Returned Functions

- As we saw, in OCaml a function can return another function as a result
  - So consider the following example

\[
\text{let addN n = (fun x -> x + n)} \quad \text{(addN 3) 4 (* returns 7 *)}
\]

- When the anonymous function is called, n isn’t even on the stack any more!
  - We need some way to keep n around after addN returns

Environments and Closures

- 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

\[
\text{let add x = (fun y -> x + y)}
\]

\[
\text{(add 3) 4 \rightarrow <cl> 4 \rightarrow 3 + 4 \rightarrow 7}
\]

Example – Closure 2

\[
\text{let mult_sum (x, y) =}
\]
\[
\text{let z = x + y in}
\]
\[
\text{fun w -> w * z}
\]

\[
\text{(mult_sum (3, 4)) 5 \rightarrow <cl> 5 \rightarrow 5 * 7 \rightarrow 35}
\]

Example – Closure 3

\[
\text{let twice (n, y) =}
\]
\[
\text{let f x = x + n in}
\]
\[
\text{f (f y)}
\]

\[
\text{twice (3, 4) \rightarrow <cl> (<cl> 4) \rightarrow <cl> 7 \rightarrow 10}
\]
Example – Closure 4

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

**Currying**

- We just saw another way for a function to take multiple arguments
  - The function consumes one argument at a time, creating closures until all the arguments are available

  ![Currying Diagram](image)

- This is called currying the function
  - Named after the logician Haskell B. Curry
  - But Schönfinkel and Frege discovered it
    - So it should probably be called Schönfinkelizing or Fregging

Curried Functions in OCaml

- OCaml has a really simple syntax for currying
  ```
  let add x y = x + y
  ```
  - This 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)
  ```

Curried Functions in OCaml (cont.)

- What is the type of add?
  ```
  let add x y = x + y
  ```

  ![Type Diagram](image)

  - Answer
    - `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

Curried Functions in OCaml (cont.)

- Currying is so common, OCaml uses the following conventions
  - `->` associates to the right
    - `int -> int -> int` is the same as `int -> (int -> int)`
  - Function application `( )` associates to the left
    - `add 3 4` is the same as `(add 3) 4`

Another Example of Currying

- A curried add function with three arguments
  ```
  let add_th x y z = x + y + z
  ```
  is the same as
  ```
  let add_th x = (fun y -> (fun z -> x+y+z))
  ```

  ![Another Currying Diagram](image)

  - Then...
    - `add_th` has type `int -> (int -> (int -> int))`
    - `add_th 4` has type `int -> (int -> int)`
    - `add_th 4 5` has type `int -> int`
    - `add_th 4 5 6` is 15
Recall Functions map & fold

- **Map**
  
  ```ocaml
  let rec map (f, l) = match l with
  | [] -> []
  | (h::t) -> (f h)::(map (f, t))
  ```

  - Type = ('a -> 'b) * 'a list -> 'b list

- **Fold**
  
  ```ocaml
  let rec fold (f, a, l) = match l with
  | [] -> a
  | (h::t) -> fold (f, f (a, h), t)
  ```

  - Type = ('a * 'b -> 'a) * 'a * 'b list -> 'a

Currying and the fold Function

- **New Fold**
  
  ```ocaml
  let rec fold f a l = match l with
  | [] -> a
  | (h::t) -> fold f (f a h) t
  ```

  - Examples
    
    ```ocaml
    let add x y = x + y
    fold add 0 [1; 2; 3] (* 6 *)
    let sum = fold add 0
    sum [1; 2; 3] (* 6 *)
    let next n = n + 1
    let len = fold next 0 (* len not polymorphic! *)
    len [4; 5; 6; 7; 8] (* 5 *)
    ```

  - What is the type of this form of fold?
    
    ```ocaml
    ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
    ```

Another Convention

- **New Map**
  
  ```ocaml
  let rec map f l = match l with
  | [] -> []
  | (h::t) -> (f h)::(map f t)
  ```

  - Examples
    
    ```ocaml
    let negate x = -x
    map negate [1; 2; 3] (* [-1; -2; -3] *)
    let negate_list = map negate
    negate_list [-1; -2; -3] (* [1; 2; 3] *)
    let sum_pair_l = map (fun (a, b) -> a + b)
    sum_pair_l[(1, 2); (3, 4)] (* [3; 7] *)
    ```

  - What is the type of this form of map?
    
    ```ocaml
    ('a -> 'b) -> 'a list -> 'b list
    ```

Another Convention (cont.)

- Instead of
  
  ```ocaml
  let rec map f l = match l with
  | [] -> []
  | (h::t) -> (f h)::(map f t)
  ```

  - It could be written
    
    ```ocaml
    let rec map f = function
    | [] -> []
    | (h::t) -> (f h)::(map f t)
    ```

Currying is Standard in OCaml

- Pretty much all functions are curried
  
  ```ocaml
  let rec map f l = match l with
  | [] -> []
  | (h::t) -> (f h)::(map f t)
  ```

  - Like the standard library map, fold, etc.
  - See /usr/local/ocaml/lib/ocaml on linuxlab
    
    ```ocaml
    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 plays a lot of tricks to avoid creating closures and to avoid allocating on the heap
  
  ```ocaml
  It's unnecessary much of the time, since functions are usually called with all arguments
  ```
Higher-Order Functions in C

- C supports function pointers

```c
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

```c
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

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

- Equivalent code in C is illegal

```c
int (*)(int x))(int) {
    return add_y;
}
int add_y(int y) {
    return x + y; // x undefined
}
```

Higher-Order Functions in Ruby

- Ruby supports higher-order functions
  - Use `yield` within method to call code block argument

```ruby
def my_collect(a)
    b = Array.new(a.length)
    0.upto(a.length-1) {
        |i|
        b[i] = yield(a[i])
    }
    return b
end
b = my_collect([5, 6, 7]) { |x| x+1 }
```

Higher-Order Functions in Ruby (cont.)

- Ruby supports closures
  - Code blocks can access non-local variables
  - Binding determined by lexical scoping

```ruby
def twice
    yield
    yield
end
x = 1
twice {x += 1}
puts x # 3
```

```ruby
def twice
    x = 0 #dynamic
    yield
    yield
end
x = 1 #lexical
twice {x += 1}
puts x # 3 not 1
```
Higher-Order Functions in Ruby (cont.)

- Ruby code blocks are actual variables
  ```ruby
  def twice
    yield # invoked with yield
    yield
  end
  twice { x += 1 } # same as x += 2
  ```

- Code blocks may be saved
  ```ruby
  def quad(&block)
    twice(&block) # used as argument
  end
  quad { x += 1 } # same as x += 2
  ```

Higher-Order Functions in Ruby (cont.)

- Ruby supports creating closures directly
  ```ruby
  Proc.new { x+=1 }
  proc { x+=1 }
  lambda { x+=1 }
  method
  ```

- An object in Java or C++ is kind of like a closure
  - It has some data (like an environment)
  - Along with some methods (i.e., function code)
  - So objects can be used to simulate closures

- So is an anonymous Java inner class
  - Inner class methods can access fields of outer class

- Back in CMSC 132 (OOP II)
  - We studied how to implement some functional patterns in OO languages