CS 3110

Map and Fold

Prof. Clarkson
Fall 2015

Today's music: Selections from the soundtrack to *2001: A Space Odyssey*

Question

How much progress have you made on A1?

A. I'm still figuring out how Enigma works.
B. My code can cipher single letters.
C. My code can cipher multiple letters, but stepping is still iffy.
D. I'm done with **`cipher`** and **`simulate`**.
E. I've finished the scavenger hunt, too.

Review

Previously in 3110:

- Lists: OCaml's awesome built-in datatype
- Pattern matching: an awesome feature not found in most imperative languages

Today:

- No new language features
- New **idioms** and **library functions**:
  - Map, fold, and other higher-order functions

Review: higher-order functions

- Functions are values
- Can use them **anywhere** we use values
  - Arguments, results, bound to variables...
- Functions can **take** functions as arguments
- Functions can **return** functions as results

Review: anonymous functions

(aka *function expressions*)

- **Syntax: `fun x -> e`**
- **Type checking:**
  - Conclude that **`fun x -> e : t1 -> t2`** if ***`e`*`:`*`t2`*** under assumption ***`x`*`:`*`t1`***
- **Evaluation:**
  - A function is already a value

Lambda

- Anonymous functions a.k.a. *lambda expressions:* `λx . e`
- The lambda means "what follows is an anonymous function"
  - x is its argument
  - e is its body
  - Just like `fun x -> e`, but slightly different syntax
- Standard feature of any functional language (ML, Haskell, Scheme, …)

- You'll see "lambda" show up in many places in PL, e.g.:
  - PHP: http://www.php.net/manual/en/function.create-function.php
  - Java 8: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
  - A popular PL blog: http://lambda-the-ultimate.org/
  - Lambda style: https://www.youtube.com/watch?v=Ci48kqp11F8

HUGE HIGHER-ORDER FUNCTIONS

Map

bad style!

`map (fun x -> shirt_color(x)) [`

  

`]`

`= [gold; blue; red]`

Map

```
map shirt_color [
```



```
]

= [gold; blue; red]
```

Map

How to implement?

Let's see some special cases...

- Write a function that adds 1 to every element of a list
- Write a function that concatenates "3110" to every element of a list

Map

```
let rec add1 = function
  | [] -> []
  | h::t -> (h+1)::(add1 t)

let rec concat3110 = function
  | [] -> []
  | h::t -> (h^"3110")::(concat3110 t)
```

...notice the common structure

Map

```
let rec add1 = function
   | [] -> []
   | h::t -> (h+1)::(add1 t)

let rec concat3110 = function
   | [] -> []
   | h::t -> (h^"3110")::(concat3110 t)
```

notice the common structure

...same except for the blue part, which says what to do with head

...which is what the function passed to **`map`** does

Map

```
let rec map f = function
| [] -> []
| x::xs -> (f x)::(map f xs)
```

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

Map is HUGE:

- You use it **all the time** once you know it
- Exists in standard library as **`List.map`**, but the idea can be used in any data structure (trees, stacks, queues…)

Map

```
let add1 =
  List.map (fun x -> x+1)

let concat3110 =
  List.map (fun s -> s^"3110")
```

Note the separation of concerns:

- `List.map` knows how to traverse the list
- The function passed in knows how to transform each element

Question

What is value of `lst` after this code?

```
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. `[1;2;3;4]`
B. `[2;4]`
C. `[false; true; false; true]`
D. `false`

Question

What is value of `lst` after this code?

```
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. `[1;2;3;4]`
B. `[2;4]`
C. `[false; true; false; true]`
D. `false`

Filter

`filter is_vulcan [`    `]`

`= [`  `]`

(er, half vulcan)

Filter

```
let rec filter f = function
| [] -> []
| x::xs -> if f x
           then x::(filter f xs)
           else filter f xs
```

`filter : ('a -> bool) -> 'a list -> 'a list`

In library: **`List.filter`**

(library implementation is tail recursive; the one above is not)

Question

What is value of `lst` after this code?

```
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. `[1;2;3;4]`
B. `[2;4]`
C. `[false; true; false; true]`
D. `false`

Question

What is value of `lst` after this code?

```
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. `[1;2;3;4]`
B. `[2;4]`
C. `[false; true; false; true]`
D. `false`

Iterators

- `map` and `filter` are *iterators*
  - Not built-in to the language, an idiom

- Benefit of iterators: separate recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - Can reuse same data processing for different data structures
  - leads to modular, maintainable, beautiful code!

- So far: iterators that change or omit data
  - what about combining data?

Combining elements

- Write a function that sums all the elements of a list
- Write a function that concatenates all the elements of a list

Combining elements

```
let rec sum = function
  | [] -> 0
  | h::t -> h + (sum t)

let rec concat = function
  | [] -> ""
  | h::t -> h ^ (concat t)
```

notice the common structure

Combining elements

```
let rec sum = function
  | [] -> 0
  | h::t -> h + (sum t)

let rec concat = function
  | [] -> ""
  | h::t -> h ^ (concat t)
```

notice the common structure

...same except for the blue part, which gives

- a value to return for empty list
- a function to combine head with result of recursive call on tail

Combining elements

```
let rec combine init op = function
   | [] -> init
   | h::t -> op h (combine init op t)

let sum = combine 0 (+)
let concat = combine "" (^)
```

combining elements, using `init` and `op`, is the essential idea behind library functions known as `fold`

Question

What should the result of combining `[1;2;3;4]` with `1` and `( * )` be?

A. 1
B. 24
C. 10
D. 0

Question

What should the result of combining **`[1;2;3;4]`** with **`1`** and **`( * )`** be?

A. 1

B. 24

C. 10

D. 0

`List.fold_right`

`List.fold_right f [a;b;c] init`

computes

`f a (f b (f c init))`

Accumulates an answer by

- repeatedly applying **`f`** to an element of list and "answer so far"
- folding in list elements "from the right"

List.fold_right

```
let rec fold_right f xs acc =
  match xs with
    | []      -> acc
    | x::xs' -> f x (fold_right f xs' acc)
```

Note: `fold_right` is the same as `combine`
(just with argument order and names changed)

`List.fold_left`

`List.fold_left f init [a;b;c]`

computes

`f (f (f init a) b) c`

Accumulates an answer by

- repeatedly applying **`f`** to "answer so far" and an element of list
- folding in list elements “from the left”

List.fold_left

```
let rec fold_left f acc xs =
  match xs with
    | []      -> acc
    | x::xs' -> fold_left f (f acc x) xs'
```

Note: `fold_left` is a different computation than `fold_right` or `combine`

...what are the differences?

Difference 1: Left vs. right

folding `[1;2;3]` with **0** and `(+)`

left to right: ((0+1)+2)+3

right to left: 1+(2+(3+0))

Both evaluate to 6; does it matter?

Yes: not all operators are associative, e.g. subtraction, division, exponentiation, …

Difference 2: Tail recursion

Which of these is tail recursive?

```
let rec fold_left f acc xs =
  match xs with
    | []     -> acc
    | x::xs' -> fold_left f (f acc x) xs'

let rec fold_right f xs acc =
  match xs with
    | []     -> acc
    | x::xs' -> f x (fold_right f xs' acc)
```

A. neither
B. fold_left
C. fold_right
D. both fold_left and fold_right
E. I don't know

Difference 2: Tail recursion

Which of these is tail recursive?

```
let rec fold_left f acc xs =
  match xs with
    | []      -> acc
    | x::xs' -> fold_left f (f acc x) xs'

let rec fold_right f xs acc =
  match xs with
    | []      -> acc
    | x::xs' -> f x (fold_right f xs' acc)
```

A. neither

B. fold_left

C. fold_right

D. both fold_left and fold_right

E. I don't know

Difference 3: Types

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Difference 3: Types

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Final value of accumulator

Difference 3: Types

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Initial value of accumulator

Difference 3: Types

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Input list

Difference 3: Types

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Operator

Difference 3: Types

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Can't keep the argument order straight? Me neither.
There is actually a rational design (accumulator is always to left/right of list (element)).
The `ListLabels` module helps.

Behold the HUGE power of fold

Implement so many other functions with fold!

```
let rev xs = fold_left (fun xs x -> x::xs) [] xs

let length xs = fold_left (fun a _ -> a+1) 0 xs

let map f xs = fold_right
  (fun x a -> (f x)::a) xs []

let filter f xs = fold_right
  (fun x a -> if f x then x::a else a) xs []
```

MapReduce

- Fold has many synonyms/cousins in various functional languages, including **`scan`** and **`reduce`**
- Google organizes large-scale data-parallel computations with MapReduce
  - open source implementation by Apache called Hadoop

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately."
[Dean and Ghemawat, 2008]

Enrollment

- The fire-code capacity of Olin 155 is the limit
- At last count, there were 20-30 more people who wanted to take 3110 than there were seats available
- I cannot add students to the course myself
- The CS department does not do waitlists or prioritization for enrollment in 3110
- But the department has agreed to open a limited number of seats above the fire-code capacity
- Keep checking Student Center today
- 3110 will be offered in the spring, and historically is always undersubscribed then

Upcoming events

- [today] Add deadline
- [Thursday] A1 soft deadline
- [Saturday] A1 hard deadline

This is huge.

THIS IS 3110