CSCI 2041

Sebastiaan Joosten

September 18th: Higher order types primer

University of Minnesota
Driven to Discover®

Overview

- map-like functions
- **filter-like functions**
- abstract map
- **abstract filter**
- coming up with helper functions

Map-like functions

- Write a function double_each that takes a list of integers, and doubles each integer
- Write a function add_each that takes a list of integer-pairs, and adds them
- Write a function selfpair_each that takes a list of integers, and turns each integer x into a pair (x,x).

double_each

- let double_each …

double_each

- let rec double_each lst = match lst with

double_each

- let rec double_each lst = match lst with
  | [] ->
  | (h::tl) ->

double_each

- let rec double_each lst = match lst with
  | [] -> []
  | (h::tl) ->

double_each

- let rec double_each lst = match lst with
  | [] -> []
  | (h::tl) -> (2*h) ::

double_each

- let rec double_each lst = match lst with
  | [] -> []
  | (h::tl) -> (2*h) :: double_each tl

double_each

- let rec double_each lst = match lst with
  | [] -> []
  | (h::tl) -> (2*h) :: double_each tl

```
val double_each : int list -> int list =
<fun>

utop # double_each [4;5;6];;
- : int list = [8; 10; 12]
```

add_each

- Write a function add_each that takes a list of integer-pairs, and adds them
- What's the type?

add_each

- val add_each : (int * int) list -> int list = <fun>

add_each

- val add_each : (int * int) list -> int list = <fun>

```
let rec add_each lst = match lst with
    | [] -> []
    | ((i1,i2)::tl) -> (i1+i2) :: add_each tl
```

selfpair_each

- Write a function selfpair_each that takes a list of integers, and turns each integer x into a pair (x,x).
- val selfpair_each : ‘a list -> (‘a,’a) list = <fun>
- Why does this type work?

selfpair_each

- Write a function selfpair_each that takes a list of integers, and turns each integer x into a pair (x,x).
- val selfpair_each : ‘a list -> (‘a,’a) list = <fun>
- Why does this type work?

selfpair_each

- Write a function selfpair_each that takes a list of integers, and turns each integer x into a pair (x,x).
- val selfpair_each : ‘a list -> (‘a,’a) list = <fun>
- Why does this type work?

```
let rec selfpair_each lst = match lst with
    | [] -> []
    | (h::tl) -> (h,h) :: selfpair_each tl
```

Filter

- Write a function filter_nonzero that takes a list of integers and returns all non-zero ones
- Write a function filter_nonnegative that takes a list of integers and returns all non-negative ones

Filter

- Write a function filter_nonzero that takes a list of integers and returns all non-zero ones

```
let rec filter_nonzero lst = match lst with
```

Filter

- Write a function filter_nonzero that takes a list of integers and returns all non-zero ones

```
let rec filter_nonzero lst = match lst with
  | [] ->
  | (h :: tl) ->
```

Filter

- Write a function filter_nonzero that takes a list of integers and returns all non-zero ones

```
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) ->
```

Filter

- Write a function filter_nonzero that takes a list of integers and returns all non-zero ones

```
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) -> if h = 0
                 then
                 else
```

Filter

- Write a function filter_nonzero that takes a list of integers and returns all non-zero ones

```
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) -> if h = 0
                 then filter_nonzero tl
                 else h :: filter_nonzero tl
```

Filter

- Write a function filter_nonnegative that takes a list of integers and returns all non-negative ones
- What do we change?

```
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) -> if h = 0
                 then filter_nonzero tl
                 else h :: filter_nonzero tl
```

Filter

- Write a function filter_nonnegative that takes a list of integers and returns all non-negative ones
- What do we change?

```
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) -> if h = 0
                 then filter_nonzero tl
                 else h :: filter_nonzero tl
```

Filter

- Write a function filter_nonnegative that takes a list of integers and returns all non-negative ones
- What do we change?

```
let rec filter_nonnegative lst = match lst with
  | [] -> []
  | (h :: tl) -> if h < 0
                 then filter_nonnegative tl
                 else h :: filter_nonnegative tl
```

Quiz time

Higher order functions

- These functions are all about taking the ‘what do we change’ out of a function

Map-like functions

```
let rec double_each lst = match lst with
    | [] -> []
    | (h::tl) -> (2*h) :: double_each tl
let rec add_each lst = match lst with
    | [] -> []
    | ((i1,i2)::tl) -> (i1+i2) :: add_each tl
let rec selfpair_each lst = match lst with
    | [] -> []
    | (h::tl) -> (h,h) :: selfpair_each tl
```

- What's the general part?

```
let rec map f lst = match lst with
```

Map-like functions

```
let rec double_each lst = match lst with
    | [] -> []
    | (h::tl) -> (2*h) :: double_each tl
let rec add_each lst = match lst with
    | [] -> []
    | ((i1,i2)::tl) -> (i1+i2) :: add_each tl
let rec selfpair_each lst = match lst with
    | [] -> []
    | (h::tl) -> (h,h) :: selfpair_each tl
```

- What's the general part?

```
let rec map f lst = match lst with [] -> []
    |
```

Map-like functions

```
let rec double_each lst = match lst with
    | [] -> []
    | (h::tl) -> (2*h) :: double_each tl
let rec add_each lst = match lst with
    | [] -> []
    | ((i1,i2)::tl) -> (i1+i2) :: add_each tl
let rec selfpair_each lst = match lst with
    | [] -> []
    | (h::tl) -> (h,h) :: selfpair_each tl
```

- What's the general part?

```
let rec map f lst = match lst with [] -> []
    | (h::tl) -> … :: …
```

Map-like functions

```
let rec double_each lst = match lst with
    | [] -> []
    | (h::tl) -> (2*h) :: double_each tl
let rec add_each lst = match lst with
    | [] -> []
    | ((i1,i2)::tl) -> (i1+i2) :: add_each tl
let rec selfpair_each lst = match lst with
    | [] -> []
    | (h::tl) -> (h,h) :: selfpair_each tl
```

- What's the general part?

```
let rec map f lst = match lst with [] -> []
    | (h::tl) -> … :: map f tl
```

Map-like functions

```
let rec double_each lst = match lst with
    | [] -> []
    | (h::tl) -> (2*h) :: double_each tl
let rec add_each lst = match lst with
    | [] -> []
    | ((i1,i2)::tl) -> (i1+i2) :: add_each tl
let rec selfpair_each lst = match lst with
    | [] -> []
    | (h::tl) -> (h,h) :: selfpair_each tl
```

- What's the general part?

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

Using that map…

```
let rec double_each lst = (let f h = 2*h
                           in map f lst)

let rec add_each lst = (let f (i1,i2) = i1 + i2
                        in map f lst)

let rec selfpair_each lst = (let f h = (h,h)
                             in map f lst)
```

- Using this map function (put above)

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

Cleaning up…

- Instead of needing a ‘let’,
  we can sometimes use ‘fun’ to define a function without giving it a name:

```
let rec double_each lst = (let f h = 2*h
                           in map f lst)
let rec double_each lst = map (fun h -> 2*h) lst
```

Cleaning up…

- Instead of needing a ‘let’,
  we can sometimes use ‘fun’ to define a function without giving it a name:

```
let rec double_each lst = map (fun h -> 2*h) lst

let rec add_each lst = map (fun (i1,i2) -> i1+i2) lst

let rec selfpair_each lst = map (fun h -> (h,h)) lst
```

- (Using our map function)

Cleaning up (2)…

- Instead of defining our own map function, we can use ocaml's built-in one: List.map

```
let rec double_each lst =
  List.map (fun h -> 2*h) lst

let rec add_each lst =
  List.map (fun (i1,i2) -> i1+i2) lst

let rec selfpair_each lst =
  List.map (fun h -> (h,h)) lst
```

Filter function

```
let rec filter_nonnegative lst = match lst with
  | [] -> []
  | (h :: tl) -> if h < 0
                 then filter_nonnegative tl
                 else h :: filter_nonnegative tl

let rec filter p lst = match lst with
  | [] -> []
  | (h :: tl) -> if p h then h :: filter p tl
                 else filter p tl
```

Filter function

```
let rec filter_nonnegative lst = match lst with
  | [] -> []
  | (h :: tl) -> if h < 0
                 then filter_nonnegative tl
                 else h :: filter_nonnegative tl
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) -> if h = 0
                 then filter_nonzero tl
                 else h :: filter_nonzero tl
let rec filter p lst = match lst with
  | [] -> []
  | (h :: tl) -> if p h then h :: filter p tl
                 else filter p tl
```

Filter function

```
let rec filter_nonnegative lst =
  filter (fun h -> … ) lst
  (* match lst with
  | [] -> []
  | (h :: tl) -> if h < 0
                 then filter_nonnegative tl
                 else h :: filter_nonnegative tl *)
let rec filter_nonzero lst = match lst with
  | [] -> []
  | (h :: tl) -> if h = 0
                 then filter_nonzero tl
                 else h :: filter_nonzero tl
let rec filter p lst = match lst with
  | [] -> []
  | (h :: tl) -> if p h then h :: filter p tl
                 else filter p tl
```

Filter function

```
let rec filter_nonnegative lst =
  filter (fun h -> h >= 0 ) lst

let rec filter_nonzero lst =
  filter (fun h -> h <> 0) lst

let rec filter p lst = match lst with
  | [] -> []
  | (h :: tl) -> if p h then h :: filter p tl
                 else filter p tl
```

Filter function

```
let rec filter_nonnegative lst =
  List.filter (fun h -> h >= 0 ) lst

let rec filter_nonzero lst =
  List.filter (fun h -> h <> 0) lst
```

Coming up with helper functions

- Worst thing to do is try and generalize a function *before* writing it!

Coming up with helper functions

- Know about a helper function? (like List.map or List.filter), you *may* use it
- Are you seeing a function that can be generalized?
  - **Don't change it** until there's a reason to do so!
- Are you seeing two functions that can be generalized?
  - Now there's a reason to do so!
  - **Reconsider:** is this worth it?
  - First make the functions appear as equal as possible.
  - Turn any differences into variables.
  - (Those variable differences are often functions)