CMSC330

Objects, Functional Programming, and lambda calculus

OOP vs. FP

- Object-oriented programming (OOP)
  - Computation as interactions between objects
  - Objects encapsulate mutable data (state)
    - Accessed / modified via object's public methods

- Functional programming (FP)
  - Computation as evaluation of functions
    - Mutable data used to improve efficiency
  - Higher-order functions implemented as closures
    - Closure = function + environment

An Integer "Stack" Abstraction in Java

```
class Stack {
  class Node {
    Integer val; Node next;
    Node(Integer v, Node n) { val = v; next = n; }
  };
  private Node theStack;
  void push(Integer v) {
    theStack = new Node(v, theStack);
  }
  Integer pop() {
    if (theStack == null)
      throw new NoSuchElementException();
    Integer temp = theStack.val;
    theStack = theStack.next;
    return temp;
  }
}
```

A “Stack” Abstraction in OCaml

```
module type STACK =
  sig
    type 'a stack
    val new_stack : unit -> 'a stack
    val push : 'a stack -> 'a -> unit
    val pop : 'a stack -> 'a
  end

module Stack : STACK =
  struct
    type 'a stack = 'a list ref
    let new_stack () = ref []
    let push s x = s := (x::!s)
    let pop s = match !s with
      [] -> failwith "Empty stack"
    | (h::t) -> s := t; h
  end
```

Another "Stack" Abstraction in OCaml

```
let new_stack () =
  let this = ref [] in
    let push x = this := (x::!this)
    and pop () = match !this with
        [] -> failwith "Empty stack"
      | (h::t) -> this := t; h
    in
    (push, pop)
```

```
# let s = new_stack ();;
val s : ('_a -> unit) * (unit -> '_a) = (<fun>, <fun>)
# Pervasives.fst s 3;;  (* applies 1st part of s to 3  *)
- : unit = ()
# Pervasives.snd s ();; (* applies 2nd part of s to () *)
- : int = 3
```

Two OCaml Stack Implementations

- 1st implementation (OOP style)
  - Based on modules
  - Specifies methods for
    - Creating stack
    - Pushing value onto stack parameter
    - Popping value from stack parameter
- 2nd implementation (FP style)
  - Based on closures
  - Creating stack returns tuple containing
    - Closure for pushing value onto created stack
    - Closure for popping value from created stack

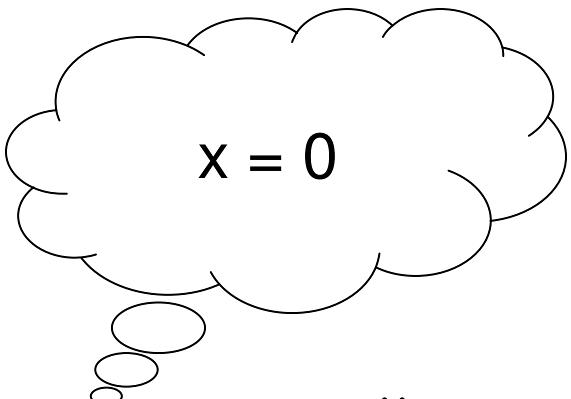
Relating Objects and Closures

- An object...
  - Is a collection of fields (data)
  - ...and methods (code)
  - When a method is invoked
    - Method has implicit this parameter that can be used to access fields of object
- A closure...
  - Is a pointer to an environment (data)
  - ...and a function body (code)
  - When a closure is invoked
    - Function has implicit environment that can be used to access variables

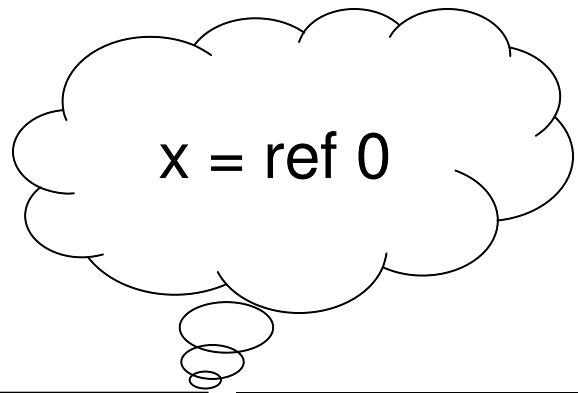
Relating Objects and Closures (cont.)

```
class C {
  int x = 0;
  void set_x(int y) { x = y; }
  int get_x() { return x; }
}
```

```
let make () =
 let x = ref 0 in
   ( (fun y -> x := y),
     (fun () -> !x) )
```




```
C c = new C();
c.set_x(3);
int y = c.get_x();
```




```
fun y -> x := y
```

```
fun () -> !x
```

```
let (set, get) = make ();;
set 3;;
let y = get ();;
```

Recall a Useful Higher-Order Function

```
let rec map f = function
   [] -> []
| (h::t) -> (f h)::(map f t)
```

- Map applies an arbitrary function f
  - To each element of a list
  - And returns the resulting modified list
- Can we encode this in Java?
  - Using object oriented programming

A Map Method for Stack

- Problem – Write a map method in Java
  - Must pass a function into another function
- Solution
  - Can be done using an object with a known method
  - Use interface to specify what method must be present

```
public interface Function {
  Integer eval(Integer arg);
}
```

A Map Method for Stack (cont.)

- Examples
  - Two classes which both implement Function interface

```
class AddOne implements Function {
  Integer eval(Integer arg) {
    return new Integer(arg + 1);
  }
}
```

```
class MultTwo implements Function {
  Integer eval(Integer arg) {
    return new Integer(arg * 2);
  }
}
```

The New Stack Class

```
class Stack {
  class Node {
    Integer val; Node next;
    Node (Integer v, Node n) { val = v; next = n; }
    Entry map(Function f) {
      if (next == null)
        return new Node(f.eval(val), null);
      else return new Node(f.eval(val), next.map(f));
    }
  }
  Node theStack;
  ...
  Stack map(Function f) {
    Stack s = new Stack();
    s.theStack = theStack.map(f);
    return s;
  }
}
```

Applying Map To A Stack

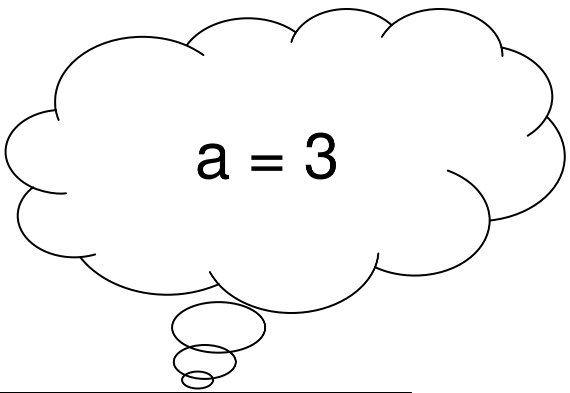
- Then to apply the function, we just do

```
Stack s = ...;
Stack t = s.map(new AddOne());
Stack u = s.map(new MultTwo());
```

  - We make a new object
    - That has a method that performs the function we want
  - This is sometimes called a callback
    - Because map “calls back” to the object passed into it
  - But it’s really just a higher-order function
    - Written more awkwardly

Relating Closures and Objects

```
let app f x = f x
```



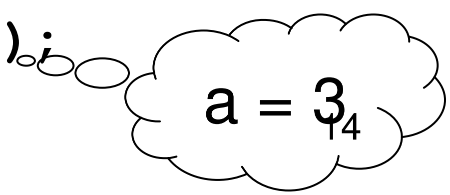

```
fun b -> a + b
```

```
let add a b = a + b;;
let f = add 3;;
app f 4;;
```

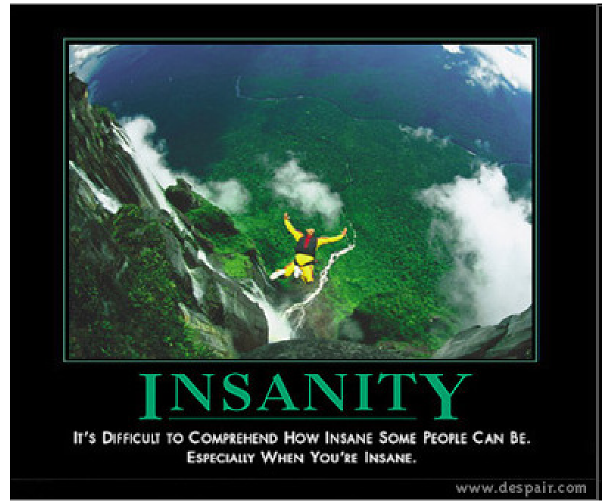
```
interface F {
  Integer eval(Integer y);
}
class C {
  static Integer app(F f, Integer x) {
    return f.eval(x);
  }
}
```

```
class G implements F {
  Integer a;
  G(Integer a) { this.a = a; }
  Integer eval(Integer y) {
    return new Integer(a + y);
  }
}
```

```
F adder = new G(3);
C.app(adder, 4);
```



Code as Data

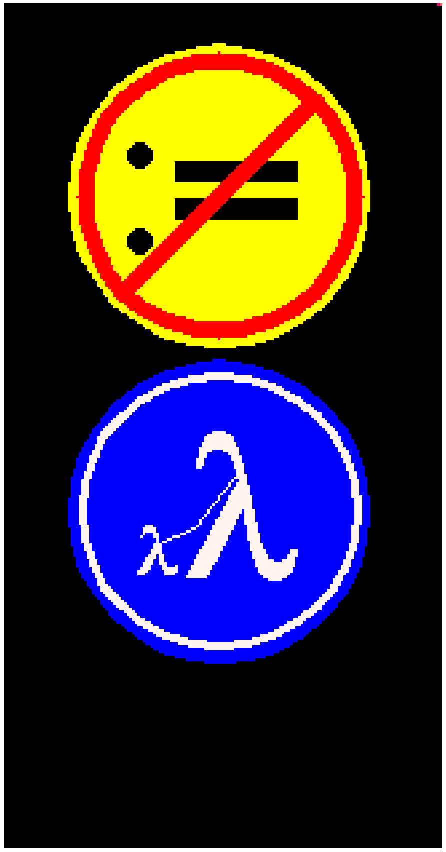


- Closures and objects are related
  - Both of them allow
    - Data to be associated with higher-order code
    - Pass code around the program
- The key insight in all of these examples
  - Treat code as if it were data
    - Allowing code to be passed around the program
    - And invoked where it is needed (as callback)
- Approach depends on programming language
  - Higher-order functions (OCaml, Ruby, Lisp)
  - Function pointers (C, C++)
  - Objects with known methods (Java)

Code as Data (cont.)

- This is a powerful programming technique
  - Solves a number of problems quite elegantly
    - Create new control structures (e.g., Ruby iterators)
    - Add operations to data structures (e.g., visitor pattern)
    - Event-driven programming (e.g., observer pattern)
  - Keeps code separate
    - Clean division between higher & lower-level code
  - Promotes code reuse
    - Lower-level code supports different callbacks

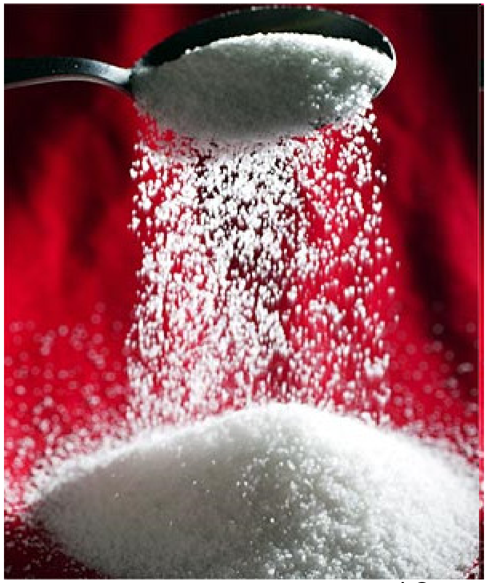
Lambda Calculus



Programming Language Features

- Many features exist simply for convenience
  - Multi-argument functions   foo (a, b, c)
    - Use currying or tuples
  - Loops   while (a < b) …
    - Use recursion
  - Side effects   a := 1
    - Use functional programming

- So what language features are really needed?

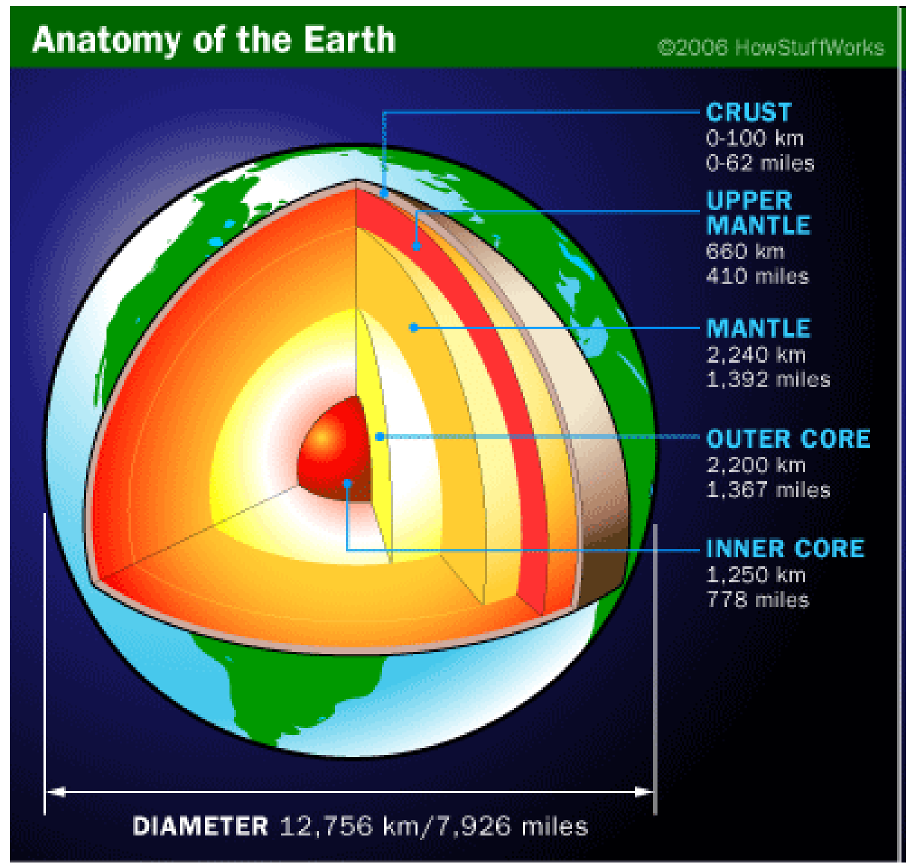


Turing Completeness

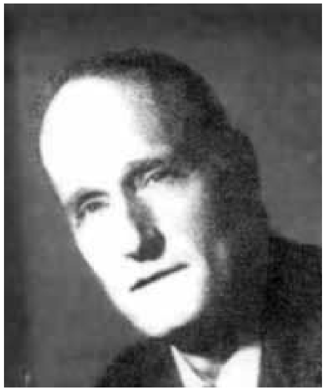
- Computational system that can
  - Simulate a Turing machine
  - Compute every Turing-computable function
- A programming language is Turing complete if
  - It can map every Turing machine to a program
  - A program can be written to emulate a Turing machine
  - It is a superset of a known Turing-complete language
- Most powerful programming language possible
  - Since Turing machine is most powerful automaton

Programming Language Theory

- Come up with a “core” language
  - That’s as small as possible
  - But still Turing complete
- Helps illustrate important
  - Language features
  - Algorithms
- One solution
  - Lambda calculus



Lambda Calculus (λ-calculus)



- Proposed in 1930s by
  - Alonzo Church
  - Stephen Cole Kleene
- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics
- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell…

Lambda Expressions

- A lambda calculus expression is defined as

e ::= x	variable
| λx.e	function
| e e	function application

- λx.e is like `(fun x -> e)` in OCaml

- That's it! Nothing but higher-order functions

Three Conveniences

- Syntactic sugar for local declarations
  - let x = e1 in e2 is short for (λx.e2) e1
- Scope of λ extends as far right as possible
  - Subject to scope delimited by parentheses
  - λx. λy.x y is same as λx.(λy.(x y))
- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml

Lambda Calculus Semantics

- All we've got are functions
  - So all we can do is call them
- To evaluate (λx.e1) e2
  - Evaluate e1 with x bound to e2
- This application is called **beta-reduction**
  - (λx.e1) e2 → e1[x/e2]
    - e1[x/e2] is e1 where occurrences of x are replaced by e2
  - We allow reductions to occur anywhere in a term

Beta Reduction Example

- $(\lambda x.\lambda z.x\ z)\ y$

  $\rightarrow (\lambda x.(\lambda z.(x\ z)))\ y$ // since λ extends to right

  $\rightarrow (\lambda x.(\lambda z.(x\ z)))\ y$ // apply $(\lambda x.e1)\ e2 \rightarrow e1[x/e2]$
  // where $e1 = \lambda z.(x\ z)$, $e2 = y$

  $\rightarrow \lambda z.(y\ z)$ // final result

Parameters
– Formal
– Actual

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

Lambda Calculus Examples

- $(\lambda x.x)\ z \rightarrow z$
- $(\lambda x.y)\ z \rightarrow y$
- $(\lambda x.x\ y)\ z \rightarrow z\ y$
  - A function that applies its argument to y

Lambda Calculus Examples (cont.)

- $(\lambda x.x\ y)\ (\lambda z.z) \rightarrow (\lambda z.z)\ y \rightarrow y$
- $(\lambda x.\lambda y.x\ y)\ z \rightarrow \lambda y.z\ y$
  - A curried function of two arguments
  - Applies its first argument to its second
- $(\lambda x.\lambda y.x\ y)\ (\lambda z.zz)\ x \rightarrow$

  $\lambda y.((\lambda z.zz)y)x \rightarrow \quad (\lambda z.zz)x \rightarrow xx$

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - $(\lambda x.x\ (\lambda x.x))\ z \rightarrow ?$
    - The rightmost "x" refers to the second binding
  - This is a function that
    - Takes its argument and applies it to the identity function
- This function is "the same" as $(\lambda x.x\ (\lambda y.y))$
  - Renaming bound variables consistently is allowed
    - This is called alpha-renaming or alpha conversion
  - Ex. $\lambda x.x = \lambda y.y = \lambda z.z \qquad \lambda y.\lambda x.y = \lambda z.\lambda x.z$

Static Scoping (cont.)

- How about the following?
  - (λx.λy.x y) y → ?
  - When we replace y inside, we don't want it to be captured by the inner binding of y
  - I.e., (λx.λy.x y) y ≠ λy.y y

- Solution
  - (λx.λy.x y) is "the same" as (λx.λz.x z)
    - Due to alpha conversion
  - So change (λx.λy.x y) y to (λx.λz.x z) y first
    - Now (λx.λz.x z) y → λz.y z

Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - (λx.e1) e2 → e1[x/e2]
  - We must first alpha-convert variables as necessary
  - Usually performed implicitly (w/o showing conversion)

- Examples
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z // y → z
  - (λx.x (λx.x)) z = (λy.y (λx.x)) z → z (λx.x) // x → y
  - (λx.x (λx.x)) z = (λx.x (λy.y)) z → z (λy.y) // x → y

Encodings

- The lambda calculus is Turing complete
- Means we can encode any computation we want
  - If we're sufficiently clever...
- Examples
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping

Booleans

- Church's encoding of mathematical logic
  - true = λx.λy.x
  - false = λx.λy.y
  - if a then b else c

- Examples
  - if true then b else c → (λx.λy.x) b c → (λy.b) c → b
  - if false then b else c → (λx.λy.y) b c → (λy.y) c → c

Booleans (cont.)

- Other Boolean operations
  - not = λx.((x false) true)
    - not true → λx.((x false) true) true → ((true false) true) → false
  - and = λx.λy.((xy) false)
  - or = λx.λy.((x true) y)

- Given these operations
  - Can build up a logical inference system

Discussion

- Lambda calculus is Turing-complete
  - Most powerful language possible
  - Can represent pretty much anything in “real” language
    - Using clever encodings
- But programs would be
  - Pretty slow
  - Pretty large
  - Pretty hard to understand
- In practice
  - We use richer, more expressive languages
  - That include built-in primitives

The Need For Types

- Consider the untyped lambda calculus
  - false = $\lambda x.\lambda y.y$
  - 0 = $\lambda x.\lambda y.y$
- Since everything is encoded as a function...
  - We can easily misuse terms…
    - false 0 $\rightarrow \lambda y.y$
    - if 0 then ...

  …because everything evaluates to some function
- The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words

Simply-Typed Lambda Calculus

- $e ::= n \mid x \mid \lambda x{:}t.e \mid e\ e$
  - Added integers n as primitives
    - Need at least two distinct types (integer & function)…
    - …to have type errors
  - Functions now include the type of their argument

Simply-Typed Lambda Calculus (cont.)

- t ::= int | t → t
  - int is the type of integers
  - t1 → t2 is the type of a function
    - That takes arguments of type t1 and returns result of type t2
  - t1 is the domain and t2 is the range
  - Notice this is a recursive definition
    - So we can give types to higher-order functions
- Will show how to compute types later
  - Example of operational semantics

Summary

- Lambda calculus shows issues with
  - Scoping
  - Higher-order functions
  - Types

- Useful for understanding how languages work