CMSC 330: Organization of Programming Languages

Closures
(Implementing Higher Order Functions)

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

- Here, **`pick_fn`** takes an **`int`** argument, and returns a function

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

```
let pick_fn n =
  let plus_three x = x + 3 in
  let plus_four x = x + 4 in
  if n > 0 then plus_three
           else plus_four
```

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)
```

- … the shorthand for which is just

```
let pick_fn n x =
  if n > 0 then x+3 else x+4
```

I.e., a multi-argument function!

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
  - But Schönfinkel and Frege discovered it
    - So maybe it should be called Schönfinkelizing or Fregging

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)
```

- Thus:
  - `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`
- This works for any number of arguments

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

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 you could encode support for 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
  - `let add a b = a + b;;`
  - `let addthree = add 3;;`
  - `addthree 4;; (* evaluates to 7 *)`

Currying is Standard In OCaml

- Pretty much all functions are curried
  - Like the standard library map, fold, etc.
  - See /usr/local/ocaml/lib/ocaml on Grace
    - In particular, look at the file list.ml for standard list functions
    - Access these functions using `List.<fn name>`
    - E.g., `List.hd`, `List.length`, `List.map`
- OCaml works hard to make currying efficient
  - Because otherwise it would do a lot of useless allocation and destruction of closures
  - What are those, you ask? Let's see …

How Do We Implement Currying?

- Implementing currying is tricky. Consider:

```
let addN n l =
  let add x = n + x in
  map add l
```

Accessing variable from outer scope

- (Equivalent to...)

```
let addN n =
  (fun l -> map (fun x -> n + x) l)
```

- When the anonymous function is called by map, n may not be on the stack any more!
  - We need some way to keep n around after addN returns

The Call Stack in C/Java/etc.

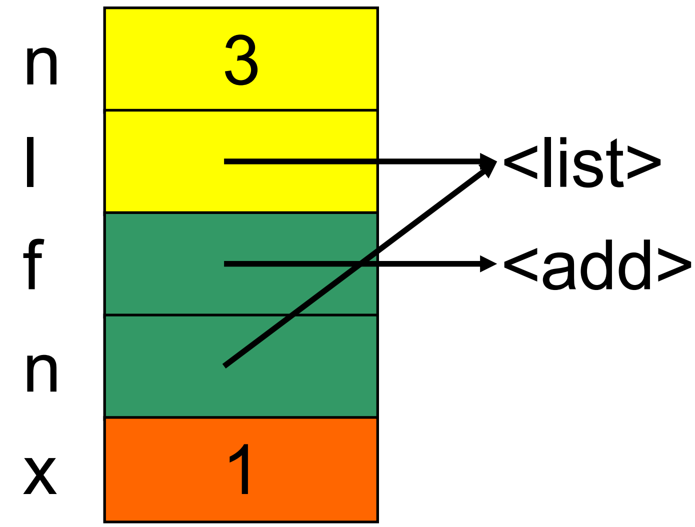
```
void f(void) {
  int x;
  x = g(3);
}
int g(int x) {
  int y;
  y = h(x);
  return y;
}
int h (int z) {
  return z + 1;
}
int main(){
  f();
  return 0;
}
```

x	4	**f**
x	3	**g**
y	4	
z	3	**h**

Now Consider Returning Functions

```
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]
```



- Uh oh...how does add know the value of n?
  - OCaml does *not* read it off the stack
    - The language could do this, but can be confusing (see above)
  - OCaml uses static scoping like C, C++, Java, and Ruby

Static Scoping (*aka* Lexical Scoping)

- In static or lexical scoping, (nonlocal) names refer to their nearest binding in the program text
  - Going from inner to outer scope
  - In our example, add refers to addN's n
  - C example:

Refers to the **x** at file scope – that's the nearest **x** going from inner scope to outer scope in the source code

```
int x;
void f() { x = 3; }
void g() { char *x = "hello"; f(); }
```

Closures Implement Static Scoping

- An environment is a mapping from variable names to values
  - Just like a stack frame

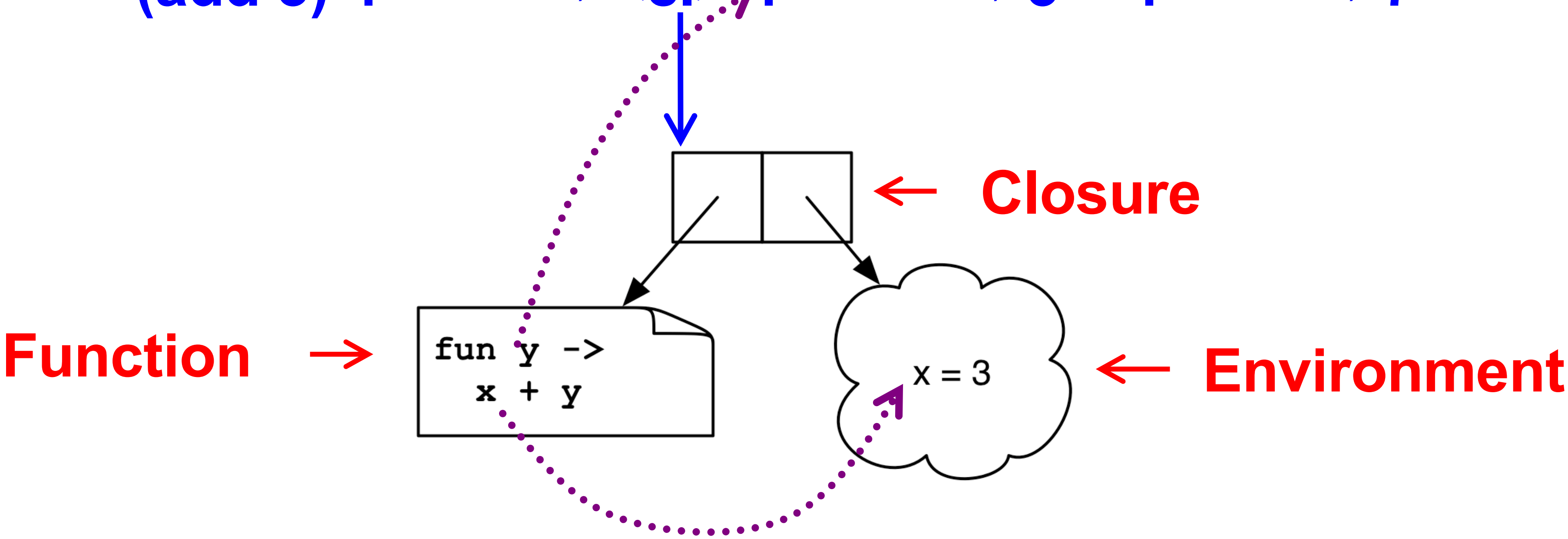
- A closure is a pair (f, e) consisting of function code f and an environment e

- 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)
```

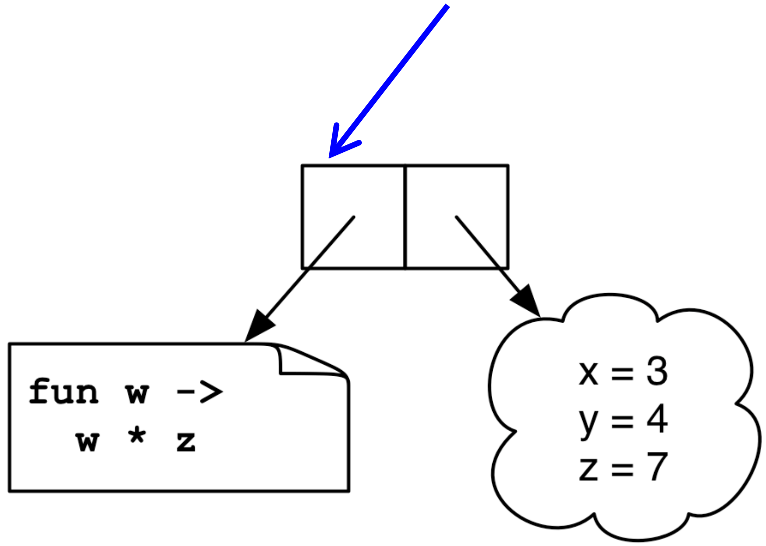
(add 3) 4 → <cl> 4 → 3 + 4 → 7



Example – Closure 2

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

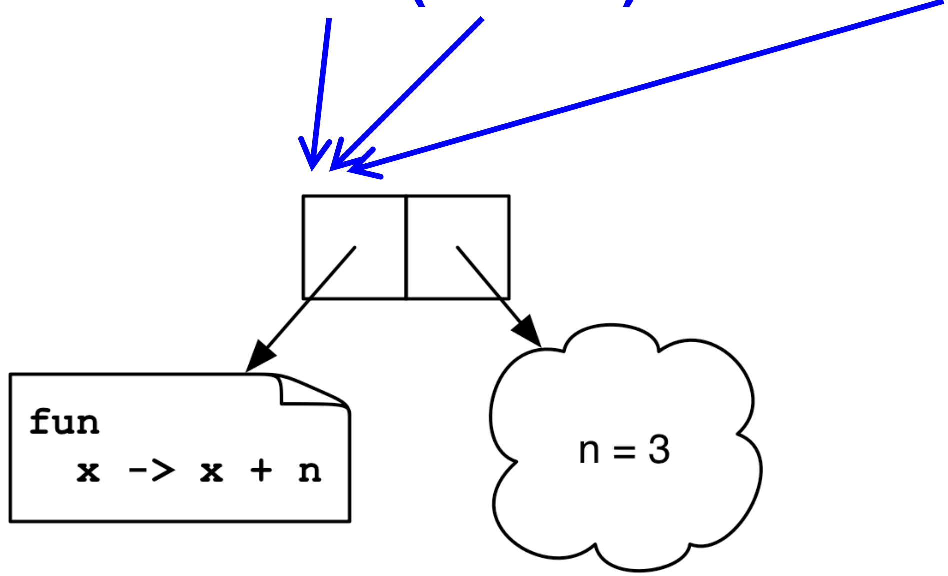
(mult_sum (3, 4)) 5 → <cl> 5 → 5 * 7 → 35



Example – Closure 3

```
let twice (n, y) =
  let f x = x + n in
    f (f y)
```

twice (3, 4) → **<cl> (<cl> 4)** → **<cl> 7** → **10**

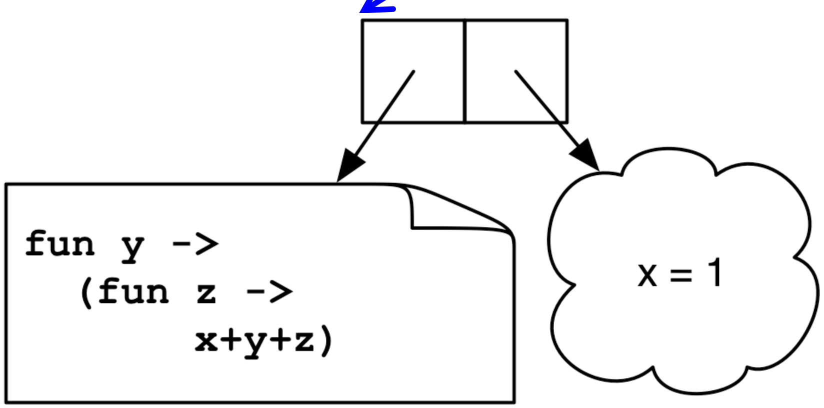


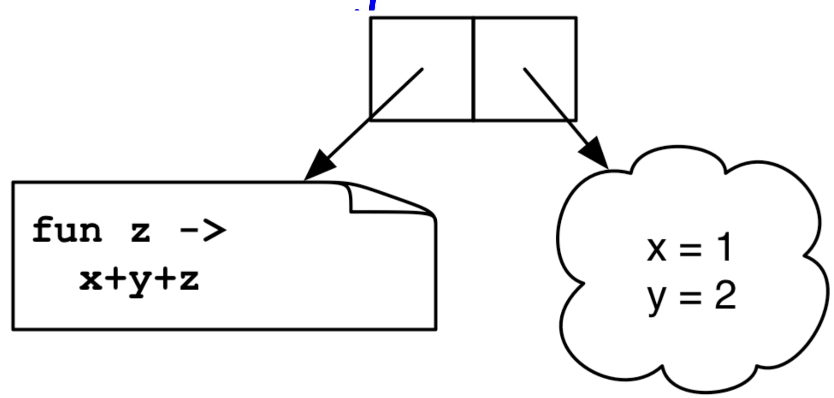
Example – Closure 4

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

add( ) took 3 arguments? The compiler figures this out and avoids making closures

(((add 1) 2) 3) →((<cl> 2) 3) →(<cl> 3) → 1+2+3





Quiz 3: What is x?

```
let a = 1;;
let a = 0;;
let b = 10;;
let f () = a + b;;
let b = 5;;
let x = f ();;
```

A. `10`

B. `1`

C. `15`

D. Error - variable name conflicts

Quiz 3: What is x?

```
let a = 1;;
let a = 0;;
let b = 10;;
let f () = a + b;;
let b = 5;;
let x = f ();;
```

A. `10`

B. `1`

C. `15`

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. `7`

B. `-2`

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. 7

B. -2

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) 1
```

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)) 1
```

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

```
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);
}
```

Higher-Order Functions in C (cont.)

- C does not support closures
  - Since no nested functions allowed
  - Unbound symbols always in global scope

```
int y = 1;
void app(int(*f)(int), n) {
  return f(n);
}
int add_y(int x) {
  return x + y;
}
int main() {
  app(add_y, 2);
}
```

Higher-Order Functions in C (cont.)

- Cannot access non-local variables in C
- OCaml code

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

- Equivalent code in C is illegal

```
int (* add(int x))(int) {
  return add_y;
}
int add_y(int y) {
  return x + y; /* error: x undefined */
}
```

Higher-Order Functions in C (cont.)

- OCaml code

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

- Works if C supports nested functions
  - Not in ISO C, but in gcc; but not allowed to return them

```
int (* add(int x))(int) {
  int add_y(int y) {
    return x + y;
  }
  return add_y; }
```

  - Does not allocate closure, so x popped from stack and add_y will get garbage (potentially) when called

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

Java 8 Example

```
public class Calculator {
    interface IntegerMath { int operation(int a, int b);  }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }
    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

Lambda expressions