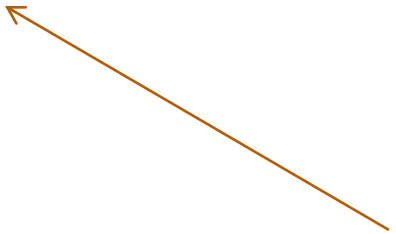
Poly-HO!



polymorphic,
higher-order
programming

HIGHER-ORDER FUNCTIONS

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 reusable 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 creating new functions?

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 reusable 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 creating new functions?
  - 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:

```
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)
```

Uses of the function:

```
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**: (λx. x+1) or (λx. x*x)

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
```

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

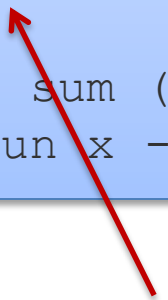
```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

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

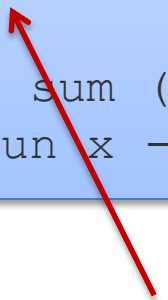
```
let rec reduce (f:int->int->int) (b:int) (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (reduce f b tl)

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

what does work is: (*)

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!

First-class functions: functions are treated like any other variable

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 *)
```

Curried Functions

Named after the logician Haskell B. Curry (1950s).

- 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, Moses Schönfinkel did some of this in 1924
  - thankfully, we don't have to talk about *Schönfinkelled* functions

Curry

Schönfinkel

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
```

SIMPLE REASONING ABOUT HIGHER-ORDER FUNCTIONS

Reasoning About Definitions

We can factor this program

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

into this program:

```
let square_all = map square
```

assuming we already have a definition of map

Reasoning About Definitions

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

```
let square_all = map square
```

Goal: Rewrite definitions so my program is simpler, easier to understand, more concise, …

Question: What are the reasoning principles for rewriting programs without breaking them? For reasoning about the behavior of programs? About the equivalence of two programs?

I want some *rules* that never fail.

Simple Equational Reasoning

Rewrite 1 (Function de-sugaring):

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

Rewrite 2 (Substitution):

if **arg** is a value or, when executed, **will always terminate without effect** and produce a value

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

roughly: all occurrences of x replaced by arg (though getting this *exactly* right is shockingly difficult)

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

Eliminating the Sugar in Map

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

Eliminating the Sugar in Map

```
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

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

let square_all =
    map square
```

Substitute map definition into square_all

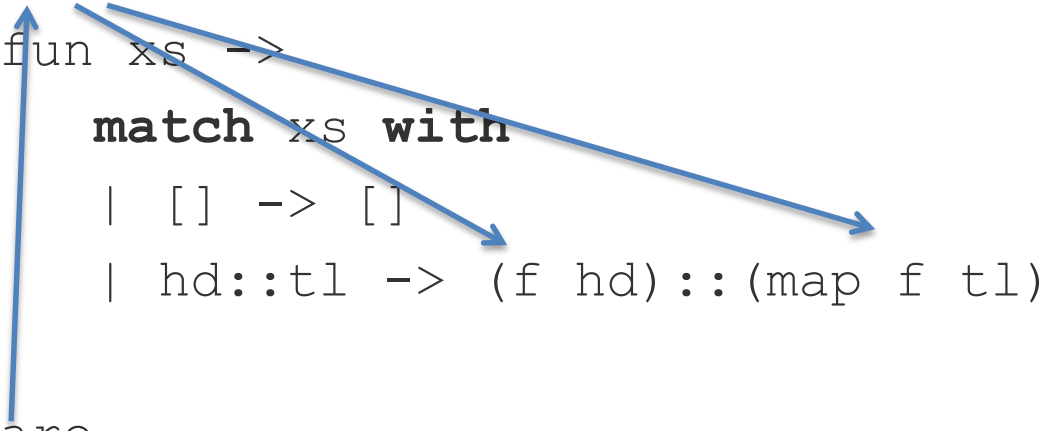
```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

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

Substitute map definition into square_all

```
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
```

Substitute map definition into square_all

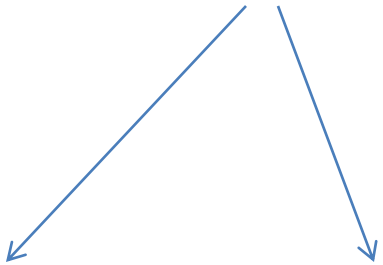
```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

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



Substitute 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)
        )
```

argument square substituted for parameter f



Expanding map square

```
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

Expanding map square

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

let square_all ys =
```

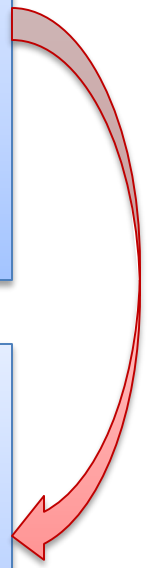
substitute again (argument ys for parameter xs)

```
            match ys with
            | [] -> []
            | hd::tl -> (square hd)::(map square tl)
```

So Far

```
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)
```



proof by simple rewriting unrolls definition once

Next Step

```
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 rec square_all ys =
  match ys with
    | [] -> []
    | hd::tl -> (square hd)::(square_all tl)
```

proof by simple rewriting unrolls definition once

proof *by induction* eliminates recursive function map

Summary

We saw this:

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

let square_all ys = map square ys
```

Is equivalent to this:

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

Morals 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

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.

(technical term: *principal type*)

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)
```

Optimization

What does this program do?

```
map f (map g [x1; x2; …; xn])
```

For each element of the list x1, x2, x3 ... xn, it executes g, creating:

```
map f ([g x1; g x2; …; g xn])
```

Then for each element of the list [g x1, g x2, g x3 ... g xn], it executes f, creating:

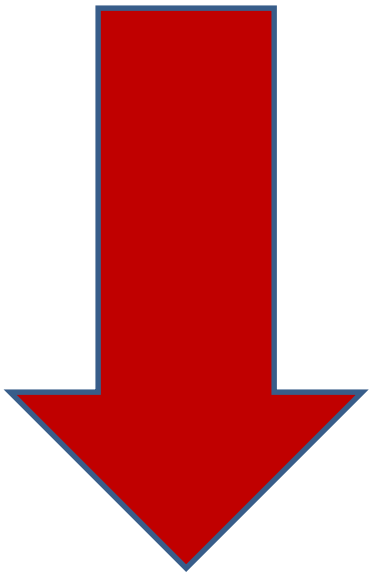
```
[f (g x1); f (g x2); …; f (g xn)]
```

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

```
map (comp f g) [x1; x2; ...; xn]
```

Deforestation

```
map f (map g [x1; x2; …; xn])
```



This kind of optimization has a name:

deforestation

(because it eliminates intermediate lists and, um, trees…)

```
map (comp f g) [x1; x2; ...; xn]
```

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

What is the type of comp?

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

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

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

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

How about reduce?

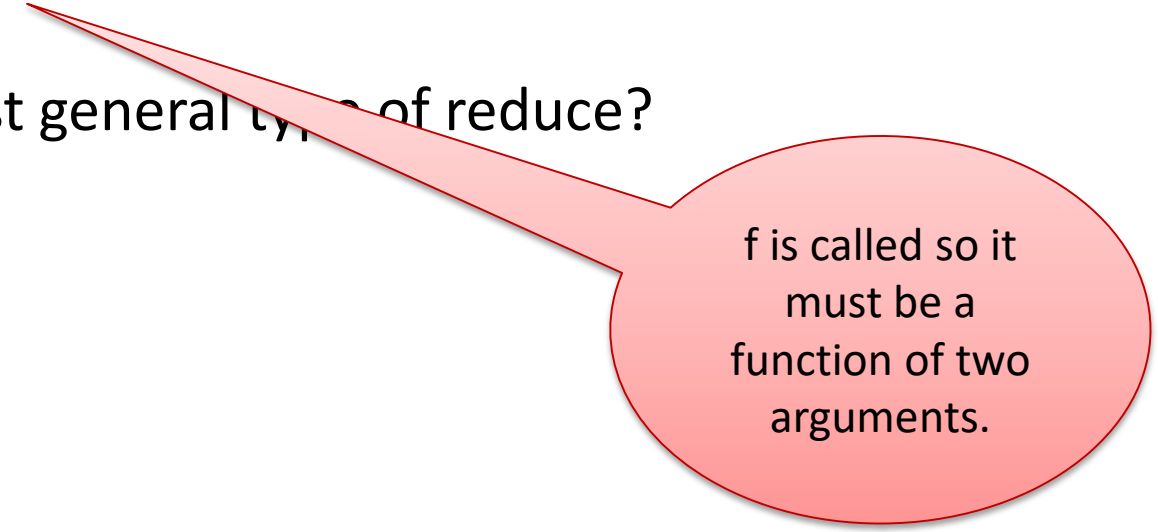
```
let rec reduce f u (xs: 'a list)  =
   match xs with
   | [] -> u
   | hd::tl -> f hd (reduce f u tl)
```

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?



How about reduce?

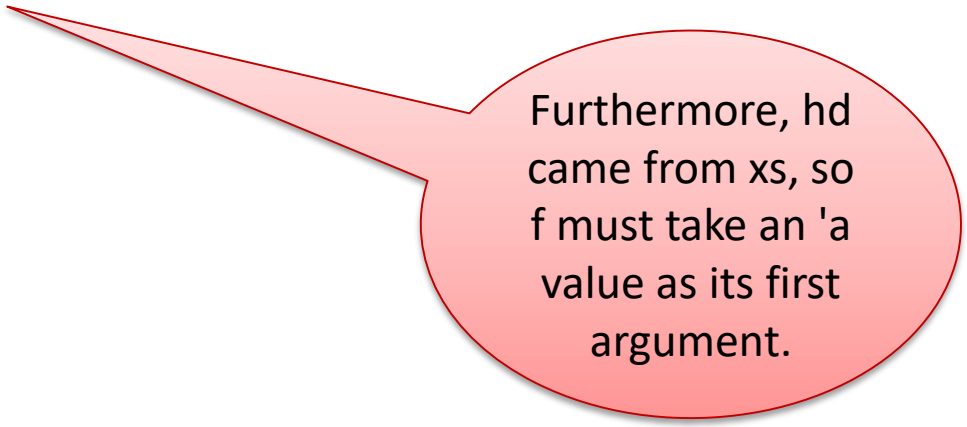
```
let rec reduce (f:? -> ? -> ?) u (xs: 'a list)  =
   match xs with
   | [] -> u
   | hd::tl -> f hd (reduce f u tl)
```

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?



How about reduce?

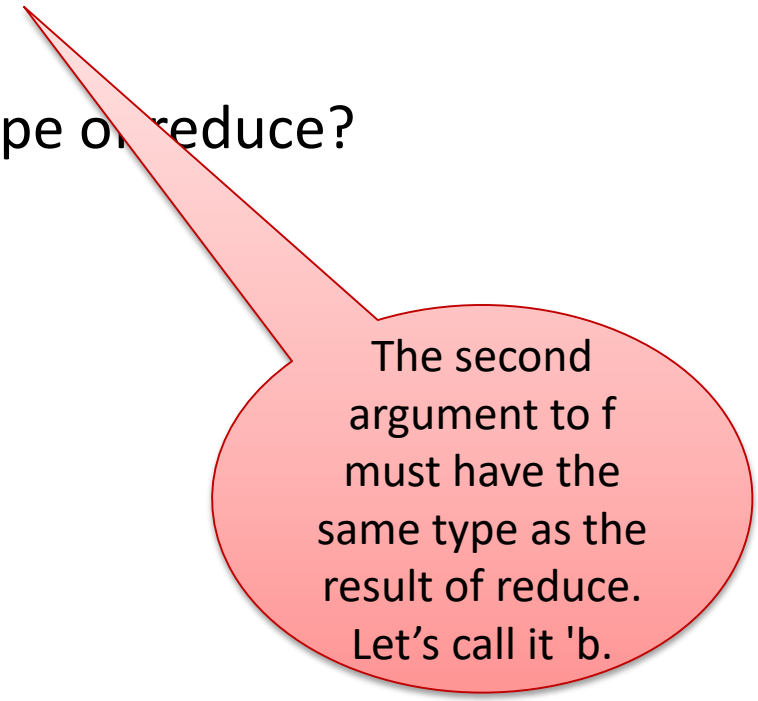
```
let rec reduce (f:'a -> ? -> ?) u (xs: 'a list)  =
   match xs with
   | [] -> u
   | hd::tl -> f hd (reduce f u tl)
```

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?



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 -> '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 -> '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
```

FUNCTIONAL DECOMPOSITION

Functional Decomposition

==

Break down complex problems in to a set of simple functions;
Recombine (compose) functions to form solution

So Far

We saw several list combinators.

A *combinator* is just a (higher-order) function that can be composed effectively with other functions

So Far

We saw several list combinators.

A *combinator* is just a (higher-order) function that can be composed effectively with other functions

List.map

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

map f [x1; x2; x3] == [f x1; f x2; f x3]
```

List.fold_right (approximately)

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

reduce g u [x1; x2; x3] == g x1 (g x2 (g x3 u))
```

What does this do?

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

let mystery0 = reduce (fun x y -> 1+y) 0
```

What does this do?

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

let mystery0 = reduce (fun x y -> 1+y) 0;;

let rec mystery0 xs =
  match xs with
   | [] -> 0
   | hd::tl ->
      (fun x y -> 1+y) hd (reduce (fun ...) 0 tl)
```

What does this do?

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

let mystery0 = reduce (fun x y -> 1+y) 0

let rec mystery0 xs =
   match xs with
   | [] -> 0
   | hd::tl -> 1 + reduce (fun ...) 0 tl
```

What does this do?

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

let mystery0 = reduce (fun x y -> 1+y) 0

let rec mystery0 xs =
   match xs with
   | [] -> 0
   | hd::tl -> 1 + mystery0 tl
```

What does this do?

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

let mystery0 = reduce (fun x y -> 1+y) 0

let rec mystery0 xs =
   match xs with
   | [] -> 0
   | hd::tl -> 1 + mystery0 tl   List Length!
```

What does this do?

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

let mystery1 = reduce (fun x y -> x::y) []
```

What does this do?

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

let mystery1 = reduce (fun x y -> x::y) []

let rec mystery1 xs =
   match xs with
   | [] -> []
   | hd::tl -> hd::(mystery1 tl)   Copy!
```

And this one?

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

let mystery2 g =
    reduce (fun a b -> (g a)::b) []
```

And this one?

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

let mystery2 g =
    reduce (fun a b -> (g a)::b) []

let rec mystery2 g xs =
   match xs with
   | [] -> []
   | hd::tl -> (g hd)::(mystery2 g tl) map!
```

Map and Reduce

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

```
val reduce : ('a -> 'b -> 'b) -> 'b ->  'a list -> 'b
```

We coded map in terms of reduce:

- ie: we showed we can compute map f xs using a call to reduce ? ? ? just by passing the right arguments in place of ? ? ?

Can we code reduce in terms of map? **NO**

Some Other Combinators: List Module

https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list

List.mapi f [a0; ...; an] == [f 0 a0; … ; f n an]
```

```
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list

List.map2 f [a0; ...; an] [b0; ...; bn] == [f a0 b0 ; … ; f an bn]
```

```
val iter : ('a -> unit) -> 'a list -> unit

List.iter f [a0; ...; an] == f a0; … ; f an
```

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, reusable 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? Alternatively, rig up a solution using filter + map.

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

- e.g., sum_squares = [3,5,2] = 38

PIPELINES

Pipe

```
let (|>) x f = f x ;;
```

Type?

Pipe

```
let (|>) x f = f x ;;
```

Type?

```
(|>) : 'a -> ('a -> 'b) -> 'b
```

Pipe

```
let (|>) x f = f x ;;
```

```
let twice f x =
    x |> f |> f;;
```

Pipe

```
let (|>) x f = f x ;;
```

```
let twice f x =
  (x |> f) |> f;;
```

left associative: x |> f1 |> f2 |> f3 == ((x |> f1) |> f2) |> f3

Pipe

```
let (|>) x f = f x ;;
```

```
let twice f x =
  x |> f |> f;;
```

```
let square x = x*x;;
```

```
let fourth x = twice square x;;
```

Pipe

```
let (|>) x f = f x ;;
```

```
let twice f x = x |> f |> f;;
let square x = x*x;;
let fourth x = twice square x;;

let compute x =
  x |> square
    |> fourth
    |> ( * ) 3
    |> print_int
    |> print_newline;;
```

PIPING LIST PROCESSORS

Another Problem

```
type student = {first:  string;
                last:   string;
                assign: float list;
                final:  float};;

let students : student list =
  [
   {first  = "Sarah";
    last   = "Jones";
    assign = [7.0;8.0;10.0;9.0];
    final  = 8.5};

   {first  = "Qian";
    last   = "Xi";
    assign = [7.3;8.1;3.1;9.0];
    final  = 6.5};
  ]
;;
```

Another Problem

```
type student = {first:   string;
                last:    string;
                assign:  float list;
                final:   float};;
```

- Create a function display that does the following:
  - for each student, print the following:
    - last_name, first_name: score
    - score is computed by averaging the assignments with the final
      - each assignment is weighted equally
      - the final counts for twice as much
    - one student printed per line
    - students printed in order of score

Another Problem

Create a function display that

- takes a list of students as an argument
- prints the following for each student:
  - last_name, first_name: score
  - score is computed by averaging the assignments with the final
    - each assignment is weighted equally
    - the final counts for twice as much
  - one student printed per line
  - students printed in order of score

```
let display (students : student list) : unit =
  students |> compute score
           |> sort by score
           |> convert to list of strings
           |> print each string
```

Another Problem

```
let compute_score
   {first=f; last=l; assign=grades; final=exam} =
   let sum x (num,tot) = (num+1,tot+.x) in
   let assign_total grades =
      List.fold_right sum grades (0,0.0) in
   let assign_score (num,tot) =
      tot /. float_of_int num in
   let assign_avg =
      assign_score (assign_total grades) in
   (f, l, (assign_avg +. exam *. 2.0) /. 3.0);;
```

```
let display (students : student list) : unit =
  students |> List.map compute_score
           |> sort by score
           |> convert to list of strings
           |> print each string
```

Another Problem

```
let student_compare (_,_,score1) (_,_,score2) =
  if score1 < score2 then 1
  else if score1 > score2 then -1
  else 0
;;
```

```
let display (students : student list) : unit =
  students |> List.map compute_score
           |> List.sort compare_score
           |> convert to list of strings
           |> print each string
```

Another Problem

```
let stringify (first, last, score) =
  last ^ ", " ^ first ^ ": " ^ string_of_float score;;
```

```
let display (students : student list) : unit =
  students |> List.map compute_score
           |> List.sort compare_score
           |> List.map stringify
           |> print each string
```

Another Problem

```
let stringify (first, last, score) =
  last ^ ", " ^ first ^ ": " ^ string_of_float score;;
```

```
let display (students : student list) : unit =
  students |> List.map compute_score
           |> List.sort compare_score
           |> List.map stringify
           |> List.iter print_endline
```

COMBINATORS FOR OTHER TYPES: PAIRS

Simple Pair Combinators

```
let both   f (x,y) = (f x, f y);;
let do_fst f (x,y) = (f x,   y);;
let do_snd f (x,y) = (  x, f y);;
```

pair combinators

Example: Piping Pairs

```
let both   f (x,y) = (f x, f y);;
let do_fst f (x,y) = (f x,   y);;
let do_snd f (x,y) = (  x, f y);;

let even x = (x/2)*2 == x;;

let process (p : float * float) =
  p |> both int_of_float       (* convert to float *)
    |> do_fst (fun x -> x/3)   (* divide fst by 3  *)
    |> do_snd (fun y -> y/2)   (* divide snd by 2  *)
    |> both even               (* test for even    *)
    |> fun (x,y) -> x && y     (* both even        *)
```

pair combinators

Summary

- (|>) passes data from one function to the next
  - compact, elegant, clear

- UNIX pipes (|) compose file processors
  - unix scripting with | is a kind of functional programming
  - but it isn't very general since | is not polymorphic
  - you have to serialize and unserialize your data at each step
    - there can be uncaught type mismatches between steps
    - we avoided that in Assignment 2, which is pretty simple …

- Higher-order *combinator libraries* arranged around types:
  - List combinators (map, fold, reduce, iter, …)
  - Pair combinators (both, do_fst, do_snd, …)

Reading Assignments

- Lecture Notes 06: Polymorphism and Higher-Order Programming