Combinators and Pipelining

CSI 3120

Amy Felty

University of Ottawa

Technique: Functional Decomposition

Application: Scripting

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

Pipe

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

Pipe

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



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

Pipe

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



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



```
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

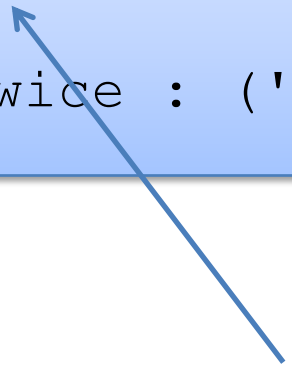
Pipe

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



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

val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

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

Pipe

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



```
let twice f x =
    x |> f |> f
 val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```



```
let square x = x*x
 val square : int -> int = <fun>
```

```
let fourth x = twice square x
 val fourth : int -> int = <fun>
```

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

val compute : int -> unit = <fun>
```

Pipe

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

# compute 8;;
50331648
- : unit = ()
# compute 3;;
19683
- : unit = ()
```

PIPING LIST PROCESSORS

(Combining combinators cleverly)

Another Problem

```
type first = string
type last = string
type assign = float list
type final = float
type student = (first * last * assign * final)

let students : student list =
  [("Sarah", "Jones", [7.0;8.0;10.0;9.0], 8.5);
   ("Qian", "Xi", [7.3;8.1;3.1;9.0], 6.5);
  ]
```

Another Problem

```
type first = string
type last = string
type assign = float list
type final = float
type student = (first * last * assign * final)
```


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

Another Problem

```
let compute_score (s:student) =
  match s with
    (f,l,grades,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 (ss : student list) : unit =
        ss |> List.map compute_score
           |> sort by score
           |> convert to list of strings
           |> print each string
```

Another Problem

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



```
let display (ss : student list) : unit =
        ss |> 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 (ss : student list) : unit =
        ss |> 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 (ss : student list) : unit =
        ss |> 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)

val both : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>
val do_fst : ('a -> 'b) -> 'a * 'c -> 'b * 'c = <fun>
val do_snd : ('a -> 'b) -> 'c * 'a -> 'c * 'b = <fun>
```

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 mod 2 = 0)

let process (p : float * float) =
  p |> both int_of_float          (* convert to int    *)
    |> do_fst ((/) 30)            (* divide 30 by fst  *)
    |> do_snd ((/) 20)            (* divide 20 by snd  *)
    |> 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 type (ie file format) mismatches between steps

- 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)