Functional Programming with Ocaml
Week 2.

Brendon Faleiro

Revisiting some topics:

- Match … with
- Map
- Filter
- Fold

Match … with:

- We can use match statements to break down the list.
- Eg. Finding the sum of the elements in a list.

```
# let rec sum l =
    match l with
    | [] -> 0
    | hd :: tl -> hd + sum tl
  ;;
 val sum : int list -> int = <fun>
```

- # sum [1;2;3;4;5];;

- : int = 15

- # sum [];;

- : int = 0

```
# let rec drop_value l to_drop =
    match l with
    | [] -> []
    | to_drop :: tl -> drop_value tl to_drop
    | hd :: tl -> hd :: drop_value tl to_drop
  ;;
```

Characters 114-122:
Warning 11: this match case is unused.
val drop_value : 'a list -> 'a -> 'a list = <fun>

```
# drop_value [1;2;3] 2;;
- : int list = []
```

Match uses patterns to instantiate variables. The patterns do not refer to preexisting variables.

The correct method:

```
# let rec drop_value l to_drop =
    match l with
    | [] -> []
    | hd :: tl ->
      let new_tl = drop_value tl to_drop in
      if hd = to_drop then new_tl else hd :: new_tl
  ;;
```

```
val drop_value : 'a list -> 'a -> 'a list = <fun>
```

```
# drop_value [1;2;3] 2;;
 - : int list = [1; 3]
```

- Now write a similar method to drop a particular value.
- Eg. Drop 0 from a list of integers:

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl
    | hd :: tl -> hd :: drop_zero tl
  ;;
```

List.map

- ‘List.map’ takes a list and a function for transforming elements of that list, and returns a new list with the transformed elements.

- Eg. List.map String.length [“Hi”; “Hello”];;

 -:int list = [2;5]

- List.map (function x -> x+1) [1;2;3;4];;

 -:int list = [2;3;4;5]

- let incBy1 = List.map (function x -> x+1);;

  val incBy1: int list -> int list = <fun>

- Important note: map returns a list with the same number of elements as the input list.

- Assume the function int_of_char x returns the ascii value of the character x. Write a function that returns the ascii values of a list of characters.

```
# let ints_of_chars = List.map (fun x -> (int_of_char x));;
```

- Write a function that multiplies every element of a list by a given number ‘n’.

```
# let multiply n list = List.map (function x -> x * n) list;;
```

Or

```
# let multiply n list =
        let f x = n* x in
        List.map f list;;
```

List.filter

- The first argument is a function that takes in one argument and returns either true or false.
- The second argument is a list of type 'a.
- The return value is a (usually trimmed) list of type 'a.

- List.filter (function x -> true) [1;2;4;5];;

-:int list = [1;2;4;5]

- List.filter (function x -> false) [1;2;3;4;5];;

-:int list = []

- List.filter (function x -> x mod 2) [1;2;3;4;5];;

Error: This expression has type int but an expression was expected of type bool

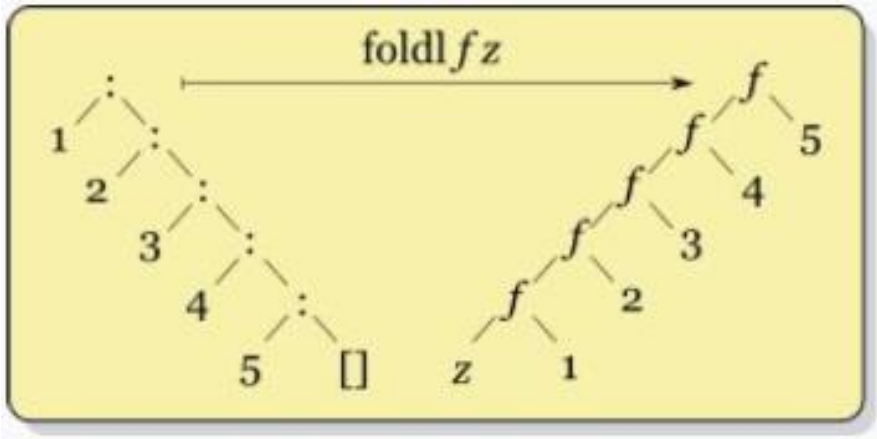
- List.filter (function x -> (x mod 2)=0) [1;2;3;4;5];;

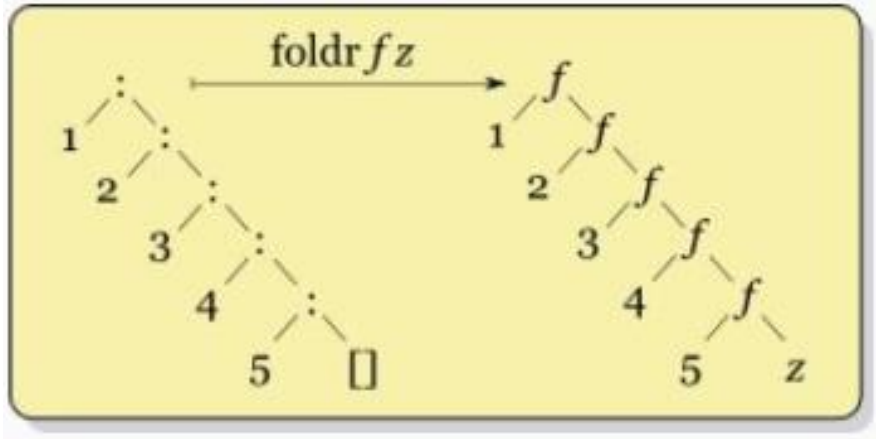
-:int list = [2;4]

fold_left / List.fold_right

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```





- To understand folding, we will first define our own list.

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

- Now we write fold_left:

```
let rec fold_left f acc lst =
  match lst with
  | Nil -> acc
  | Cons (h,t) -> fold_left f (f acc h) t
```

- And now we can similarly write fold_right:

```
let rec fold_right f lst acc =
  match lst with
  | Nil -> acc
  | Cons (h,t) -> f h (fold_right f t acc)
```

```
# let rec make_list n = if n > 0 then n :: make_list (n-1) else [];;

# let rec make_list n list = if n > 0 then make_list (n-1) (n::list) else list;;
```

- What is the difference between the above functions?
- Which is tail-recursive?
- Why is tail recursive better?

- Since fold_left is tail recursive there is no stack overflow.

- fold_right is not tail recursive, and so there is a possibility of a stack overflow.

- Given function f, base case result a, list b:-

  - Fold_left has the form :

    f (… (f (f a b1) b2) …) bn

  - Fold_right has the form:

    f b1 (f b2 (f b3 (f b4 (… (f bn a)))))

- Fold to get the sum of the elements of a list:

```
let sum (lst : int list) =
  let fold_function (acc : int) (elem : int) : int =
    acc + elem
  in
  List.fold_left fold_function 0 lst
```

Iteration	acc	elem
0	0	1
1	1	2
2	3	3
3	6	4
4	10	–

- Implement the map function using fold:
- Firstly, should we use fold left or fold right???

```
let map (f : 'a -> 'b) (lst : 'a list) : 'b list =
  let fold_function (elem : 'a) (acc : 'b list) =
    (f elem)::acc
  in
  List.fold_right fold_function lst []
```

- In certain cases the direction of folding matters, and we should make use of the correct folding mechanism.