Functional Decomposition

COS 326
David Walker
Princeton University
Functional Decomposition

==

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

Such problems can often be solved using a combinator library.
(a set of functions that fit together nicely)

The list library, which contains map and fold, is a combinator library.
PIPELINES
let (|>) x f = f x ;;
let (|>) x f = f x ;;

Type?

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

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

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

(Combining combinators cleverly)
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

```typescript
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
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

```plaintext
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 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
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 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);
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 (.currentTimeMillis/3) (* divide fst by 3 *)
    |> do_snd (currentTime/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 (\(\mid\)) 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)