Poly-HO!

COS 326
David Walker
Princeton University

polymorphic, higher-order programming
Some Design & Coding Rules
Some Design & Coding Rules

• *Laziness* can be a really good force in design.

• Never write the same code twice.
  – factor out the common bits into a re-usable procedure.
  – better, use someone else’s (well-tested, well-documented, and well-maintained) procedure.

• Why is this a good idea?
  – why don’t we just cut-and-paste snippets of code using the editor instead of abstracting them into procedures?
Some Design & Coding Rules

• **Laziness** can be a really good force in design.

• Never write the same code twice.
  – factor out the common bits into a re-usable procedure.
  – better, use someone else’s (well-tested, well-documented, and well-maintained) procedure.

• Why is this a good idea?
  – why don’t we just cut-and-paste snippets of code using the editor instead of abstracting them into procedures?
  – find and fix a bug in one copy, have to fix in all of them.
  – decide to change the functionality, have to track down all of the places where it gets used.
Factoring Code in OCaml

Consider these definitions:

```ocaml
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:

```ocaml
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);;
```
A higher-order function captures the recursion pattern:

```ocaml
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:

```ocaml
let inc x = x+1;;
let inc_all xs = map inc xs;;
```
Factoring Code in OCaml

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

```ocaml
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:

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

let square y = y*y;;
let square_all xs = map square xs;;
```
Factoring Code in OCaml

A higher-order function captures the recursion pattern:

```ocaml
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:

```ocaml
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:

```ocaml
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:

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

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

We can use an anonymous function instead.

Originally, Church wrote this function using λ instead of fun:

\((\lambda x. x+1)\) or \((\lambda x. x^2)\)
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)
let add x y = x + y;;
let mul x y = x * y;;

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

let sum xs = reduce add 0 xs ;;
let prod xs = reduce mul 1 xs ;;
Using Anonymous Functions

```ocaml
let rec reduce (f:int->int->int) (u:int) (xs:int list) : int =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u 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) (u:int) (xs:int list) : int =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u 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
More on Anonymous Functions

Function declarations are actually abbreviations:

```
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)
```

So, *fun’s* are values we can bind to a variable, just like 3 or “moo” or true.

OCaml obeys the *principle of orthogonal language design*. 
Simplifying further:

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

is shorthand for:

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

That is, add is a function which:

– when given a value x, \textit{returns a function} \((\text{fun y -> x+y})\) which:
  • when given a value y, returns x+y.
Curried Functions

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

Currying: encoding a multi-argument function using nested, higher-order functions.

Named after the logician Haskell B. Curry.
– was trying to find minimal logics that are powerful enough to encode traditional logics.
– much easier to prove something about a logic with 3 connectives than one with 20.
– the ideas translate directly to math (set & category theory) as well as to computer science.
– (actually, Curry ripped off Moses Schönfinkel)
– (thankfully, we don't have to talk about Schönfinkelled functions)
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 (orthogonal design), 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.
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;;
```
SIMPLE REASONING ABOUT HIGHER-ORDER FUNCTIONS
Reasoning About Definitions

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

let square_all = map square;;
```

**Fundamental question**: How can I rewrite these definitions so my program is simpler, easier to understand, more concise, can be refactored, ...

I want some *rules* for doing so that never fail.
Simple Equational Reasoning

Rewrite 1 (Function de-sugaring):

```
let f x = body
```

```
let f = (fun x -> body)
```

Rewrite 2 (Substitution):

```
(fun x -> ... x ...) arg
```

```
... arg ...
```

Rewrite 3 (Eta-expansion):

```
let f = def
```

```
let f x = (def) x
```

- if f has a function type
- chose name x wisely so it does not shadow other names used in def

if arg is a value or, when executed, will always terminate without effect and produce a value
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl);;
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl);;

let rec map =
  (fun f ->
    (fun xs ->
      match xs with
      | [] -> []
      | hd::tl -> (f hd)::(map f tl))));;
Consider square_all

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

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

let square_all =
  (fun f ->
   (fun xs ->
    match xs with
    | [] -> []
    | hd::tl -> (f hd)::(map f tl)
   ) square ;;
let rec map =
    (fun f ->
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (f hd)::(map f tl)));

let square_all =
    (fun f ->
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (f hd)::(map f tl)
        ) square ;;
let rec map =
  (fun f ->
   (fun xs ->
    match xs with
    | []  -> []
    | hd::tl -> (f hd)::(map f tl))));;

let square_all =
  (fun f ->
   (fun xs ->
    match xs with
    | []  -> []
    | hd::tl -> (f hd)::(map f tl))
    ) square;;
let rec map =
    (fun f ->
      (fun xs ->
        match xs with
        | [] -> []
        | hd::tl -> (f hd)::(map f tl))));;

let square_all =
    (fun xs ->
      match xs with
      | [] -> []
      | hd::tl -> (square hd)::(map square tl)
    )
    ;;
let rec map =
  (fun f ->
    (fun xs ->
      match xs with
      | [] -> []
      | hd::tl -> (f hd)::(map f tl))));;

let square_all ys =
  (fun xs ->
    match xs with
    | [] -> []
    | hd::tl -> (square hd)::(map square tl)
  ) ys
;;

add argument via eta-expansion
let rec map =
    (fun f ->
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (f hd)::(map f tl))));;

let square_all ys =
    match ys with
    | [] -> []
    | hd::tl -> (square hd)::(map square tl)
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl);

let square_all xs = map square xs

let square_all ys =
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(map square tl);

let square_all ys =
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(map square tl);

proof by simple rewriting unrolls definition once
proof by induction eliminates recursive function map
What Happened?

We saw this:

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

let square_all ys = map square
```

Is equivalent to this:

```ocaml
let square_all ys =
    match ys with
    | [] -> []
    | hd::tl -> (square hd)::(map square tl);
```

Moral of the story

• (1) OCaml’s **HOT** (higher-order, typed) functions capture recursion patterns
• (2) we can figure out what is going on by **equational reasoning**.
• (3) ... but we typically need to do **proofs by induction** to reason about recursive (inductive) functions
POLY-HO!
Here’s an annoying thing

What if I want to increment a list of floats?
Alas, I can’t just call this map. It works on ints!

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl);;
```
Here’s an annoying thing

What if I want to increment a list of floats?
Alas, I can’t just call this map. It works on ints!

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

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

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

map (fun x -> x +. 2.0) [3.1415; 2.718; 42.0] ;;

map String.uppercase ["greg"; "victor"; "joe"] ;;
Type of the undecorated map?

```ocaml
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 list
```

We often use greek letters like $\alpha$ or $\beta$ 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

```ocaml
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

```ocaml
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 (≥) [2.718; 3.1415; 42.0]
  == [42.0 ; 3.1415; 2.718]

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) ;;
```ocaml
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) ;;
```
What does this program do?

\[
\text{map } f \left( \text{map } g \ [x_1; x_2; \ldots; x_n] \right)
\]

For each element of the list \(x_1, x_2, x_3 \ldots x_n\), it executes \(g\), creating:

\[
\text{map } f \left( [g \ x_1; g \ x_2; \ldots; g \ x_n] \right)
\]

Then for each element of the list \([g \ x_1, g \ x_2, g \ x_3 \ldots g \ x_n]\), it executes \(f\), creating:

\[
[f \ (g \ x_1); f \ (g \ x_2); \ldots; f \ (g \ x_n)]
\]

Is there a faster way? Yes! (And query optimizers for SQL do it for you.)

\[
\text{map } (\text{comp } f \ g) \ [x_1; x_2; \ldots; x_n]
\]
What is the type of comp?

```ocaml
let comp f g x = f (g x) ;;
```
What is the type of `comp`?

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

```
comp : (char -> char) ->
     (char -> char) ->
     (char -> char)
```
How about reduce?

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

What’s the most generally...

Based on the patterns, we know xs must be a (‘a list) for some type ‘a.
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?

```ocaml
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?

```ocaml
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?

```ocaml
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?

```ocaml
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?

```ocaml
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?

```ocaml
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?

```ocaml
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?

If `xs` is empty, then `reduce` returns `u`. So `u`’s type must be `‘b`. 
How about reduce?

```ocaml
define 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?
How about reduce?

```ocaml
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
The List Library

NB: map and reduce are already defined in the List library.
   – However, reduce is called “fold_right”.
   – (Good bet there’s a “fold_left” too.)

I’ll continue to call “fold_right” reduce for 3 reasons:
   – Analogy with Google’s Map/Reduce
   – The library’s arguments to fold_right are in the wrong order
   – Makes the example fit on a slide.
Summary

• Map and reduce are two *higher-order functions* that capture very, very common *recursion patterns*

• Reduce is especially powerful:
  – related to the “visitor pattern” of OO languages like Java.
  – can implement most list-processing functions using it, including things like copy, append, filter, reverse, map, etc.

• We can write clear, terse, reuseable code by exploiting:
  – higher-order functions
  – anonymous functions
  – first-class functions
  – polymorphism
Practice Problems

Using map, write a function that takes a list of pairs of integers, and produces a list of the sums of the pairs.

– e.g., list_add [(1,3); (4,2); (3,0)] = [4; 6; 3]
– Write list_add directly using reduce.

Using map, write a function that takes a list of pairs of integers, and produces their quotient if it exists.

– e.g., list_div [(1,3); (4,2); (3,0)] = [Some 0; Some 2; None]
– Write list_div directly using reduce.

Using reduce, write a function that takes a list of optional integers, and filters out all of the None’s.

– e.g., filter_none [Some 0; Some 2; None; Some 1] = [0;2;1]
– Why can’t we directly use filter? How would you generalize filter so that you can compute filter_none?

Using reduce, write a function to compute the sum of squares of a list of numbers.

– e.g., sum_squares = [3,5,2] = 38