

## CMSC 330: Organization of Programming Languages

### Objects vs. Functional Programming

## 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

CMSC 330

2

## 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;
  }
}
```

CMSC 330

3

## 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
```

CMSC 330

4

## 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
```

CMSC 330

5

## Two OCaml Stack Implementations

- ▶ 1<sup>st</sup> implementation (OOP style)
  - Based on modules
  - Specifies methods for
    - Creating stack
    - Pushing value onto stack parameter
    - Popping value from stack parameter
- ▶ 2<sup>nd</sup> 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

CMSC 330

6

## 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

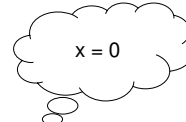
CMSC 330

7

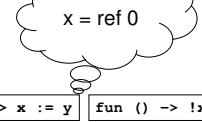
## Relating Objects and Closures (cont.)

```
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 ();;
```

CMSC 330

8

## Encoding Objects with Functions

- ▶ We can apply this transformation in general

```
class C { f1 ... fn; m1 ... mn; }
```

- becomes

```
let make () =
  let f1 = ...
  ...
  and fn = ... in
  ( fun ... , (* body of m1 *)
    ...
    fun ... , (* body of mn *)
  )
```

} Tuple containing closures

- `make ()` is like the constructor
- The closure environment contains the fields

CMSC 330

9

## 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

CMSC 330

10

## 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);
}
```

CMSC 330

11

## 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);
  }
}
```

CMSC 330

12

## 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;
  }
}
```

CMSC 330

13

## 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());
```

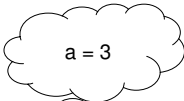
- We make a new object
  - That has a method that performs the function we want
- This is sometimes called a **callback**
  - Because map "calls back" to the object passed into it
- But it's really just a **higher-order function**
  - Written more awkwardly

CMSC 330

14

## Relating Closures and Objects

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



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

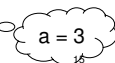
```
let add a b = a + b;;
let f = add 3;;
app f 4;;
```

CMSC 330

```
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);
```



## Encoding Functions with Objects

- We can apply this transformation in general

```
... (fun x -> (* body of fn *)) ...
let h f ... = ... f y ...
```

- becomes

```
interface F { Object eval(Object x); }
class G implements F {
  Object eval(Object x) { /* body of fn */ }
}
class C {
  Typ h(F f, ...) {
    ... f.eval(y) ...
  }
}
```

- F is the interface to the callback
- G represents the particular function

CMSC 330

16

## 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)

CMSC 330

17

## 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

CMSC 330

18