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

- Solution for homework ***Assignment #1*** will be posted on Piazza.
- Homework ***Assignment #2*** is already posted and will be due on Friday, February 21, by 11:59 pm.
- Please do not wait until February 19-20 to start working on homework ***Assignment #2***!

TopHat Q0.1 – Q0.3

Learning Goals for Today

- More on functional programming:
  - *more complex functions*
  - *higher-order functions*
  - *code factoring*
  - *anonymous functions*
  - *currying*
- More on polymorphism
- Data types, inductive data types

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

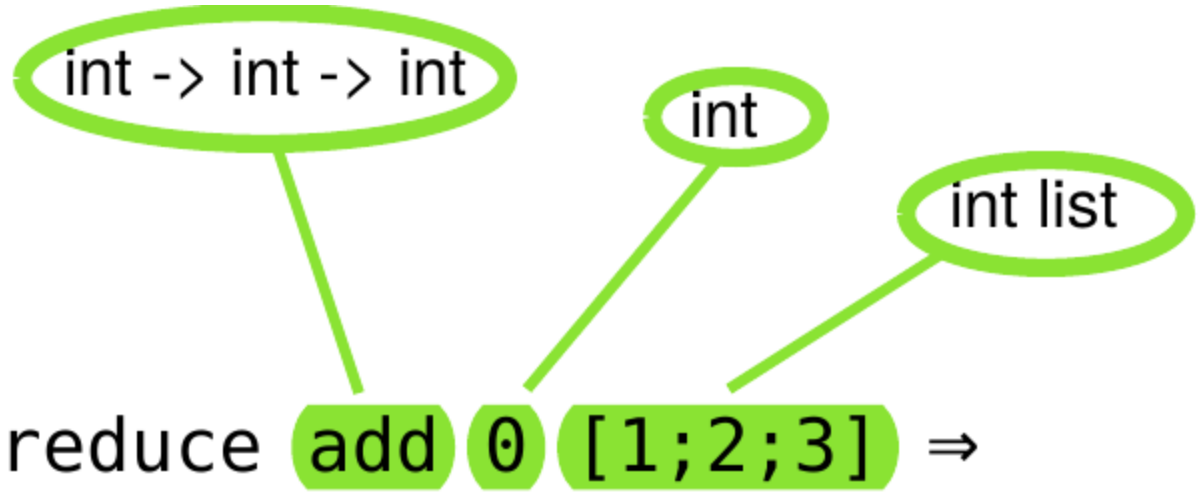
```
let rec sum (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (RECURSIVE CALL ON tl)

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

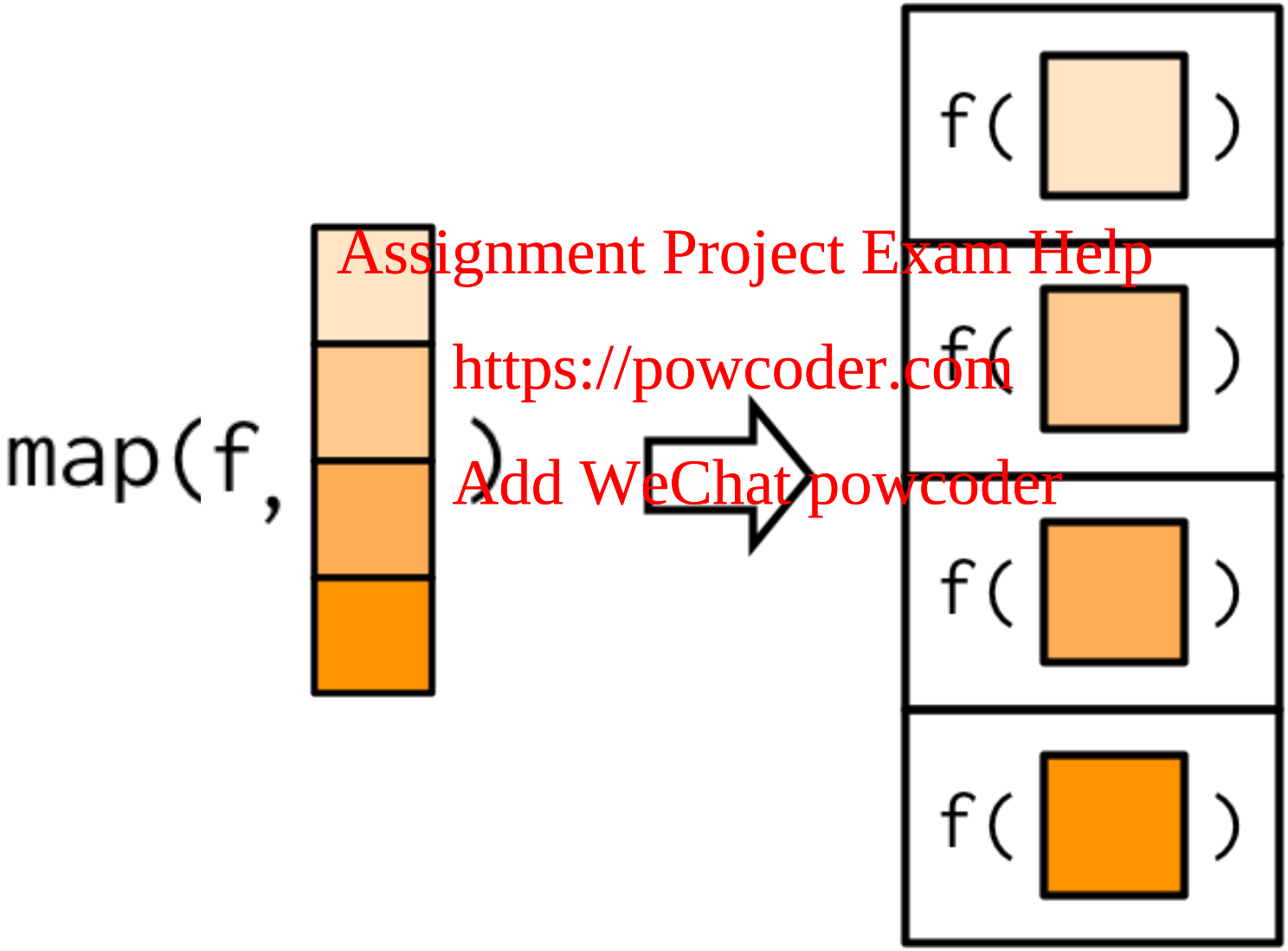
```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

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
```



TopHat Q1-Q6

Data Types

OCaml So Far

- We have seen a number of basic types:
  - int
  - float
  - char
  - string
  - bool
- We have seen a few structured types:
  - pairs
  - tuples
  - options
  - lists
- In this lecture, we will see some more general ways to define our own new types and data structures

Type Abbreviations

- We have already seen some type abbreviations:

```
type point = float * float
```

Type Abbreviations

- We have already seen some type abbreviations:

```
type point = float * float
```


- These abbreviations can be helpful documentation:


```
let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```


- But they add nothing of *substance* to the language
  - they are equal in every way to an existing type

Type Abbreviations

- We have already seen some type abbreviations:

```
type point = float * float
```


- As far as OCaml is concerned, you could have written:


```
let distance (p1:float*float)
             (p2:float*float) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

- Since the types are equal, you can *substitute* the definition for the name wherever you want
  - we have not added any new data structures

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```


a value with type my_bool is one of two things:

- Tru, or
- Fal

read the "|" as "or"

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```


Tru and Fal are called "constructors"

a value with type my_bool is one of two things:

- Tru, or
- Fal

read the "|" as "or"

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```

```
type color = Blue | Yellow | Green | Red
```


there's no need to stop at 2 cases; define as many alternatives as you want

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```

```
type color = Blue | Yellow | Green | Red
```


- Creating values:

```
let b1 : my_bool = Tru
let b2 : my_bool = Fal
let c1 : color = Yellow
let c2 : color = Red
```

use constructors to create values

Data types

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```


- Using data type values:


```
let print_color (c:color) : unit =
  match c with
   | Blue ->
   | Yellow ->
   | Green ->
   | Red ->
```


use pattern matching to determine which color you have; act accordingly

Data types

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```


- Using data type values:


```
let print_color (c:color) : unit =
  match c with
   | Blue -> print_string "blue"
   | Yellow -> print_string "yellow"
   | Green -> print_string "green"
   | Red -> print_string "red"
```

Data types

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```


- Using data type values:


```
let print_color (c:color) : unit =
  match c with
   | Blue -> print_string "blue"
   | Yellow -> print_string "yellow"
   | Green -> print_string "green"
   | Red -> print_string "red"
```


Why not just use strings to represent colors instead of defining a new type?

Data types

```
type color = Blue | Yellow | Green | Red
```

oops!:

```
let print_color (c:color) : unit =
  match c with
   | Blue -> print_string "blue"
   | Yellow -> print_string "yellow"
   | Red -> print_string "red"
```


Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

Data types

```
type color = Blue | Yellow | Green | Red
```

oops!:

```
let print_color (c:color) : unit =
  match c with
   | Blue -> print_string "blue"
   | Yellow -> print_string "yellow"
   | Red -> print_string "red"
```


Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

OCaml's datatype mechanism allow you to create types that contain *precisely* the values you want!

TopHat Q7-Q9

Inductive Data Types

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree


```
type key = string
type value = int
```



```
type tree =
  Leaf
| Node of key * value * tree * tree
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf ->
  | Node (k', v', left, right) ->
```

Again, the type definition specifies the cases you must consider

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types: Another Example

- Recall, we used the type "int" to represent natural numbers
  - but that was kind of broken: it also contained negative numbers
  - we had to use a dynamic test to guard entry to a function:

```
let double (n : int) : int =
  if n < 0 then
    raise (Failure "negative input!")
  else
    double_nat n
```


  - it would be nice if there was a way to define the natural numbers exactly, and use OCaml's type system to guarantee no client ever attempts to double a negative number

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:


```
type nat = Zero | Succ of nat
```

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:


```
type nat = Zero | Succ of nat
```



```
let rec nat_to_int (n : nat) : int =
 match n with
   Zero -> 0
 | Succ n -> 1 + nat_to_int n
```

```
type ('key, 'val) tree =
    Leaf
  | Node of 'key * 'val * ('key, 'val) tree * ('key, 'val) tree


type 'a stree = (string, 'a) tree

type sitree = int stree
```


General form:

definition:

```
type 'x f = body
```

use:

```
arg f
```

A more conventional notation would have been (but is not ML):

definition:

```
type f x = body
```

use:

```
f arg
```

TopHat Q10-Q13

What is the type of comp?

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

What is the type of comp?

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



```
comp : ('b -> 'd) ->
       ('a -> 'b) ->
       ('a -> 'c)
```

How about reduce?

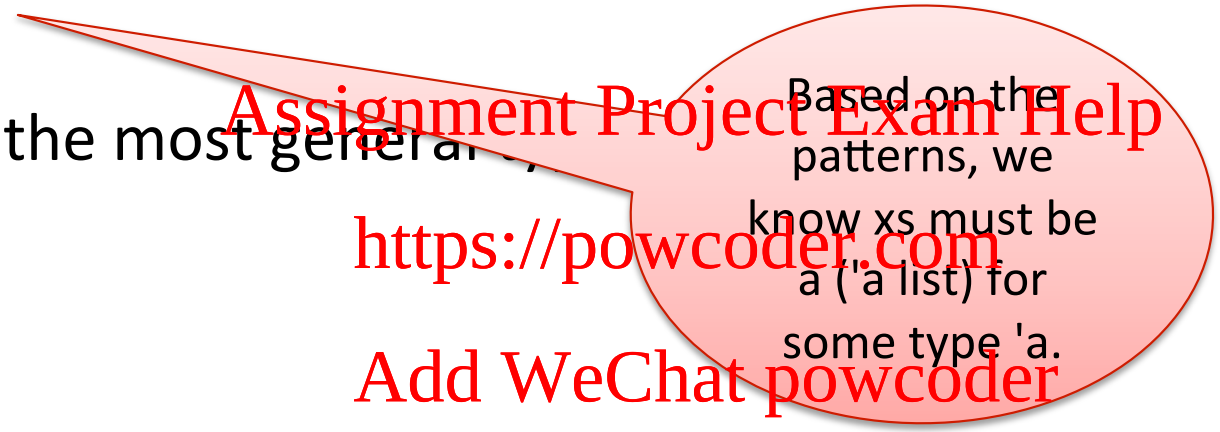
```
let rec reduce f u xs =
   match xs with
   | [] -> u
   | hd::tl -> f hd (reduce f u tl)
```


What's the most general type of reduce?

How about reduce?

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

What's the most general ...



How about reduce?

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

What's the most general type of reduce?

How about reduce?

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

What's the most general type of reduce?

f is called so it must be a function of two arguments.

How about reduce?

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

What's the most general type of reduce?

How about reduce?

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

What's the most general type of reduce?

Furthermore, hd came from xs, so f must take an 'a value as its first argument.

How about reduce?

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

What's the most general type of reduce?

How about reduce?

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

What's the most general type of reduce?

The second argument to f must have the same type as the result of reduce. Let's call it 'b.

How about reduce?

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

What's the most general type of reduce?

The result of f must have the same type as the result of reduce overall: 'b.

How about reduce?

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

What's the most general type of reduce?

How about reduce?

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

What's the most general type of reduce?

If xs is empty, then reduce returns u. So u's type must be 'b.

How about reduce?

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

What's the most general type of reduce?

How about reduce?

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


What's the most general type of reduce?


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