Functional Decomposition

COS 326
David Walker
Princeton University

Functional Decomposition

==

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

Last Time

We saw several list combinators.

A *library of combinators* is just a set of (higher-order) functions that operate over some data type T.

Last Time

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 list -> 'b list

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

can we code reduce in terms of map?

Some Other Combinators: List Module

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

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

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
```

```
val sort : ('a -> 'a -> int) -> 'a list -> 'a list

val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

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;;
```

Pipe

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

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

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 score gs exam = List.fold_right sum gs (0,0.0) in

  let (number, total) = score grades exam in
  (f, l, total /. float_of_int number)
;;
```

```
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);;
```

pair combinators

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

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

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 your assignment, which is pretty simple …
- Higher-order *combinator libraries* arranged around types:
  - List combinators (map, fold, reduce, iter, …)
  - Pair combinators (both, do_fst, do_snd, …)
  - Network programming combinators (Frenetic: frenetic-lang.org)

End