Higher Order Programming

CS3100 Fall 2019

Review

Last time

- Pattern Matching

Today

- New Idioms and library functions.
  - Map, Reduce and Other higher order functions.

Double and Square

In [24]:

```
let double x = 2 * x
let square x = x * x
```

Out[24]:

```
val double : int -> int = <fun>
```

Out[24]:

```
val square : int -> int = <fun>
```

In [25]:

```
double 10
```

Out[25]:

```
- : int = 20
```

In [26]:

```
square 2
```

Out[26]:

```
- : int = 4
```

Quad and Fourth

In [27]:

```
let quad x = 2 * 2 * x
let fourth x = (x * x) * (x * x)
```

Out[27]:

```
val quad : int -> int = <fun>
```

Out[27]:

```
val fourth : int -> int = <fun>
```

In [28]:

```
quad 10
```

Out[28]:

```
- : int = 40
```

In [29]:

```
fourth 2
```

Out[29]:

```
- : int = 16
```

Quad and Fourth

Abstract away the details using `double` and `square` .

In [30]:

```
let quad x = double (double x)
```

Out[30]:

```
val quad : int -> int = <fun>
```

In [31]:

```
quad 10
```

Out[31]:

```
- : int = 40
```

In [32]:

```
let fourth x = square (square x)
```

Out[32]:

```
val fourth : int -> int = <fun>
```

In [33]:

```
fourth 2
```

Out[33]:

```
- : int = 16
```

Quad and Fourth

Abstract the act of applying twice.

In [34]:

```
let twice f x = f (f x)
```

Out[34]:

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

In [35]:

```
let quad x = twice double x
```

Out[35]:

```
val quad : int -> int = <fun>
```

In [36]:

```
let quad = twice double
```

Out[36]:

```
val quad : int -> int = <fun>
```

In [37]:

```
quad 10
```

Out[37]:

```
- : int = 40
```

Quad and Fourth

Abstract the act of applying twice.

In [38]:

```
let fourth = twice square
```

Out[38]:

```
val fourth : int -> int = <fun>
```

In [39]:

```
fourth 2
```

Out[39]:

```
- : int = 16
```

Applying a function for an arbitrary number of times

Instead of twice, what if I wanted to apply `n` time over an argument where `n` is supplied as an argument

In [40]:

```
let rec apply n f x =
  if n = 1 then f x
  else f (apply (n-1) f x)
```

Out[40]:

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

In [41]:

```
let quad = apply 6 double
```

Out[41]:

```
val quad : int -> int = <fun>
```

In [42]:

```
quad 10
```

Out[42]:

```
- : int = 640
```

Higher Order Programming over Lists

Map

&

Fold

(sibling of reduce)

MapReduce

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other ***functional languages.*** "

[Dean and Ghemawat, 2008]

Map

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

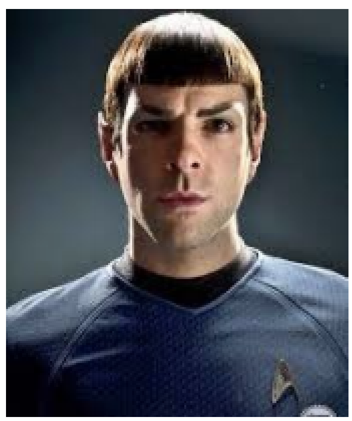
  

```
]
```

[Gold ; Blue ; Red]

Map

```
map shirt_color [
```

```
]
```

[Gold ; Blue ; Red]

Map

`List.map` takes a list `[a1; a2; ...; an]` and a higher-order function `f` and returns `[f a1; f a2; ...; f an]`.

In [43]:

```
List.map
```

Out[43]:

```
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

In [44]:

```
List.map (fun x -> x + 1) [1;2;3]
```

Out[44]:

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

Map

In [45]:

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

Out[45]:

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Is there a problem with this implementation?

- Not tail recursive.
  - Generally not an issue with map over list.
  - Recursion depth bound by the size of the list.

rev_map

In [46]:

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

Out[46]:

```
val rev_map : ('a -> 'b) -> 'a list -> 'b list -> 'b list =
<fun>
```

In [47]:

```
let l = rev_map (fun x -> x + 1) [1;2;3] [] in
List.rev l
```

Out[47]:

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

Fold

- Fold is a function for combining elements.
- Fold is very powerful => very generic / difficult to understand.
- Let's take a simple example first.

In [48]:

```
let rec sum_of_elements acc l =
  match l with
  | [] -> acc
  | x::xs -> sum_of_elements (x + acc) xs

let sum_of_elements = sum_of_elements 0
```

Out[48]:

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

Out[48]:

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

In [49]:

```
sum_of_elements [1;2;3;4;5]
```

Out[49]:

```
- : int = 15
```

Fold

What is going on here?

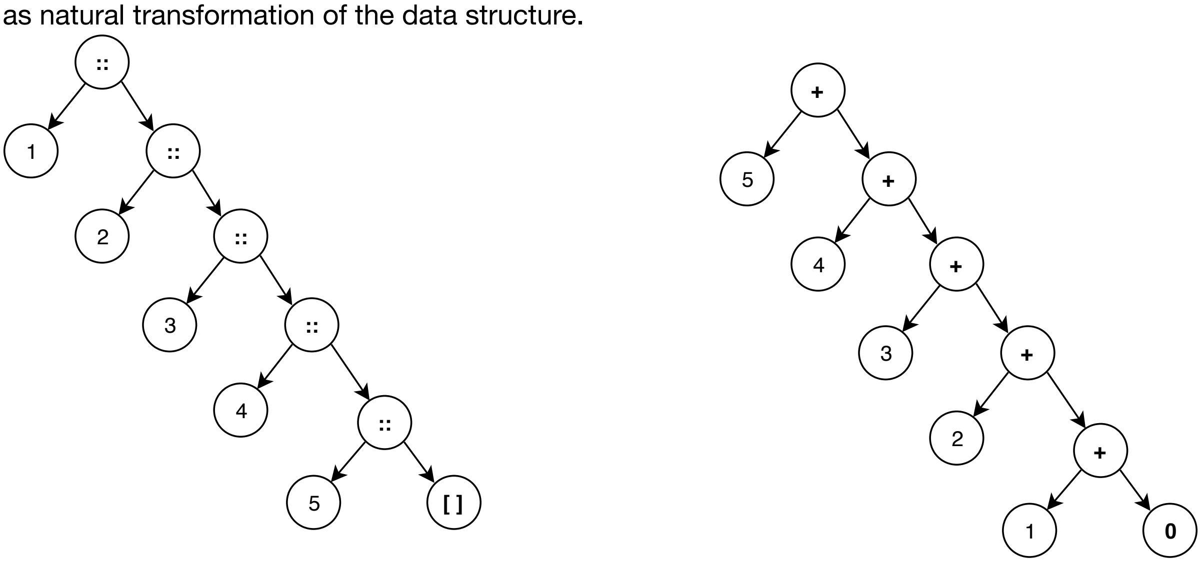
```
let rec sum_of_elements acc l =
  match l with
  | [] -> acc
  | x::xs -> sum_of_elements (x + acc) xs

let sum_of_elements = sum_of_elements 0
```

- There is **traversal** over the shape of the list.
- There is an `accumulator` which keeps track of the current sum so far.
- There is a function `+` that is applied to each element and accumulator.
- There is the `initial value` of the accumulator which is `0`.

Fold (left)

as natural transformation of the data structure.



Fold

In [50]:

```
List.fold_left
```

Out[50]:

```
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

- **First argument:** `('a -> 'b -> 'a)` is the function appplied to each element.
  - `'a` is accumulator and `'b` is current list element
- **Second argument:** `'a` is the initial value of the accumulator.
- **Third argument:** `'b list` is the list.
- **Result:** `'a` is the value of the accumulator at the end of the traversal.

Sum of elements using fold_left

```
let rec sum_of_elements acc l =
  match l with
  | [] -> acc
  | x::xs -> sum_of_elements (x + acc) xs

let sum_of_elements = sum_of_elements 0
```

In [51]:

```
List.fold_left (fun acc x -> acc + x) 0 [1;2;3;4;5]
```

Out[51]:

```
- : int = 15
```

In [52]:

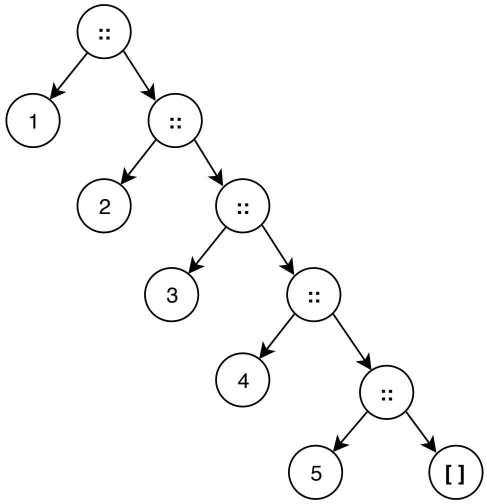
```
let rec fold_left f acc l =
  match l with
  | [] -> acc
  | x::xs -> fold_left f (f acc x) xs
```

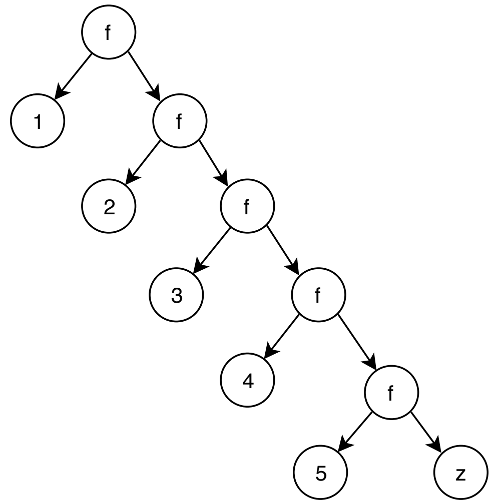
Out[52]:

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <
fun>
```

fold_right

Fold from the right.





fold_right

In [53]:

```
List.fold_right
```

Out[53]:

```
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

In [54]:

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

Out[54]:

```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
<fun>
```

- Not tail recursive!

Behold the power of fold

Any time you need to traverse the list, you can use `fold`.

In [55]:

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

Out[55]:

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

In [56]:

```
let length l = fold_left (fun acc _ -> acc + 1) 0 l
```

Out[56]:

```
val length : 'a list -> int = <fun>
```

In [57]:

```
let map f l = fold_right (fun x acc -> (f x) :: acc) l []
```

Out[57]:

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `map` is not tail recursive since `fold_right` is not a tail recursive function.

Exercise

Implement `exists : ('a -> bool) -> 'a list -> bool` function. `exists p l` returns `true` if there exists an element `e` in `l` such that `p e` is true. Otherwise, `exists p l` returns `false`.

In [58]:

```
let exists p l = failwith "not implemented"
```

Out[58]:

```
val exists : 'a -> 'b -> 'c = <fun>
```

In [59]:

```
assert (exists (fun e -> e = 0) [1;3;0] = true)
```

```
Exception: Failure "not implemented".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "[59]", line 1, characters 8-39
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Exercise

Implement `append : 'a list -> 'a list -> 'a list` using `fold_right`.

In [60]:

```
let append l1 l2 = failwith "not implemented"
```

Out[60]:

```
val append : 'a -> 'b -> 'c = <fun>
```

In [61]:

```
assert (append [1;2] [3;4] = [1;2;3;4])
```

```
Exception: Failure "not implemented".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "[61]", line 1, characters 8-26
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Fin.