CS 320 : Functional Programming in Ocaml

(based on slides from David Walker, Princeton, Lukasz Ziarek, Buffalo and myself.)

Marco Gaboardi

MSC 116

gaboardi@bu.edu

Announcements

- Homework ***Assignment #1*** is due Tuesday, February 11 (**Today!**), no later than 11:59 pm.
- Homework ***Assignment #2*** is posted Tuesday, February 11 (**Today!**), and due Friday, February 21.
- Please do not wait until February 19-20 to start working on homework ***Assignment #2***!

Learning Goals for Today

- More on functional programming:
  - *more complex functions*
  - *higher-order functions*
  - *code factoring*
  - *currying*
- More on polymorphism

Functional programming

Factoring Code in OCaml

Consider these definitions:

```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```



```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```

Factoring Code in OCaml

Consider these definitions:

```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```



```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```


The code is almost identical – factor it out!

Factoring Code in OCaml

A *higher-order* function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```

Factoring Code in OCaml

A *higher-order* function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```


Uses of the function:

```
let inc x = x+1
let inc_all xs = map inc xs
```

Factoring Code in OCaml

A *higher-order* function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```


Uses of the function:

```
let inc x = x+1
let inc_all xs = map inc xs

let square y = y*y
let square_all xs = map square xs
```

Writing little functions like inc just so we call map is a pain.

Factoring Code in OCaml

A higher-order function captures the recursion pattern:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```

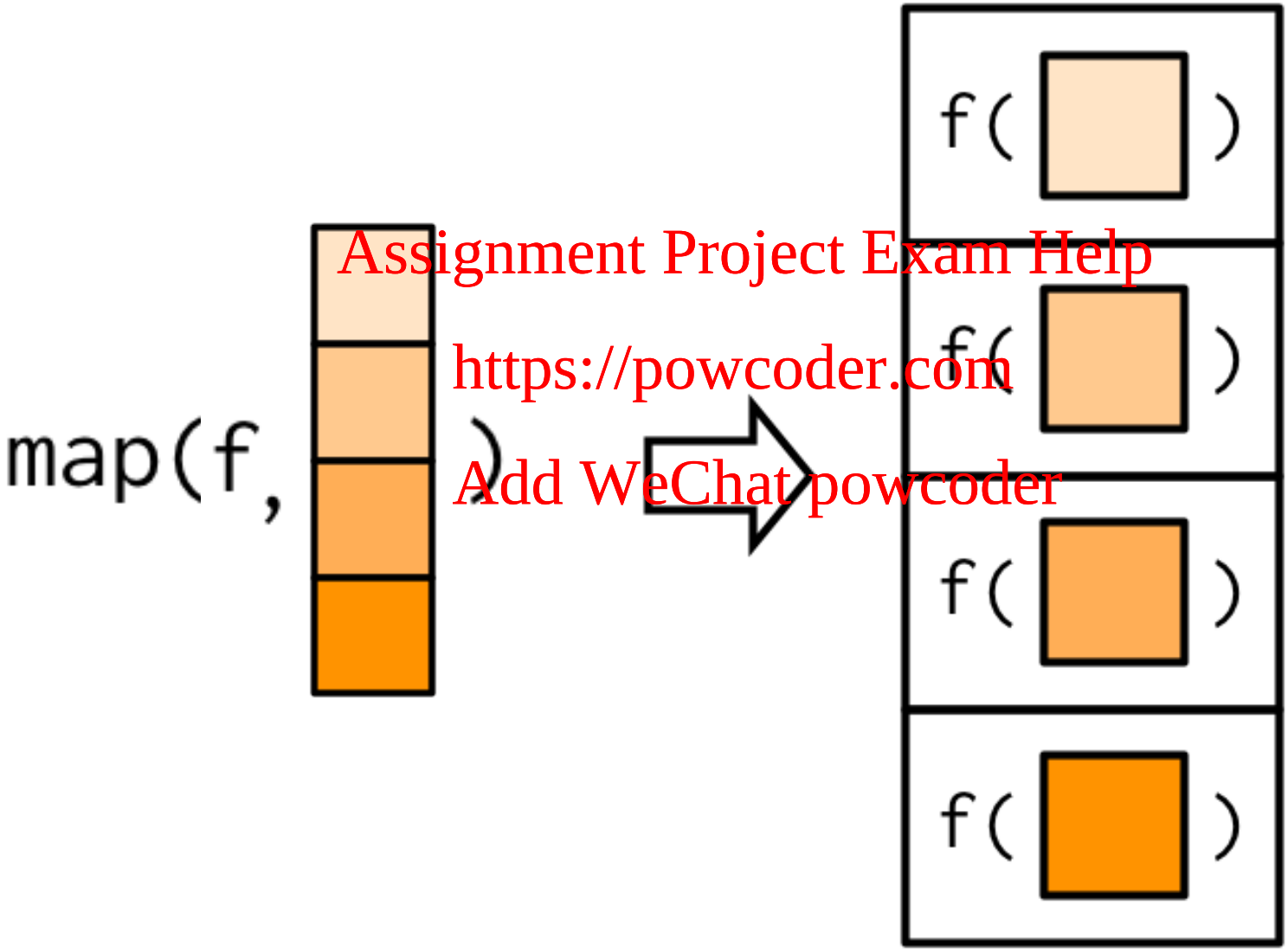

We can use an *anonymous* function instead.

Originally, Church wrote this function using λ instead of **fun**: (λx. x+1) or (λx. x*x)

Uses of the function:

```
let inc_all xs = map (fun x -> x + 1) xs

let square_all xs = map (fun y -> y * y) xs
```



TopHat Q1-Q8

Another example

```
let rec sum (xs:int list) : int =
  match xs with
  | [] -> 0
  | hd::tl -> hd + (sum tl)

let rec prod (xs:int list) : int =
  match xs with
  | [] -> 1
  | hd::tl -> hd * (prod tl)
```


Goal: Create a function called reduce that when supplied with a few arguments can implement both sum and prod. Define sum2 and prod2 using reduce.

(Try it)

Goal: If you finish early, use map and reduce together to find the sum of the squares of the elements of a list.

(Try it)

Another example

```
let rec sum (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> hd + (sum tl)

let rec prod (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> hd * (prod tl)
```

Another example

```
let rec sum (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> hd OP (RECURSIVE CALL ON tl)

let rec prod (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> hd OP (RECURSIVE CALL ON tl)
```

Another example

```
let rec sum (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (RECURSIVE CALL ON tl)

let rec prod (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (RECURSIVE CALL ON tl)
```

A generic reducer

```
let add x y = x + y
let mul x y = x * y

let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce add 0 xs
let prod xs = reduce mul 1 xs
```

A generic reducer

```
let add x y = x + y
let mul x y = x * y

let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce add 0 xs
let prod xs = reduce mul 1 xs
```

```
reduce add 0 [1;2;3] ⇒
add 1 (reduce add 0 [2;3])  ⇒
add 1 (add 2 (reduce add 0 [3]))  ⇒
add 1 (add 2 (add 3 (reduce add 0 [])))  ⇒
add 1 (add 2 (add 3 0))  ⇒
add 1 (add 2 3)  ⇒
add 1 5  ⇒
6
```

Using Anonymous Functions

```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce (fun x y -> x+y) 0 xs
let prod xs = reduce (fun x y -> x*y) 1 xs
```

Using Anonymous Functions

```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce (fun x y -> x+y) 0 xs
let prod xs = reduce (fun x y -> x*y) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```

Using Anonymous Functions

```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce (+) 0 xs
let prod xs = reduce ( * ) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```

Using Anonymous Functions

```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce (+) 0 xs
let prod xs = reduce (*) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```


wrong

Using Anonymous Functions

```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

let sum xs = reduce (+) 0 xs
let prod xs = reduce (*) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```


wrong -- creates a comment! ug. OCaml -0.1

More on Anonymous Functions

Function declarations:

```
let square x = x*x

let add x y = x+y
```


are *syntactic sugar* for:


```
let square = (fun x -> x*x)
```



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

In other words, *functions are values* we can bind to a variable, just like 3 or “moo” or true.

Functions are 2nd class no more!

One argument, one result

Simplifying further:

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

is shorthand for:


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


That is, add is a function which:

- when given a value x, *returns a function* (fun y -> x+y) which:
  - when given a value y, returns x+y.

Curried Functions

Currying: *verb. gerund or present participle*

(1) to prepare or flavor with hot-tasting spices

(2) to encode a multi-argument function using nested, higher-order functions.

(1)

(2)

```
fun x -> (fun y -> x+y) (* curried *)
fun x y -> x + y        (* curried *)
fun (x,y) -> x+y        (* uncurried *)
```

What is the type of add?

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

Add's type is:


```
int -> (int -> int)
```


which we can write as:

```
int -> int -> int
```

That is, the arrow type is right-associative.

What's so good about Currying?

In addition to simplifying the language, currying functions so that they only take one argument leads to two major wins:

1. We can *partially apply* a function.
2. We can more easily *compose* functions.

Partial Application

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

Curried functions allow defs of new, *partially applied* functions:

```
let inc = add 1
```


Equivalent to writing:


```
let inc = (fun y -> 1+y)
```


which is equivalent to writing:

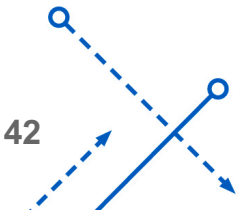
```
let inc y = 1+y
```

also:

```
let inc2 = add 2
let inc3 = add 3
```

TopHat Q9-Q12

Polymorphism



Here's an annoying thing

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl);;
```


What if I want to increment a list of floats?

Alas, I can't just call this map. It works on ints!

Here's an annoying thing

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl);;
```


What if I want to increment a list of floats?

Alas, I can't just call this map. It works on ints!


```
let rec mapfloat (f:float->float) (xs:float list) :
            float list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(mapfloat f tl);;
```

Turns out

```
let rec map f xs =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)

let ints = map (fun x -> x + 1) [1; 2; 3; 4]

let floats = map (fun x -> x +. 2.0) [3.1415; 2.718]

let strings = map String.uppercase [“sarah”; “joe”]
```

Type of the undecorated map?

```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```



```
map : ('a -> 'b) -> 'a list -> 'b list
```

Type of the undecorated map?

```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

map : ('a -> 'b) -> 'a list -> 'b lis
```


We often use greek letters like α or β to represent type variables.

Read as:

- for any types 'a and 'b,
- if you give map a function from 'a to 'b,
- it will return a function
  - which when given a list of 'a values
  - returns a list of 'b values.

We can say this explicitly

```
let rec map (f:'a -> 'b) (xs:'a list) : 'b list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)

map : ('a -> 'b) -> 'a list -> 'b list
```


The OCaml compiler is smart enough to figure out that this is the *most general* type that you can assign to the code.

We say map is *polymorphic* in the types 'a and 'b – just a fancy way to say map can be used on any types 'a and 'b.

Java generics derived from ML-style polymorphism (but added after the fact and more complicated due to subtyping)

More realistic polymorphic functions

```
let rec merge (lt:'a->'a->bool) (xs:'a list) (ys:'a list) : 'a list =
  match (xs,ys) with
  | ([],_) -> ys
  | (_,[]) -> xs
  | (x::xst, y::yst) ->
      if lt x y then x::(merge lt xst ys)
      else y::(merge lt xs yst)

let rec split (xs:'a list) (ys:'a list) (zs:'a list) : 'a list * 'a list =
  match xs with
  | [] -> (ys, zs)
  | x::rest -> split rest zs (x::ys)

let rec mergesort (lt:'a->'a->bool) (xs:'a list) : 'a list =
  match xs with
  | ([] | _::[]) -> xs
  | _ -> let (first,second) = split xs [] [] in
         merge lt (mergesort lt first) (mergesort lt second)
```

More realistic polymorphic functions

```
mergesort : ('a->'a->bool) -> 'a list -> 'a list

mergesort (<) [3;2;7;1]
  == [1;2;3;7]

mergesort (>) [2; 3; 42]
  == [42 ; 3; 2]

mergesort (fun x y -> String.compare x y < 0) [“Hi”; “Bi”]
  == [“Bi”; “Hi”]

let int_sort = mergesort (<)
let int_sort_down = mergesort (>)
let str_sort = mergesort (fun x y -> String.compare x y < 0)
```

Another Interesting Function

```
let comp f g x = f (g x)

let mystery = comp (add 1) square
```



```
let comp = fun f -> (fun g -> (fun x -> f (g x)))

let mystery = comp (add 1) square
```



```
let mystery =
  (fun f -> (fun g -> (fun x -> f (g x)))) (add 1) square
```

```
let mystery =              fun x -> (add 1) (square x)
```

```
let mystery x = add 1 (square x)
```