CMSC 330: Organization of Programming Languages

Objects and Functional Programming

and

Names and Bindings

OOP vs. FP

- Object-oriented programming (OOP)
  - Computation as interactions between objects
  - Objects encapsulate state, which is usually mutable
    - 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

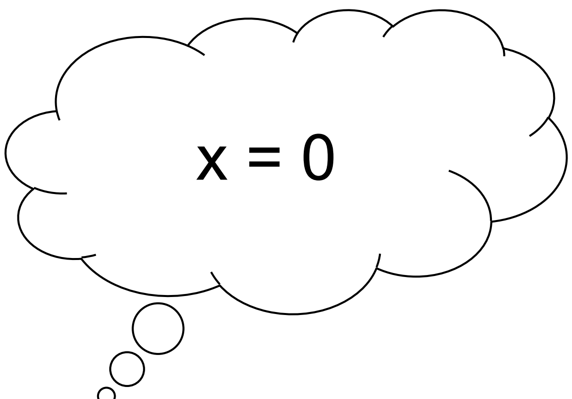
Relating Objects and Closures

- 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

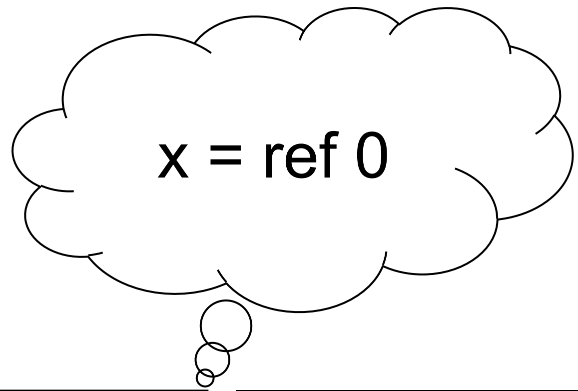
Relating Objects and Closures (cont.)

```
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 ();;
```

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

Relating Closures and Objects

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

a = 3

```
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);
```

a = 3

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

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

An Integer List Abstraction in Java

```
public class MyList {

  private class ConsNode {
    int head; MyList tail;
    ConsNode (int h, MyList l) { head = h; tail = l; }
  }

  private ConsNode contents;

  public MyList () {
    contents = null;
  }

  public MyList (int h, MyList l) {
    contents = new ConsNode (h, l);
  }

  public MyList cons (int h) {
    return (new MyList (h, this));
  }

  public int hd () {
    return contents.head;
  }

  public MyList tl () {
    return contents.tail;
  }

  public boolean isNull () {
    return (contents == null);
  }

}
```

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 Lists in Java

- 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 IntFunction {
  int eval(int arg);
}
```

A Map Method for Lists (cont.)

- Examples
  - Two classes which both implement Function interface

```
class AddOne implements IntFunction {
  int eval (int arg) {
    return (arg + 1);
  }
}
```

```
class MultTwo implements IntFunction {
  int eval(int arg) {
    return (arg * 2);
  }
}
```

The New List Class

```
class MyList {
  …
  public MyList map (IntFunction f) {
    if (this.isNull()) return this;
    else return (this.tl()).map(f).cons (f.eval (this.hd()));
  }
}
```

Applying Map To Lists

- Then to apply the function, we just do

```
MyList l = ...;
MyList l1 = l.map(new AddOne());
MyList l2 = l.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

We Can Do This for Fold Also!

- Recall fold

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

  - Fold accumulates a value (in a) as it traverses a list
  - f is used to determine how to “fold” the head of a list into a
- This can be done in Java using an approach similar to map!

A Fold Method for Lists in Java

- Problem – Write a fold 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 IntBinFunction {
  Integer eval(Integer arg1, Integer arg2);
}
```

A Fold Method for Lists (cont.)

- Examples
  - A classes which implements IntBinFunction interface

```
class Sum implements IntBinFunction {
  Integer eval(Integer arg1, Integer arg2) {
    return new Integer(arg1 + arg2);
  }
}
```

The New New List Class

```
class MyList {
  …
  public MyList map (IntFunction f) {
    if (this.isNull()) return this;
    else return (this.tl()).map(f).cons (f.eval (this.hd()));
  }

  public int fold (IntBinFunction f, int a) {
    if (this.isNull()) return a;
    else return (this.tl()).fold(f, f.eval(a, this.hd()));
  }
}
```

Applying Fold to Lists

- To apply the fold function, we just do this:

```
MyList l = ...;
int s = l.fold (new AddOne(), 0);
```

- The result is that s contains the sum of the elements in l

Names & Binding Overview

- Order of bindings
- Namespaces
- Static (lexical) scopes
- Dynamic scopes
- Funargs

Names and Binding

- Programs use names to refer to things
  - E.g., in x = x + 1, x refers to a variable
- A binding is an association between a name and what it refers to
  - int x; /* x is bound to a stack location containing an int */
  - int f (int) { ... } /* f is bound to a function */
  - class C { ... } /* C is bound to a class */
  - let x = e1 in e2 (* x is bound to e1 *)

Name Restrictions

- Languages often have various restrictions on names to make scanning and parsing easier
  - Names cannot be the same as keywords in the language
  - OCaml function names must be lowercase
  - OCaml type constructor and module names must be uppercase
  - Names cannot include special characters like ; , : etc
    - Usually names are upper- and lowercase letters, digits, and _ (where the first character can't be a digit)
    - Some languages also allow more symbols like ! or -

Names and Scopes

- Good names are a precious commodity
  - They help document your code
  - They make it easy to remember what names correspond to what entities
- We want to be able to reuse names in different, non-overlapping regions of the code

Names and Scopes (cont.)

- A scope is the region of a program where a binding is active
  - The same name in a different scope can refer to a different binding (refer to a different program object)
- A name is in scope if it's bound to something within the particular scope we’re referring to

Example

```
void w(int i) {
  ...
}

void x(float j) {
  ...
}

void y(float i) {
  ...
}

void z(void) {
   int j;
   char *i;
   ...
}
```

- i is in scope
  - in the body of w, the body of y, and after the declaration of j in z
  - but all those i's are different
- j is in scope
  - in the body of x and z

Ordering of Bindings

- Languages make various choices for when declarations of things are in scope

Order of Bindings – OCaml

- let x = e1 in e2 – x is bound to e1 in scope of e2
- let rec x = e1 in e2 – x is bound in e1 and in e2

```
let x = 3 in
  let y = x + 3 in...    (* x is in scope here *)
```

```
let x = 3 + x in ...     (* error, x not in scope *)
```

```
let rec length = function
     [] -> 0
   | (h::t) -> 1 + (length t)   (* ok, length in scope *)
in ...
```

Order of Bindings – C

- All declarations are in scope from the declaration onward

```
int i;
int j = i;  /* ok, i is in scope */
i = 3;      /* also ok */
```

```
void f(...) { ... }

int i;
int j = j + 3;    /* error */
f(...);           /* ok, f declared */
```

```
f(...);  /* may be error; need prototype (or oldstyle C) */

void f(...) { ... }
```

Order of Bindings – Java

- Declarations are in scope from the declaration onward, except for methods and fields, which are in scope throughout the class

```
class C {
  void f(){
    ...g()...    // OK
  }

  void g(){
    ...
  }
}
```

Shadowing Names

- **Shadowing** is rebinding a name in an inner scope to have a different meaning
  - May or may not be allowed by the language

C

```
int i;

void f(float i) {

   {
      char *i = NULL;
      ...
   }

}
```

OCaml

```
let g = 3;;
let g x = x + 3;;
```

Java

```
void h(int i) {
   {
      float i; // not allowed
      ...
   }
}
```

Namespaces

- Languages have a "top-level" or outermost scope
  - Many things go in this scope; hard to control collisions
- Common solution seems to be to add a hierarchy
  - OCaml: Modules
    - List.hd, String.length, etc.
    - open to add names into current scope
  - Java: Packages
    - java.lang.String, java.awt.Point, etc.
    - import to add names into current scope
  - C++: Namespaces
    - namespace f { class g { ... } }, f::g b, etc.
    - using namespace to add names to current scope

Mangled Names

- What happens when these names need to be seen by other languages?
  - What if a C program wants to call a C++ method?
    - C doesn't know about C++'s naming conventions
- For multilingual communication, names are often mangled into some flat form
  - E.g., **`class C { int f(int *x, int y) { ... } }`** becomes symbol **`__ZN1C3fEPii`** in g++
  - E.g., native valueOf(int) in java.lang.String corresponds to the C function **`Java_java_lang_String_valueOf__I`**

Static Scope Recall

- In static scoping, a name refers to its closest binding, going from inner to outer scope in the program text
  - Languages like C, C++, Java, Ruby, and OCaml are statically scoped

```
int i;

{
  int j;

  {
    float i;

    j = (int) i;
  }
}
```

Free and Bound Variables

- The bound variables of a scope are those names that are declared in it
- If a variable is not bound in a scope, it is free
  - The bindings of variables which are free in a scope are inherited from declarations of those variables in outer scopes in static scoping

j is bound in scope 1

j is free in scope 2

```
{ /* 1 */
  int j;

  { /* 2 */
    float i;

    j = (int) i;
  }
}
```

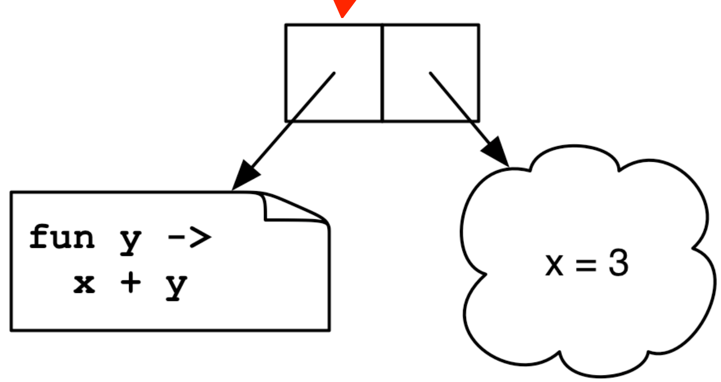
i is bound in scope 2

Static Scoping and Nested Functions

- To allow arbitrary nested functions with higher-order functions and static scoping, we needed closures

```
let add x = (fun y -> x + y)
```

`(add 3) 4` → `<closure> 4` → `3 + 4` → `7`



Functional Arguments (Funargs)

- Funarg problem
  - Difficult to implement functions as first-class objects in stack-based programming languages
- Downwards funargs
  - Passing function as parameter to another function call
  - Can be implemented efficiently
    - Since stack frame will still be on stack when funarg is used
    - Techniques such as access links / displays (see CMSC 430)
- Upwards funargs
  - Returning a function from a function call
  - Implementation requires closures (stored on heap)

Example

```
let f x =
  let g y = x + y in
    g 3
f 1
```

x	1
y	3

- When g is called, x is still on the stack

when is g called?

Answer: when f is called with parameter x

Downward Funarg Example

```
let app f z = f z

let f x =
  let g y = x + y in
    app g 3
f 1
```

x	1
f	g
z	3
y	3

- Function g is passed as parameter to app
  - I.e., g is a downward funarg
- When g is called, x is still on the stack
  - Closure is not needed

Upward Funarg Example

```
let add x = fun y -> x + y
((add 1) 2)
```

x 1 y 2

- Function (fun y -> …) is returned by add
  - I.e., it is an upward funarg
- When (fun y -> …) is called
  - Add has already exited
  - x is no longer on the stack
  - Closure is needed

Dynamic Scope

- In a language with dynamic scoping, a name refers to its closest binding at runtime
  - LISP was the common example

Scheme (top-level scope only is dynamic)

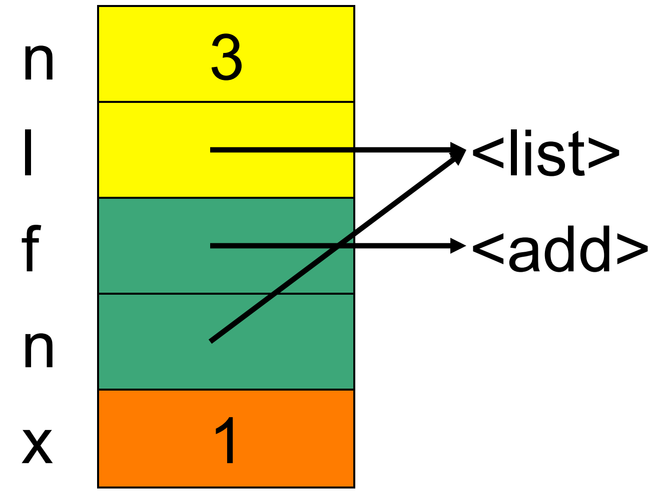
```
(define f (lambda () a))
; defines a no-argument function which returns a

(define a 3)     ; bind a to 3
(f)              ; calls f and returns 3
(define a 4)
(f)              ; calls f and returns 4
```

Previous OCaml Call Stack Example

```
let map (f, n) = match n with
   [] -> []
| (h::t) -> (f h)::(map (f, t))
let addN (n, l) =
   let add x = n + x in
   map (add, l)

addN (3, [1; 2; 3])
```



- How to determine value of n in add ?
  - Dynamic scope: reads it off the stack (n = <list>)
  - Static scope: lexical binding (n = param n to addN)

Nested Dynamic Scopes

- Full dynamic scopes can be nested
  - Static scope relates to the program text
  - Dynamic scope relates to program execution trace

```
Perl (the keyword local introduces dynamic scope)

$l = "global";

sub A {
  local $l = "local";
  B();
}

sub B { print "$l\n"; }

B(); A(); B();
```

```
global
local
global
```

Static vs. Dynamic Scope

Static scoping

- Local understanding of function behavior
- Know at compile-time what each name refers to
- A little more work to implement (keep a link to the lexical nesting scope in stack frame)

Dynamic scoping

- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (keep a global table of stacks of variable/value bindings)