CMSC 330: Organization of Programming Languages

Closures
(Implementing Higher Order Functions)

lec 06

Returning Functions as Results

- In OCaml you can pass functions as arguments to **`map`**, **`fold`**, etc. and you can return functions as results

```
# let pick_fn n =
    let plus3 x = x + 3 in
    let plus4 x = x + 4 in
    if n > 0 then plus3 else plus4
val pick_fn : int -> (int->int) = <fun>

# let g = pick_fn 2;;
val g : int -> int = <fun>
# g 4;;    (* evaluates to 7 *)
```

in OCaml, functions take their args one at a time

Multi-argument Functions

- Consider a rewriting of the prior code (above)

```
let pick_fn n =
  if n > 0 then (fun x->x+3) else (fun x->x+4)
```

- Here’s another version

```
let pick_fn n =
  (fun x -> if n > 0 then x+3 else x+4)
```

Currying

- We just saw a way for a function to take multiple arguments!
  - I.e., no separate concept of multi-argument functions – can encode one as a *function that takes a single argument and returns a function that takes the rest*
- This encoding is called currying the function
  - Named after the logician Haskell B. Curry.
    - three programming languages are named after him: Haskell, Brook, and Curry

Curried Functions In OCaml

- OCaml syntax defaults to currying. E.g.,

```
let add x y = x + y
```

  - is identical to all of the following:

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

  - `add` has type `int -> (int -> int)`
  - `add 3` has type `int -> int`
    - `add 3` is a function that adds 3 to its argument
  - `(add 3) 4 = 7`

you can define add 3:
let g = add 3;
g 10
returns int = 13;

Syntax Conventions for Currying

- Because currying is so common, OCaml uses the following conventions:
  - `->` associates from the right
    - Thus `int -> int -> int` is the same as
    - `int -> (int -> int)`
  - function application associates from the left
    - Thus `add 3 4` is the same as
    - `(add 3) 4`

Quiz 1: Which f definition is equivalent?

```
let f a b = a / b;;
```

```
A. let f b = fun a -> a / b;;
B. let f = fun a -> (fun b -> a / b);;
C. let f = fun a | b -> a / b;;
D. let f (a, b) = a / b;;
```

Quiz 1: Which f definition is equivalent?

```
let f a b = a / b;;
```

```
A. let f b = fun a -> a / b;;
B. let f = fun a -> (fun b -> a / b);;
C. let f = fun a | b -> a / b;;
D. let f (a, b) = a / b;;
```

Quiz 2: What is enabled by currying?

A. Passing functions as arguments
B. Passing only a portion of the expected arguments
C. Naming arguments
D. Recursive functions

because you can pass one arg to a multi-arg function and define that.

let g = plus 3 _ (* missing second argument *)

g 10; is valid

Quiz 2: What is enabled by currying?

A. Passing functions as arguments
B. Passing only a portion of the expected arguments
C. Naming arguments
D. Recursive functions

Multiple Arguments, Partial Application

- Another way for passing multiple arguments is using tuples
  - `let f (a,b) = a / b (* int*int -> int *)`
  - `let f a b = a / b (* int-> int-> int *)`

- Is there a benefit to using currying instead?
  - Supports **partial application** – useful when you want to provide some arguments now, the rest later

Closure

OCaml Example

```
let foo x =
  let bar = fun y -> x + y in
 bar
;;
```

```
foo 10 = ?

(fun y -> x + y) 10?
```

Where is **x**?

Another Example

```
let x = 1 in
  let f = fun y -> x in
  let x = 2 in
f 0
```

What does this expression should evaluate to?

A. 1
B. 2

Another Example

```
let x = 1 in
  let f = fun y -> x in
  let x = 2 in
f 0
```

What does this expression should evaluate to?

A. 1
B. 2

Scope

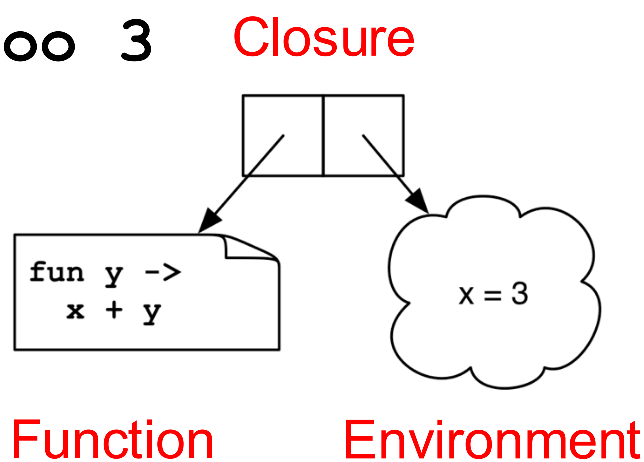
- **Dynamic scope**
  - The body of a function is evaluated in the current dynamic environment at the time the function is **called**, not the old dynamic environment that existed at the time the function was defined.

- **Lexical scope**
  - The body of a function is evaluated in the old dynamic environment that existed at the time the function was **defined**, not the current environment when the function is called.

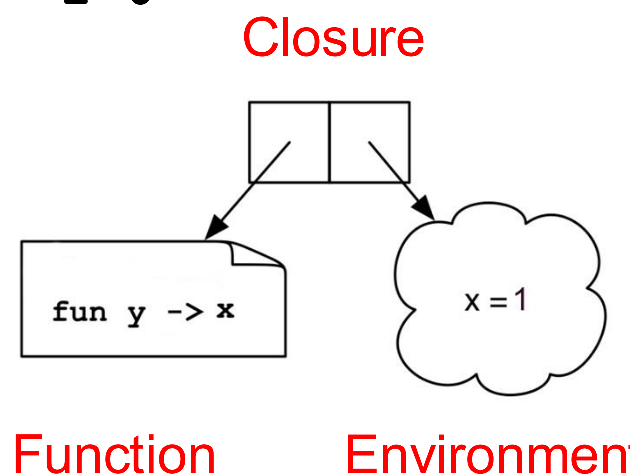
Closure

```
let foo x =
  let bar y = x + y
in
bar ;;
```

foo 3




```
let x = 1 in
let f = fun y -> x
in
let x = 2 in
f 0
```



Closures Implement Static Scoping

- An environment is a mapping from variable names to values
  - Just like a stack frame
- A closure is a pair (f, e) consisting of function code f and an environment e
- When you invoke a closure, f is evaluated using e to look up variable bindings

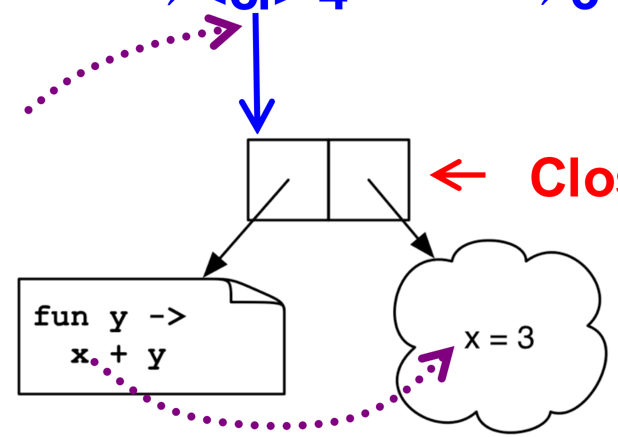
Example – Closure 1

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

(add 3) 4 → <cl> 4 → 3 + 4 → 7

Closure

Function

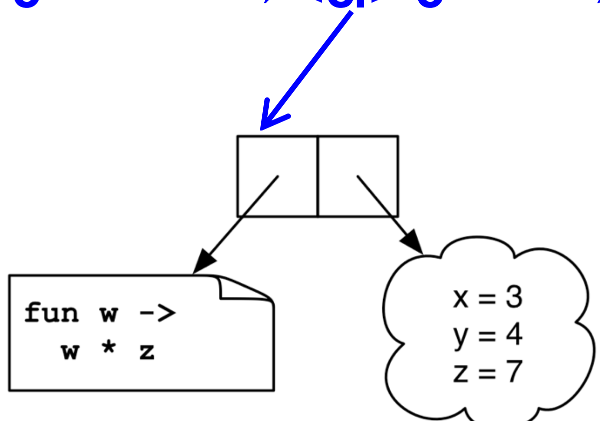


Environment

Example – Closure 2

```
let mult_sum (x, y) =
  let z = x + y in
    fun w -> w * z
```

(mult_sum (3, 4)) 5 → <cl> 5 → 5 * 7 → 35



Quiz 3: What is x?

```
let a = 0;;
let b = 10;;
let f () = a + b;;
let b = 5;;
let x = f ();;
```

A. `15`

B. `1`

C. `10`

D. Error - variable name conflicts

Quiz 3: What is x?

```
let a = 0;;
let b = 10;;
let f () = a + b;;
let b = 5;;
let x = f ();;
```

A. 15

B. 1

C. 10

D. Error - variable name conflicts

Quiz 4: What is z?

```
let f x = fun y -> x - y in
  let g = f 2 in
  let x = 3 in
  let z = g 4 in
z;;
```

A. `-2`

B. `7`

C. `-1`

D. Type Error – insufficient arguments

Quiz 4: What is z?

```
let f x = fun y -> x - y in
let g = f 2 in
let x = 3 in
let z = g 4 in
z;;
```

A. -2

B. 7

C. -1

D. Type Error – insufficient arguments

Quiz 5: What does this evaluate to?

```
let f x = x+1 in
 let g = f in
 g (fun i -> i+1) 10
```

A. Type Error

B. 1

C. 2

D. 3

Quiz 5: What does this evaluate to?

```
let f x = x+1 in
let g = f in
(g (fun i -> i+1)) 10
```

A. **Type Error** – Too many arguments passed to g (application is *left associative*)

B. 1

C. 2

D. 3

Higher-Order Functions in C

- C supports function pointers, but does not support closures

```
typedef int (*int_func)(int);
void app(int_func f, int *a, int n) {
  for (int i = 0; i < n; i++)
    a[i] = f(a[i]);
}
int add_one(int x) { return x + 1; }
int main() {
  int a[] = {5, 6, 7};
  app(add_one, a, 3);
}
```

Java Example

```
public class Test{
  public void doSomething(){
    int a = 10;  //must be final
    Runnable runnable = new Runnable(){
      public void run(){
         int b = a + 1;
         System.out.println(b);
      }
    };
    (new Thread(runnable)).start(); //runs later
       //a = 100; //not allowed
  }
  public static void main(String[] args){
    Test t = new Test();
    t.doSomething();
  }
}// a=10 is removed from the stack here
```

Needed later, makes copy of a

Java 8 Supports Lambda Expressions

- Ocaml's

```
fun (a, b) -> a + b
```

- Is like the following in Java 8

```
(a, b) -> a + b
```

- Java 8 supports closures, and variations on this syntax