup* as carefully as we defined *variable lookup* for our PLs

Review: variable lookup

Rules for "looking things up" is a key part of PL semantics

- ML: Look up *variables* in the appropriate environment
  - Lexical scope for closures
  - *Field names* (for records) are different: not variables
- Racket: Like ML plus **`let`**, **`letrec`**
- Ruby:
  - Local variables and blocks mostly like ML and Racket
  - But also have instance variables, class variables, methods (all more like record fields)
    - Look up in terms of **`self`**, which is special

Using **`self`**

- **`self`** maps to some “current” object
- Look up instance variable **`@x`** using object bound to **`self`**
- Look up class variables **`@@x`** using object bound to **`self.class`**
- Look up methods…

Ruby method lookup

The semantics for method calls also known as message sends

`e0.m(e1,…,en)`

1. Evaluate **`e0`**, **`e1`**, …, **`en`** to objects **`obj0`**, **`obj1`**, …, **`objn`**
   - As usual, may involve looking up **`self`**, variables, fields, etc.
2. Let **`C`** be the class of **`obj0`** (every object has a class)
3. If **`m`** is defined in **`C`**, pick that method, else recur with the superclass of **`C`** unless **`C`** is already **`Object`**
   - If no **`m`** is found, call **`method_missing`** instead
     - Definition of **`method_missing`** in **`Object`** raises an error
4. Evaluate body of method picked:
   - With formal arguments bound to **`obj1`**, …, **`objn`**
   - With **`self`** bound to **`obj0`** -- this implements dynamic dispatch!

Punch-line again

`e0.m(e1,…,en)`

To implement dynamic dispatch, evaluate the method body with **`self`** mapping to the *receiver* (result of **`e0`**)

- That way, any **`self`** calls in body of **`m`** use the receiver's class,
  - Not necessarily the class that defined **`m`**

- This much is the same in Ruby, Java, C#, Smalltalk, etc.

Comments on dynamic dispatch

- This is why **`distFromOrigin2`** worked in **`PolarPoint`**

- More complicated than the rules for closures
  - Have to treat **`self`** specially
  - May seem simpler only if you learned it first
  - Complicated does not necessarily mean inferior or superior

Static overloading

In Java/C#/C++, method-lookup rules are similar, but more complicated because > 1 methods in a class can have same name

- Java/C/C++: Overriding only when number/types of arguments the same
- Ruby: same-method-name always overriding

Pick the "best one" using the *static* (!) types of the arguments

- Complicated rules for "best"
- Type-checking error if there is no "best"

Relies fundamentally on type-checking rules

- Ruby has none

A simple example, part 1

In ML (and other languages), closures are closed

```
fun even x = if x=0 then true  else odd  (x-1)
and odd  x = if x=0 then false else even (x-1)
```

So we can shadow **odd**, but any call to the closure bound to **odd** above will "do what we expect"

- Does not matter if we shadow **even** or not

```
(* does not change odd - too bad; this would
   improve it *)
fun even x = (x mod 2)=0
```

```
(* does not change odd - good thing; this would
   break it *)
fun even x = false
```

A simple example, part 2

In Ruby (and other OOP languages), subclasses can change the behavior of methods they do not override

```
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  # improves odd in B objects
  def even x ; x % 2 == 0 end
end
class C < A  # breaks odd in C objects
  def even x ; false end
end
```

The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in subclasses even if it is not overridden

- Maybe on purpose, maybe by mistake
- Observable behavior includes calls-to-overridable methods

- So *harder* to reason about “the code you're looking at”
  - Can avoid by disallowing overriding
    - “private” or “final” methods

- So *easier* for subclasses to affect behavior without copying code
  - Provided method in superclass is not modified later

DECOMPOSITION

Breaking things down

- In functional (and procedural) programming, break programs down into functions that perform some operation
- In object-oriented programming, break programs down into classes that give behavior to some kind of data

This lecture:

  - These two forms of *decomposition* are so exactly opposite that they are two ways of looking at the same "matrix"
  - Which form is "better" is somewhat personal taste, but also depends on how you expect to *change/extend software*
  - For some operations over two (multiple) arguments, functions and pattern-matching are straightforward, but with OOP we can do it with *double dispatch* (multiple dispatch)

The expression example

Well-known and compelling example of a common *pattern*:

- Expressions for a small language
- Different variants of expressions: ints, additions, negations, …
- Different operations to perform: **`eval`**, **`toString`**, **`hasZero`**, …

Leads to a matrix (2D-grid) of variants and operations

- Implementation will involve deciding what "should happen" for each entry in the grid *regardless of the PL*

	`eval`	**`toString`**	**`hasZero`**	…
`Int`				
`Add`				
`Negate`				
…				

Standard approach in ML

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *datatype*, with one *constructor* for each variant
  - (No need to indicate datatypes if dynamically typed)
- "Fill out the grid" via one function per column
  - Each function has one branch for each column entry
  - Can combine cases (e.g., with wildcard patterns) if multiple entries in column are the same

[See the ML code]

Standard approach in OOP

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *class*, with one *abstract method* for each operation
  - (No need to indicate abstract methods if dynamically typed)
- Define a *subclass* for each variant
- So "fill out the grid" via one class per row with one method implementation for each grid position
  - Can use a method in the superclass if there is a default for multiple entries in a column

[See the Ruby and Java code]

A big course punchline

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- FP and OOP often doing the same thing in *exact* opposite way
  - Organize the program "by rows" or "by columns"
- Which is "most natural" may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste
- Code layout is important, but there is no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple "views" (e.g., rows / columns)

Extensibility

	`eval`	`toString`	`hasZero`	`noNegConstants`
`Int`				
`Add`				
`Negate`				
`Mult`				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* need not change old code
- Functions [see ML code]:
  - Easy to add a new operation, e.g., **`noNegConstants`**
  - Adding a new variant, e.g., **`Mult`** requires modifying old functions, but ML type-checker gives a to-do list if original code avoided wildcard patterns

Extensibility

	`eval`	`toString`	`hasZero`	`noNegConstants`
`Int`				
`Add`				
`Negate`				
`Mult`				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* are easy and/or do not change old code
- Objects [see Ruby code]:
  - Easy to add a new variant, e.g., **`Mult`**
  - Adding a new operation, e.g., **`noNegConstants`** requires modifying old classes, but Java type-checker gives a to-do list if original code avoided default methods

The other way is possible

- Functions allow new operations and objects allow new variants without modifying existing code *even if they didn't plan for it*
  - Natural result of the decomposition

Optional:

- Functions can support new variants somewhat awkwardly "if they plan ahead"
  - *Not explained here: Can use type constructors to make datatypes extensible and have operations take function arguments to give results for the extensions*

- Objects can support new operations somewhat awkwardly "if they plan ahead"
  - *Not explained here: The popular Visitor Pattern uses the double-dispatch pattern to allow new operations "on the side"*

Thoughts on Extensibility

- Making software extensible is valuable and hard
  - If you know you want new operations, use FP
  - If you know you want new variants, use OOP
  - If both? Languages like Scala try; it's a hard problem
  - Reality: The future is often hard to predict!

- Extensibility is a double-edged sword
  - Code more reusable without being changed later
  - But makes original code more difficult to reason about locally or change later (could break extensions)
  - Often language mechanisms to make code *less* extensible (ML modules hide datatypes; Java's **`final`** prevents subclassing/overriding)

Binary operations

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Situation is more complicated if an operation is defined over multiple arguments that can have different variants
  - Can arise in original program or after extension

- Function decomposition deals with this much more simply…

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

ML Approach

Addition is different for most **`Int`**, **`String`**, **`Rational`** combinations

- Run-time error for non-value expressions

Natural approach: pattern-match on the pair of values

- For *commutative* possibilities, can re-call with **`(v2,v1)`**

```
fun add_values (v1,v2) =
  case (v1,v2) of
      (Int i, Int j) => Int (i+j)
    | (Int i, String s) => String (Int.toString i ^ s)
    | (Int i, Rational(j,k)) => Rational (i*k+j,k)
    | (Rational _, Int _) => add_values (v2,v1)
    | … (* 5 more cases (3*3 total): see the code *)

fun eval e =
  case e of
      …
    | Add(e1,e2) => add_values (eval e1, eval e2)
```

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	`String`	`Rational`
`Int`			
`String`			
`Rational`			

Worked just fine with functional decomposition -- what about OOP…

What about OOP?

Starts promising:

- Use OOP to call method **`add_values`** to one value with other value as result

```
class Add
  …
  def eval
    e1.eval.add_values e2.eval
  end
end
```

Classes **`Int`**, **`MyString`**, **`MyRational`** then all implement

- Each handling 3 of the 9 cases: "add **`self`** to argument"

```
class Int
  …
  def add_values v
    … # what goes here?
  end
end
```

First try

- This approach is common, but is "not as OOP"
  - *So do not do it on your homework*

```
class Int
  def add_values v
    if v.is_a? Int
       Int.new(v.i + i)
    elsif v.is_a? MyRational
       MyRational.new(v.i+v.j*i,v.j)
    else
       MyString.new(v.s + i.to_s)
  end
end
```

- A "hybrid" style where we used dynamic dispatch on 1 argument and then switched to Racket-style type tests for other argument
  - Definitely not "full OOP"

Another way…

- **`add_values`** method in **`Int`** needs "what kind of thing" **`v`** has
  - Same problem in **`MyRational`** and **`MyString`**
- In OOP, "always" solve this by calling a method on **`v`** instead!
- But now we need to "tell" **`v`** "what kind of thing" **`self`** is
  - We know that!
  - "Tell" **`v`** by calling different methods on **`v`**, passing **`self`**
- Use a "programming trick" (?) called *double-dispatch*…

Double-dispatch "trick"

- **`Int`**, **`MyString`**, and **`MyRational`** each define all of **`addInt`**, **`addString`**, and **`addRational`**
  - For example, **`String`**'s **`addInt`** is for adding concatenating an integer argument to the string in **`self`**
  - 9 total methods, one for each case of addition

- **`Add`**'s **`eval`** method calls **`e1.eval.add_values e2.eval`**, which dispatches to **`add_values`** in **`Int`**, **`String`**, or **`Rational`**
  - **`Int`**'s **`add_values`**: **`v.addInt self`**
  - **`MyString`**'s **`add_values`**: **`v.addString self`**
  - **`MyRational`**'s **`add_values`**: **`v.addRational self`**

  So **`add_values`** performs "2nd dispatch" to the correct case of 9!

[Definitely see the code]

Why showing you this

- Honestly, partly to belittle full commitment to OOP
- To understand dynamic dispatch via a sophisticated idiom
- Because required for the homework
- To contrast with *multimethods* (optional)

Works in Java too

- In a statically typed language, double-dispatch works fine
  - Just need all the dispatch methods in the type

```
abstract class Value extends Exp {
  abstract Value add_values(Value other);
  abstract Value addInt(Int other);
  abstract Value addString(Strng other);
  abstract Value addRational(Rational other);
}
class Int extends Value { … }
class Strng extends Value { … }
class Rational extends Value { … }
```

[See Java code]

Being Fair

Belittling OOP style for requiring the manual trick of double dispatch is somewhat unfair…

What would work better:

- **`Int`**, **`MyString`**, and **`MyRational`** each define three methods all named **`add_values`**
  - One **`add_values`** takes an **`Int`**, one a **`MyString`**, one a **`MyRational`**
  - So 9 total methods named **`add_values`**
  - **`e1.eval.add_values e2.eval`** picks the right one of the 9 at run-time using the classes of the two arguments
- Such a semantics is called *multimethods* or *multiple dispatch*

FINAL

Final Exam

Next Thursday, 8:30-10:20

- Focus primarily on material since the midterm
  - Including topics on homeworks and not on homeworks
  - Will also have a little ML, just like the course has had

- You will need to write code and English

Final: What to Expect

Practice finals will be *slightly* more predictive.
More forgiving partial credit.

Topics:

- functional programming / list processing
- thunks, streams, promises
- references, purity, aliasing, shallow vs. deep copy
- anonymous funcs, lexical scope, higher order funcs
- blocks and procs
- subclassing and dynamic dispatch
- static typing vs. dynamic typing, soundness, completeness
- implementing closures


1110
6

Victory Lap

A victory lap is an extra trip
around the track

- By the exhausted victors (us) ☺

Review course goals

- Slides from Introduction and Course-Motivation

Some big themes and perspectives

- Stuff for five years from now more than for the final

Course evaluations: please do take some time

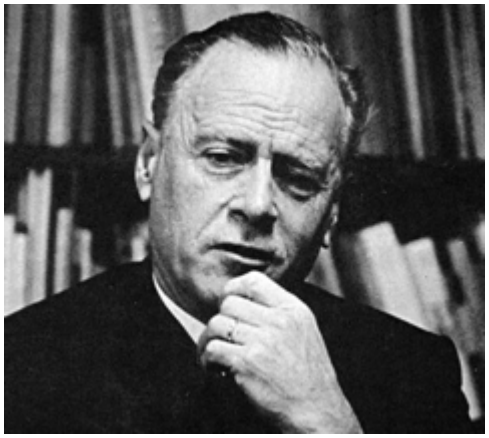
I **really** like studying programming languages.

Super stoked to explore PL with all of you.

Why?

We shape our tools and thereafter our tools shape us.

Marshall McLuhan



M. K. Asante

I discover that I think in words. The more words I know, the more things I can think about...

Reading was illegal because if you limit someone's vocab, you limit their thoughts. They can't even think of freedom because they don't have the language to.

I **really** like studying programming languages.

Super stoked to explore PL with all of you.

Why?

PL helps us *break free* to think thoughts, ask questions, and solve problems that would otherwise be inaccessible.

Looking back on the quarter…

We had 10 short weeks to learn *the fundamental concepts* of PL.

Curiosity and persistence will get you everywhere.

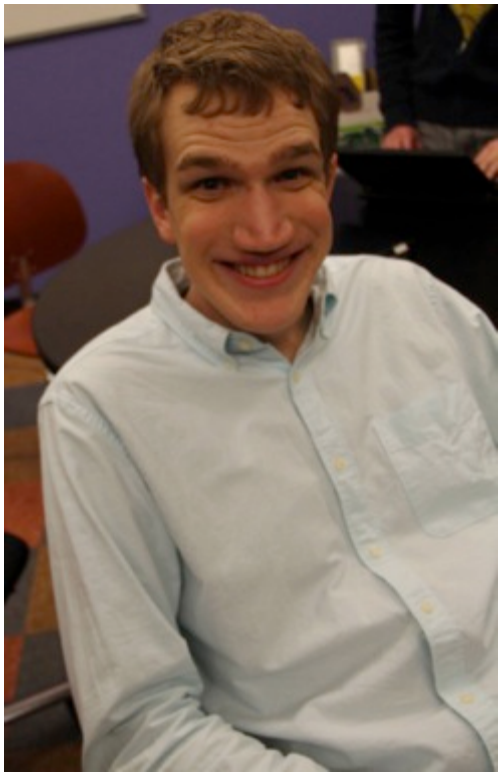
We'll become better programmers:

- Even in languages we won't use
- Learn the core ideas around which *every* language is built, despite countless surface-level differences and variations

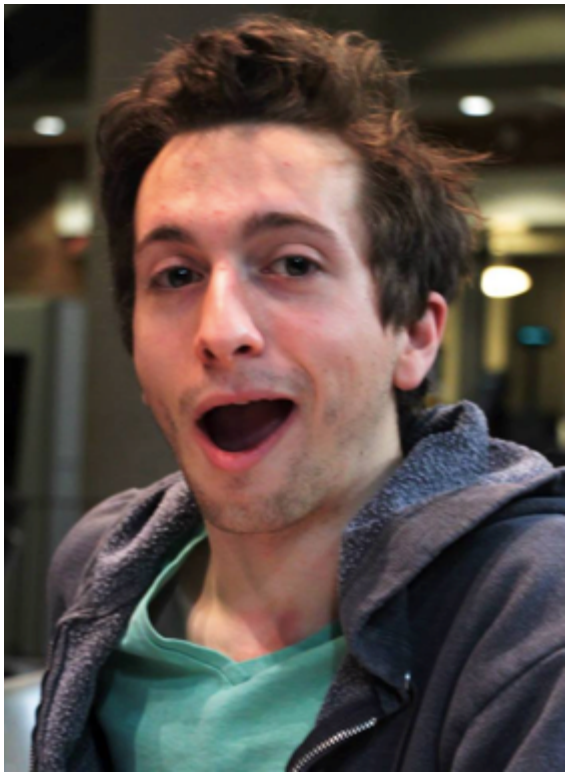
THANK YOU Incredible Guides!!!

super ultra helpful, extraordinarily smart, stellar smiles

Armando Diaz Tolentino



Riley Klingler



Max Sherman

THANK YOU Our Guide in Spirit!!!

(spiritual guide?)

Dan Grossman

Creator of this flavor of 341.

THANK YOU. . . YOU!!1!!eleven!!one!!1!

- And a huge thank you to all of **you**
  - Great attitude about a very different view of software
  - Good class attendance and questions

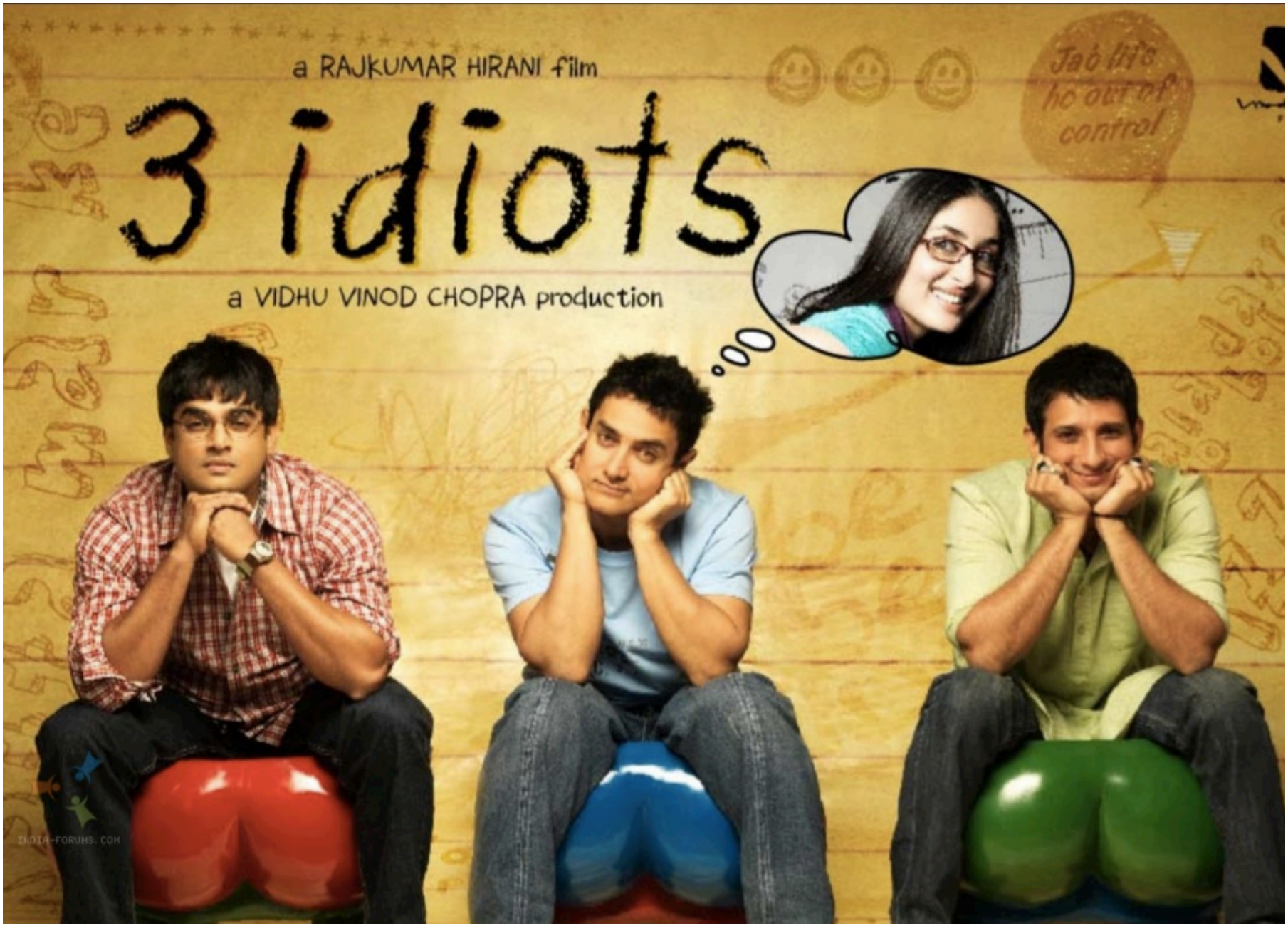
- Computer science ought to be challenging and fun!

What this course is about

- Many essential concepts relevant in any programming language
  - And how these pieces fit together
- Use ML, Racket, and Ruby:
  - They let various important concepts “shine”
  - Using multiple languages shows how the same concept just can “look different” or actually be slightly different
  - In many ways simpler than Java
- Big focus on *functional programming*
  - Not using *mutation* (assignment statements) (!)
  - Using *first-class functions* (can’t explain that yet)
  - But many other topics too

Why learn this?

To free our minds from the shackles of imperative programming.


a RAJKUMAR HIRANI film
3 idiots
a VIDHU VINOD CHOPRA production
Jab life ho out of control

I **really** like studying programming languages.

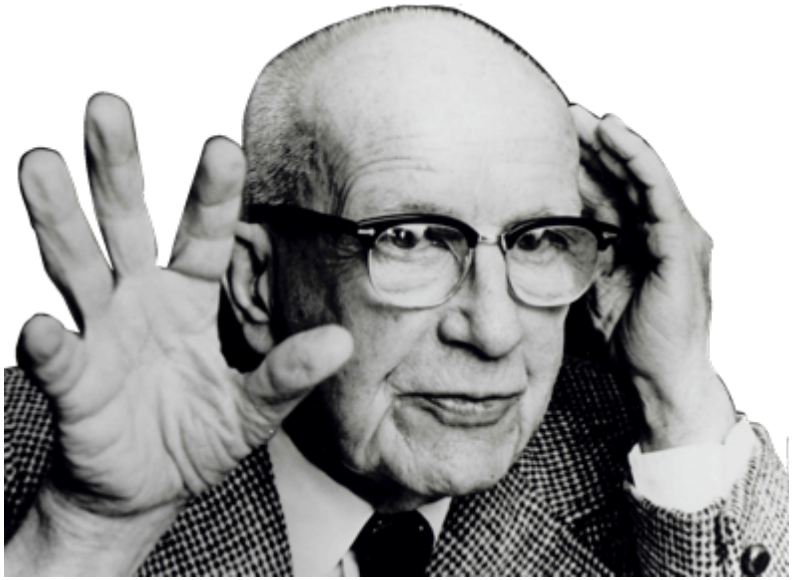
Super stoked to explore PL with all of you.

Why?

If you are in a shipwreck and all the boats are gone, a piano top buoyant enough to keep you afloat may come along and make a fortuitous life preserver.

This is not to say, though, that the best way to design a life preserver is in the form of a piano top.

I think we are clinging to a great many piano tops in accepting yesterday's fortuitous contrivings as constituting the only means for solving a given problem.



R. Buckminster Fuller

More Detailed Course Motivation

- Why learn fundamental concepts that appear in all languages?
- Why use languages quite different from C, C++, Java, Python?
- Why focus on functional programming?
- Why use ML, Racket, and Ruby in particular?
- Not: Language X is better than Language Y

[You won't be tested on this stuff]

Summary

- No such thing as a “best” PL
- Fundamental concepts easier to teach in some (multiple) PLs
- A good PL is a relevant, elegant interface for writing software
  - There is no substitute for precise understanding of PL semantics
- Functional languages have been on the leading edge for decades
  - Ideas have been absorbed by the mainstream, but very slowly
  - First-class functions and avoiding mutation increasingly essential
  - Meanwhile, use the ideas to be a better C/Java/PHP hacker
- Many great alternatives to ML, Racket, and Ruby, but each was chosen for a reason and for how they complement each other

[From Course Motivation]

SML, Racket, and Ruby are a useful *combination* for us

	dynamically typed	statically typed
functional	Racket	SML
object-oriented	Ruby	Java

ML: polymorphic types, pattern-matching, abstract types & modules

Racket: dynamic typing, “good” macros, minimalist syntax, eval

Ruby: classes but not types, very OOP, mixins

[and much more]

Really wish we had more time:

Haskell: laziness, purity, type classes, monads

Prolog: unification and backtracking

[and much more]

Benefits of No Mutation

[An incomplete list]

1. Can freely alias or copy values/objects: Unit 1

2. More functions/modules are equivalent: Unit 4

3. No need to make local copies of data: Unit 5

4. Depth subtyping is sound: Unit 8

State updates are appropriate when you are modeling a phenomenon that is inherently state-based

- A fold over a collection (e.g., summing a list) is not!

Some other highlights

- Function closures are *really* powerful and convenient…
  - … and implementing them is not magic
- Datatypes and pattern-matching are really convenient…
  - … and exactly the opposite of OOP decomposition
- Sound static typing prevents certain errors…
  - … and is inherently approximate
- Subtyping and generics allow different kinds of code reuse…
  - … and combine synergistically
- Modularity is really important; languages can help

From the syllabus

Successful course participants will:

- Internalize an accurate understanding of what functional and object-oriented programs mean
- Develop the skills necessary to learn new programming languages quickly
- Master specific language concepts such that they can recognize them in strange guises
- Learn to evaluate the power and elegance of programming languages and their constructs
- Attain reasonable proficiency in the ML, Racket, and Ruby languages and, as a by-product, become more proficient in languages they already know