Functional Decomposition

COS 326
Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
Functional Decomposition

Break down complex problems into a set of simple functions; Recombine (compose) functions to form solution
We saw several list combinators.

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

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

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

\[ \text{map } f \ [x1; x2; x3] = [f \ x1; f \ x2; f \ x3] \]

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

\[ \text{reduce } g \ u \ [x1; x2; x3] = g \ x1 \ (g \ x2 \ (g \ x3 \ u)) \]
What does this do?

```ocaml
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?

```ocaml
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?

```plaintext
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
```
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?

```ocaml
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!
```
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 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!
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 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

We coded \texttt{map} in terms of \texttt{reduce}:

\begin{itemize}
  \item ie: we showed we can compute \texttt{map \ f \ xs} using a call to \texttt{reduce \ ? \ ? \ ?} just by passing the right arguments in place of \texttt{? \ ? \ ?}
\end{itemize}

Can we code \texttt{reduce} in terms of \texttt{map}?
Val \text{mapi} : (\text{int} \rightarrow \text{a} \rightarrow \text{unit}) \rightarrow \text{a list} \rightarrow \text{unit}

List.mapi f [a0; \ldots; an] == f 0 a0; \ldots; f n an

Val \text{map2} : (\text{a} \rightarrow \text{b} \rightarrow \text{c}) \rightarrow \text{a list} \rightarrow \text{b list} \rightarrow \text{c list}

List.map2 f [a0; \ldots; an] [b0; \ldots; bn] == [f a0 b0; \ldots; f an bn]

Val \text{iter} : (\text{a} \rightarrow \text{unit}) \rightarrow \text{a list} \rightarrow \text{unit}

List.iter f [a0; \ldots; an] == f a0; \ldots; f an

Val \text{sort} : (\text{a} \rightarrow \text{a} \rightarrow \text{int}) \rightarrow \text{a list} \rightarrow \text{a list}

Val \text{stable_sort} : (\text{a} \rightarrow \text{a} \rightarrow \text{int}) \rightarrow \text{a list} \rightarrow \text{a list}

\url{http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html}
PIPELINES
let (|>) x f = f x ;;

Type?
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
let (|>) x f = f x ;;

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

let square x = x*x;;

let fourth x = twice square;;
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

```haskell
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

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

```haskell
type student = {first: string;
    last: string;
    assign: float list;
    final: float};
```
Do Professors Dream of Homework-grade Databases?

(1968 novel)
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

```ocaml
let display (students : student list) : unit =
  students |> compute score
  |> sort by score
  |> convert to list of strings
  |> print each string
```
let compute_score
{first=f; last=l; assign=grades; final=exam} =

let sum x (num,tot) = (num +. 1., tot +. x) in

let score gs e = List.fold_right sum gs (2., 2. *. e) in

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

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

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

let stringify (first, last, score) =
last ^ ", " ^ first ^ ": " ^ string_of_float score;;
COMBINATORS FOR OTHER TYPES:
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
Example: Piping Pairs

```ocaml
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 int *) |
    |> do_fst ((/) 3) (* divide fst by 3 *) |
    |> do_snd ((/) 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 (ie: file format) 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)