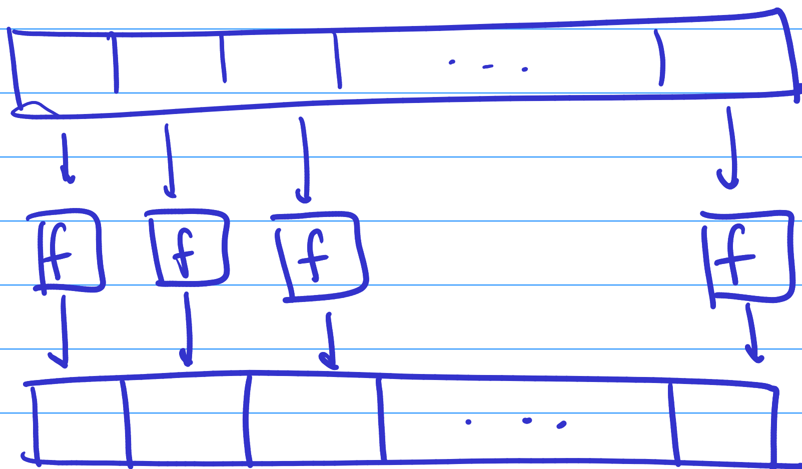
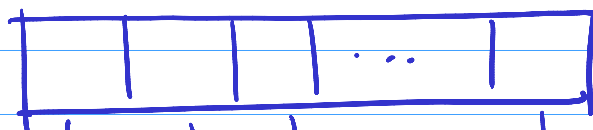
Functions over lists

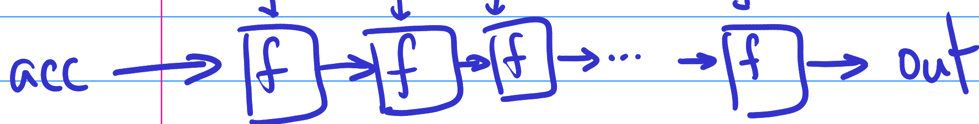
- List.map   map

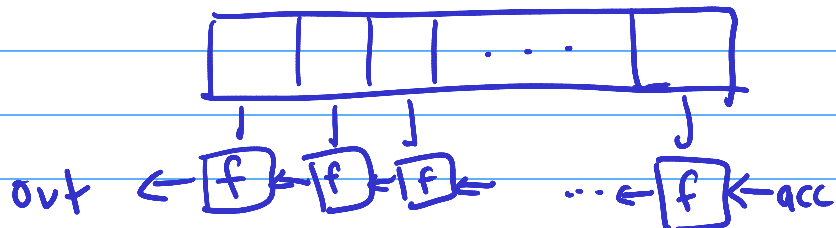


- List.filter   filter

- fold_left   fold_right

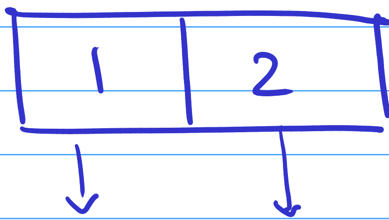


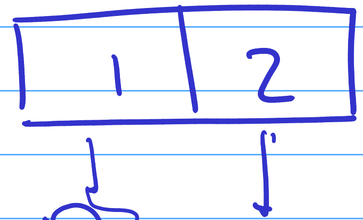


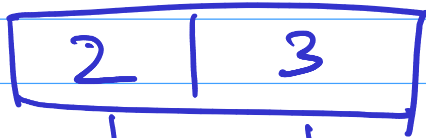


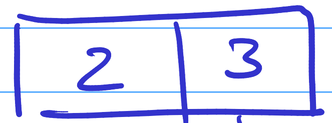
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

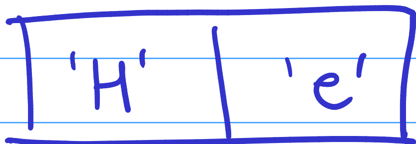
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

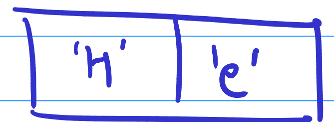












fold_left

10	20

10 → to_str, 20 → to_str

"" → ^ → ^ → "1020"

fold_right

10	20

10 → to_str, 20 → to_str

"2010" ← ^ ← ^ ← ""

Ex: Find length of a list.

```
46 l = some list
47 ans = 0
48 for x in l:
49       ans = ans + 1
50
51
52
53 List.fold_left (fun ans x -> ans + 1) 0 l
```

Ex: Find the ith element of a list

```
List.fold_left
  (fun acc x -> if fst acc = i
                then (fst acc + 1, Some(x))
                else (fst acc + 1, snd acc))
   (0, None)
   l
```

Folds are all you need!

Then, problem: Implement map using fold.

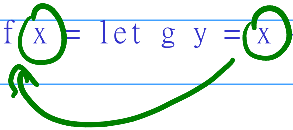
```
let map f l = List.fold_left (fun ans x -> ans @ [ f x ]) [] l
```

```
 map Char.uppercase_ascii [ 'H'; 'e'; 'l'; 'l'; 'o' ];;
- : char list = ['H'; 'E'; 'L'; 'L'; 'O']
```

Scoping rules

Static / Lexical scoping

```
let f x = let g y = x + y in g
```



But g "remembers" the value of x from when it was created.

The data structure that does the "remembering" is called the closure

```
─( 17:16:55 )─< command 66 >──────────────────────────{ counter: 0 }─
utop # let g1 = f 2;;
val g1 : int -> int = <fun>
─( 17:17:01 )─< command 67 >──────────────────────────{ counter: 0 }─
utop # let g2 = f 3;;
val g2 : int -> int = <fun>
─( 17:17:09 )─< command 68 >──────────────────────────{ counter: 0 }─
utop # g1 8;;
- : int = 10
─( 17:17:13 )─< command 69 >──────────────────────────{ counter: 0 }─
utop # g2 8;;
- : int = 11
```