Anonymous Functions

- Recall code blocks in Ruby
  
  ```ruby
  (1..10).each { |x| print x }
  ```
  - Here, we can think of `{ |x| print x }` as a function

- We can do this (and more) in Ocaml
  
  ```ocaml
  range_each (1,10) (fun x -> print_int x)
  ```
  - where
  ```ocaml
  let rec range_each (i,j) f =
  if i > j then ()
  else
    let _ = f i in (* ignore result *)
    range_each (i+1,j) f
  ```

Anonymous Functions

- Passing functions around is very common
  - So often we don’t want to bother to give them names

- Use `fun` to make a function with no name

  ```ocaml
  fun x -> x + 3
  ```

  (fun x -> x + 3) 5 = 8

All Functions Are Anonymous

- Functions are first-class, so you can bind them to other names as you like
  ```ocaml
  let f x = x + 3
  let g = f
  g 5 = 8
  ```

- In fact, `let` for functions is just shorthand
  ```ocaml
  let f x = body
  ↓ stands for
  let f = fun x -> body
  ```
Examples

- let next x = x + 1
  - Short for let next = fun x -> x + 1

- let plus (x, y) = x + y
  - Short for let plus = fun (x, y) -> x + y
  - Which is short for
    - let plus = fun z ->
      (match z with (x, y) -> x + y)

- let rec fact n =
  - if n = 0 then 1 else n * fact (n-1)
  - Short for let rec fact = fun n ->
    (if n = 0 then 1 else n * fact (n-1))

Higher-Order Functions

- In OCaml you can pass functions as arguments, and return functions as results
  - let plus_three x = x + 3
  - let twice f z = f (f z)
  - twice plus_three 5
  - twice : ('a->'a) -> 'a -> 'a

  - let plus_four x = x + 4
  - let pick_fn n =
    - if n > 0 then plus_three else plus_four
    (pick_fn 5) 0
  - pick_fn : int -> (int->int)

Currying

- We just saw a way for a function to take multiple arguments
  - The function consumes one argument at a time, returning a function that takes the rest

- 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)

  - 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
Curried Functions In OCaml (cont.)

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

Mental Shorthand

- You can think of curried types as defining multi-argument functions
  - Type `int -> float -> float` is a function that takes an `int` and a `float` and returns a `float`
  - Type `int -> int -> int -> int` is a function that takes three `ints` and returns an `int`

- The bonus is that you can partially apply the function to some of its arguments
  - And apply that to the rest of the arguments later

Another Example Of Currying

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

- 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`

Implementing this is Challenging!

- Implementing functions that return other functions requires a clever data structure called a closure
  - We'll see how these are implemented later

- In the meantime, we will explore using higher order functions, and then discuss how they are implemented
The Map Function

- Let’s write the map function (like Ruby’s collect)
  - Takes a function and a list, applies the function to each element of the list, and returns a list of the results

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

```
let add_one x = x + 1
let negate x = -x
map add_one [1; 2; 3]
map negate [9; -5; 0]
```

- Type of map?

The Map Function (cont.)

- What is the type of the map function?

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

```
let add_one x = x + 1
let negate x = -x
map add_one [1; 2; 3]
map negate [9; -5; 0]
```

Pattern Matching With Fun

- match can be used within fun

```
map (fun l -> match l with (h::_) -> h)
  [ [1; 2; 3]; [4; 5; 6; 7]; [8; 9] ]
  = [1; 4; 8]
```

- But use named functions for complicated matches

```
map (fun (x, y) -> x+y) [(1,2); (3,4)]
  = [3; 7]
```

The Fold Function

- Common pattern
  - Iterate through list and apply function to each element, keeping track of partial results computed so far

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

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

- a = “accumulator”
- Usually called fold left to remind us that f takes the accumulator as its first argument

- What’s the type of fold?

```
= ('a -> 'b list -> 'a) -> 'a list -> 'b list
```

**Example**

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

```plaintext
let add a x = a + x
fold add 0 [1; 2; 3; 4] →  
fold add 1 [2; 3; 4] →  
fold add 3 [3; 4] →  
fold add 6 [4] →  
fold add 10 [] →  
10
```

We just built the **sum** function!

---

**Another Example**

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

```plaintext
let next a _ = a + 1
fold next 0 [2; 3; 4; 5] →  
fold next 1 [3; 4; 5] →  
fold next 2 [4; 5] →  
fold next 3 [5] →  
fold next 4 [] →  
4
```

We just built the **length** function!

---

**Using Fold to Build Reverse**

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

```plaintext
let prepend a x = x::a
fold prepend [] [1; 2; 3; 4] →  
fold prepend [1] [2; 3; 4] →  
fold prepend [2; 1] [3; 4] →  
fold prepend [3; 2; 1] [4] →  
fold prepend [4; 3; 2; 1] [] →  
[4; 3; 2; 1]
```

Can you build the **reverse** function with **fold**?

---

**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
  - Otherwise it would do a lot of useless allocation of closures (which we see later) when all arguments are provided
A Convention

Since functions are curried, function can often be used instead of match
- function declares an anonymous function of one argument
- Instead of

```
let rec sum l = match l with
  [] -> 0
| (h::t) -> h + (sum t)
```
- It could be written

```
let rec sum = function
  [] -> 0
| (h::t) -> h + (sum t)
```

A Convention (cont.)

Instead of

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

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

Nested Functions

In OCaml, you can define functions anywhere
- Even inside of other functions

```
let pick_one n =
  if n > 0 then (fun x -> x + 1)
  else (fun x -> x - 1)
(pick_one -5) 6 (* returns 5 *)
```

let sum l =
  fold (fun a x -> a + x) 0 l

Nested Functions (cont.)

You can also use let to define functions inside of other functions

```
let pick_one n =
  if n > 0 then add_one
  else sub_one
```

```
let sum l =
  let add a x = a + x in
  fold add 0 l
```

```
let pick_one n =
  let add_one x = x + 1 in
  let sub_one x = x - 1 in
  if n > 0 then add_one else sub_one
```
How About This?

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

- (Equivalent to...)

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

Within a function, variables are accessible from the outer scope by using lambda expressions. This allows for the creation of new functions on the fly, as demonstrated by the `addN` function above.

Returned Functions

In OCaml, a function can return another function as a result; this is what currying is doing. Consider the following example:

```ocaml
let addN n = (fun x -> x + n) (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.

The Call Stack in C/Java/etc.

```c
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;
}
```

```plaintext
x 4 f
x 3 g
y 4 add
z 3 h
```

Now Consider Returning Functions

```ocaml
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
```

```plaintext
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:

```c
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

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

\((\text{add } 3) 4\) → \(<\text{cl}>4\) → \(3 + 4\) → \(7\)

Example – Closure 2

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

\((\text{mult}_\text{sum} (3, 4)) 5\) → \(<\text{cl}>5\) → \(5 * 7\) → \(35\)
Example – Closure 3

let twice (n, y) =
    let f x = x + n in
    f (f y)
twice (3, 4) → <cl> (4) → <cl> 7 → 10

Example – Closure 4

let add x = (fun y -> (fun z -> x + y + z))
(add 1) 2) 3) → (3) → 1+2+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

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

- Equivalent code in C is illegal

```c
int (* add(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 C (cont.)

- OCaml code

```ocaml
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

```c
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

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  # implicit block
  yield   # invoked with yield
  yield
end

twice { x += 1 }  # same as x += 2
```

- Code blocks may be saved

```ruby
def quad (&block)  # explicit block
  twice (&block)   # used as argument
  twice (&block)
end

quad { x += 1 }  # same as x += 4
```

Higher-Order Functions in Ruby (cont.)

- Ruby supports creating closures directly

- Proc.new
- proc
- lambda
- method

```ruby
c1 = Proc.new { x+=1 }
c2 = proc   { x+=1 }
c3 = lambda { x+=1 }
def foo
  x+=1
end
c4 = method { :foo }

c.call  # x+=1
```

Higher-Order Functions in Java/C++

- 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
Java 8 Supports Lambda Expressions

- Ocaml's
  
  ```
  function (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

```java
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));
    }
}
```