

CMSC 330: Organization of Programming Languages

Relationships between Functional, Imperative, and Object-Oriented Programming

OCaml Language Choices

- Implicit or explicit declarations?
 - Explicit – variables must be introduced with `let` before use
 - But you don't need to specify types
- Static or dynamic types?
 - Static – but you don't need to state types
 - OCaml does *type inference* to figure out types for you
 - Good: less work to write programs
 - Bad: easier to make mistakes, harder to find errors
- Functional programming

CMSC 330

2

What about Imperative Programming?

- In C or Java, we're used to doing things like:

```
int x = 3;
int y = 4;

void foo(void) {
    x = 42;
    y = x + 2;
}

int bar(void) {
    return x + y;
}
```

- Can we model this without imperative constructs?

- Imperative = able to *change* values in memory

CMSC 330

3

Idea: "Thread" State through Fns

```
type state = (char * int) list
let read (s:state) (x:char):int = List.assoc x s
let write (s:state) (x:char) (i:int):state =
    let s' = List.remove_assoc s x in
    (x,i)::s'

let foo (s0:state):state = (* could change to state*unit *)
    let s1 = write s0 'x' 42 in
    let s2 = write s1 'y' ((read s1 'x') + 2) in
    s2

let bar (s0:state):(state*int) =
    (s0, (read s0 'x') + (read s0 'y'))
```

CMSC 330

4

This Can Actually be a Good Idea

- The Haskell language is *purely functional*
 - No way to write to memory, ever
- But, you can play the trick we just saw
 - In Haskell, something that behaves like the state type is a *monad*
 - Used for a bunch of different things
 - And there's some interesting theory to go with it
- OCaml is only *mostly functional*
 - It does actually have imperative constructs

CMSC 330

5

Object-Oriented Languages

- We've seen how we could model state without imperative features
- What about object-oriented constructs?
 - Can we encode them?

CMSC 330

6

The Stack Class

```
class Stack<Typ> {
  class Entry {
    Typ elt; Entry next;
    Entry(Typ e, Entry n) { elt = e; next = n; }
  };
  private Entry theStack;
  void push(Typ e) {
    theStack = new Entry(e, theStack);
  }
  Typ pop() {
    if (theStack == null) throw new NoSuchElementException();
    Typ temp = theStack.elt;
    theStack = theStack.next;
    return temp;
  }
}
```

CMSC 330

7

Writing a "Stack" in OCaml: Take 1

```
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

8

Writing a "Stack" in OCaml: Take 2

```
let new_stack () =
  let this = ref [] in
  let push x = this := (x::!this) in
  let 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>)
# fst s 3;;
- : unit = ()
# snd s ();;
- : int = 3
```

CMSC 330

9

Relating Objects and Closures

- An object...
 - Is a collection of fields (data)
 - ...and methods (code)
 - When a method is invoked, it is passed an implicit *this* parameter it can use to access fields
- A closure...
 - Is a pointer to an environment (data)
 - ...and a function body (code)
 - When a closure is invoked, it is passed its environment it can use to access variables

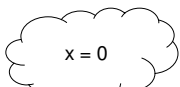
CMSC 330

10

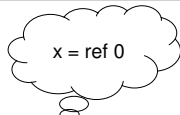
Relating Objects and Closures (cont'd)

```
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

11

Encoding Objects with Lambda

- We can apply this transformation in general

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

– becomes

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

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

CMSC 330

12

Recall a Useful Higher-Order Function

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

- Can we encode this in Java?

CM/SC 330

13

A Map Method for Stack

- To write a map method, we need some way of passing a function into another function
 - We can do that with an object with a known method

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

CM/SC 330

14

A Map Method for Stack, cont'd

- Here are two classes which both implement this Function interface:

```
class AddOne implements Function {
  Object eval(Object arg) {
    Integer old = (Integer) arg;
    return new Integer(old.intValue() + 1);
  }
}
```

```
class MultTwo implements Function {
  Object eval(Object arg) {
    Integer old = (Integer) arg;
    return new Integer(old.intValue() * 2);
  }
}
```

CM/SC 330

15

A Map Method for Stack, cont'd

```
class Stack {
  class Entry {
    Object elt; Entry next;
    Entry(Object x, Entry n) { elt = x; next = n; }
    Entry map(Function f) {
      if (next == null) return new Entry(f.eval(elt), null);
      else return new Entry(f.eval(elt), next.map(f));
    }
  }
  Entry theStack;
  ...

  Stack map(Function f) {
    Stack s = new Stack();
    s.theStack = theStack.map(f);
    return s;
  }
}
```

CM/SC 330

16

A Map Method for Stack, con't.

- Then to apply the function, we just do

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

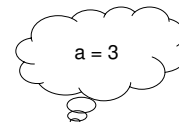
- We make a new object that has a method that performs the function we want
- This is sometimes called a *callback*, because `map` “calls back” to the object passed into it
- But it’s really just a higher-order function, written more awkwardly

CM/SC 330

17

Relating Closures and Objects

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



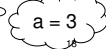
```
fun b -> a + b
```

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

CM/SC 330

```
interface F {
  Object eval(Object y);
}
class C {
  static Object app(F f, Object x) {
    return f.eval(x);
  }
}
class G implements F {
  int a;
  G(int a) { this.a = a; }
  Object eval(Object y) {
    return
      new Integer(a +
        ((Integer) y).intValue());
  }
}
```

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



CM/SC 330

18

Encoding Lambda 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

Code as Data

- The key insight in all of these examples is to treat *code* as if it were *data*
 - Higher-order functions allow code to be passed around the program
 - As does object-oriented programming
- This is a powerful programming technique
 - And it can solve a number of problems quite elegantly
- Closures and objects are related
 - Both of them allow data to be associated with higher-order code as its passed around (but we can even get by without this)