Programming Languages and Techniques (CIS120)

Lecture 9

Higher-order functions: transform and fold

Lecture notes: Chapter 9

List transformations

A fundamental design pattern
using first-class functions

Phone book example

```
type entry = string * int
let phone_book = [ ("Pat", 2155559092); … ]

let rec get_names (p : entry list) : string list =
  begin match p with
  | ((name, num)::rest) -> name :: get_names rest
  | [] -> []
  end

let rec get_numbers (p : entry list) : int list =
  begin match p with
  | ((name, num)::rest) -> num :: get_numbers rest
  | [] -> []
  end
```

Can we use first-class functions to refactor code to share common structure?

Phone book example

```
type entry = string * int
let phone_book = [ ("Pat", 2155559092); … ]

let rec get_names (p : entry list) : string list =
  begin match p with
  | (e::rest) -> fst e :: get_names rest
  | [] -> []
  end

let rec get_numbers (p : entry list) : int list =
  begin match p with
  | (e::rest) -> snd e :: get_numbers rest
  | [] -> []
  end
```

fst and snd are functions that access the parts of a tuple:

```
let fst (x,y) = x
let snd (x,y) = y
```

Refactoring

```
let rec helper (f: entry->'b) (p: entry list) : 'b list =
  begin match p with
  | (e::rest) -> f e :: helper f rest
  | [] -> []
  end

let get_names (p: entry list) : string list =
  helper fst p
let get_numbers (p: entry list) : int list =
  helper snd p
```

fst and snd are functions that access the parts of a tuple:

```
let fst (x,y) = x
let snd (x,y) = y
```

The argument `f` determines what happens with the entry at the head of the list

Going even more generic

```
let rec helper (f: entry->'b) (p: entry list) : 'b list =
  begin match p with
  | (e::rest) -> f e :: helper f rest
  | [] -> []
  end

let get_names (p: entry list) : string list =
  helper fst p
let get_numbers (p: entry list) : int list =
  helper snd p
```

Now let's make it work for *all* lists, not just lists of entries…

Going even more generic

```
let rec helper (f: 'a->'b) (p: 'a list) : 'b list =
  begin match p with
  | (e::rest) -> f e :: helper f rest
  | [] -> []
  end

let get_names (p: entry list) : string list =
  helper fst p
let get_numbers (p: entry list) : int list =
  helper snd p
```

‘a stands for (string*int)
‘b stands for int

snd : (string*int) -> int

Transforming Lists

```
let rec transform (f: 'a->'b) (l:'a list) : 'b list =
  begin match l with
  | []    -> []
  | h::t -> (f h)::(transform f t)
  end
```

List transformation

(a.k.a. "*mapping* a function across a list")

- foundational function for programming with lists
- used over and over again
- part of OCaml standard library (called List.map)

*many languages (including OCaml) use the terminology "map" for the function that transforms a list by applying a function to each element. Don't confuse List.map with "finite map".

What is the value of this expresssion?

```
transform (fun (x:int) -> x > 0)
    [0 ; -1; 1; -2]
```

1. [0; -1; 1; -2]
2. [1]
3. [1; 1; 0; 1]
4. [false; false; true; false]
5. runtime error

ANSWER: 4

The ‘fold’ design pattern

Refactoring code, again

- Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
  begin match l with
  | [] -> false
  | h :: t -> h || exists t
  end
```

```
let rec acid_length (l : acid list) : int =
  begin match l with
  | [] -> 0
  | h :: t -> 1 + acid_length t
  end
```

base case:
Simple answer when the list is empty

combine step:
Do something with the head of the list and the result of the recursive call

- Can we factor out this pattern using first-class functions?

Preparation

```
let rec exists (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || exists t
   end
```

```
let rec acid_length (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + acid_length t
   end
```

Preparation

```
let rec helper (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || helper t
   end

let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + helper t
   end

 let acid_length (l : acid list) = helper l
```

Abstracting with respect to Base

```
let rec helper (l : bool list) : bool =
   begin match l with
   | [] -> false
   | h :: t -> h || helper t
   end

let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> 1 + helper t
   end

 let acid_length (l : acid list) = helper l
```

Abstracting with respect to Base

```
let rec helper (base : bool) (l : bool list) : bool =
    begin match l with
    | [] -> base
    | h :: t -> h || helper base t
    end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
    begin match l with
    | [] -> base
    | h :: t -> 1 + helper base t
    end

 let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

 let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> h || helper base t
   end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> 1 + helper base t
   end

 let acid_length (l : acid list) = helper 0 l
```

Abstracting with respect to Combine

```
let rec helper (combine : bool -> bool -> bool)
               (base : bool) (l : bool list) : bool =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : acid -> int -> int)
               (base : int) (l : acid list) : int =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

 let acid_length (l : acid list) =
    helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

Making the Helper Generic

```
let rec helper (combine : 'a -> 'b -> 'b)
               (base : 'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : 'a -> 'b -> 'b)
               (base : 'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | h :: t -> combine h (helper combine base t)
   end

 let acid_length (l : acid list) =
    helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
             (base:'b) (l : 'a list) : 'b =
   begin match l with
   | [] -> base
   | x :: t -> combine x (fold combine base t)
   end

let exists (l : bool list) : bool =
    fold (fun (h:bool) (acc:bool) -> h || acc) false l

let acid_length (l : acid list) : int =
    fold (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

- `fold` (a.k.a. "reduce")
  - Like transform, foundational function for programming with lists
  - Captures the pattern of recursion over lists
  - Also part of OCaml standard library (`List.fold_right`)
  - Similar operations for other recursive datatypes (`fold_tree`)

How would you rewrite this function

```
let rec sum (l : int list) : int =
   begin match l with
   | [] -> 0
   | h :: t -> h + sum t
   end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int) -> acc + 1)`
   base is: `0`

2. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `0`

3. combine is: `(fun (h:int) (acc:int) -> h + acc)`
   base is: `1`

4. sum can't be written with `fold`.

Answer: 2

How would you rewrite this function

```
let rec reverse (l : int list) : int list =
   begin match l with
   | [] -> []
   | h :: t -> reverse t @ [h]
   end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int list) -> h :: acc)`
   base is: `0`

2. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `0`

3. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
   base is: `[]`

4. reverse can't be written by with `fold`.

Answer: 3

Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Everyday programming practice offers many more examples
  - objects bundle "functions" (a.k.a. methods) with data
  - iterators ("cursors" for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- Also heavily used at "large scale": Google's MapReduce
  - Framework for transforming (mapping) sets of key-value pairs
  - Then "reducing" the results per key of the map
  - Easily distributed to 10,000 machines to execute in parallel!