List Module:

- `List.length ('a list)`
  - Returns type int
- `List.hd ('a list)`
- `List.tl ('a list)`
- `List.nth ('a list) n`
  - Returns n-th element in list
- `List.rev ('a list)`
- `List.flatten ('a list list)`
  - Converts to a normal list
- `List.map f ('a list)`
  - **Returns type 'b list**
  - `('a -> 'b) -> 'a list -> 'b list`
- `List.fold_left f acc ('b list)`
  - `f (...(f (f acc b1) b2) ...) bn)`
  - `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
- `List.fold_right f ('a list) acc`
  - `f a1 (f a2 (...(f an acc)...))`
- `List.mem a ('a list)`
  - Returns true or false
- `List.filter p ('a list)`
  - Returns all elements that satisfy the predicate p
  - `Example p: (fun x -> x mod 2 = 0)`
- `List.exists p ('a list)`
  - Return a bool value
- `List.assoc a (('a * 'b) list)`
  - If there is a pair (a,b), b is returned
- `List.split (('a * 'b) list)`
  - `Returns ('a list) * ('b list)`
- `List.combine ('a list) ('b list)`
  - `Returns ('a * 'b) list`
- `List.sort f ('a list)`
  - Sorts according to the comparison function f, which must return 0 for equal elements

A higher-order function should return a function!

```
(* for [1;2;3] the result is [1;3;6] *)
let psums lst =
  let rec helper l a =
    match l with
    | [] -> [a]
    | x::xs -> a::(helper xs (x + a))
  in
  match lst with
  | [] -> [0]
  | x::xs -> helper xs x;;
```

```
(* Flattens a list list to a normal list)
let smash l = List.fold_left (@) [] l;;
```

```
let rec inter item lst =
  match lst with
    | [] -> [[item]]
    | x::xs -> (item::lst)::(List.map (fun u ->
(x::u)) (inter item xs));;
```

```
let rec perms l =
  match l with
    | [] -> [[]]
    | x::xs -> smash (List.map (fun u -> (inter
x u)) (perms xs));;
```

```
(* Find trace of matrix, from class *)
let trace m =
  let rec helper m acc =
    match m with
    | [] -> acc
    | (x::xs)::rows ->
      helper (List.map (fun y::ys -> ys) rows)
(acc + x)
```

Key JAVA Notes:

- Method lookup at runtime is resolved by the **actual** type of an object (RHS)
  - `MyInt i = new GaussInt();`
- If the actual type of an object does not have the method, there's an error
- When there is **overloading**, it is resolved by the type-checker (declared type)
  - Finds the method of correct signature
  - Type-checking looks at **declared** types (LHS)
- Method lookup still needs to pass the type-checker
  - i.e. the declared type needs to contain the method

Mutable Closures:

- Example: `let a = ref 0`
- Update: `a := 1` equiv. to `a.contents <- 1`
  - a is NOT 1, a is the name of the memory cell (record with mutable cell) that stores integers
- Extracts stored value: `!a`
  - `# u := !u + 1;;`
  - `- : unit = ()`
- `x = y` : checks for equality of values (by dereferencing them)
- `x == y` : checks for structural equality (whether they are the same/ different records)
  - `let u = ref 7`
  - `let v = u`
  - `u == v`
  - `-: true`

- Alias: two names for the same memory cell (**¡watch out!** if one of the cell changes, the other cell will too)
    - `let u = ref 7 ;;`
    - `let v = u ;;`
    - `u := 8 ;;`
    - `!v AND !u`
    - `-: int = 8`

The basic update command has the form: `exp1 := exp2`

The evaluation rule:
1. First evaluate exp1 and verify that the result is a location.
2. Then evaluate exp2 and verify that the value obtained has the type appropriate to the location. Note, what gets stored are values, you cannot store unevaluated expressions.
3. Replace the contents of the location from step 1 with the value in step 2.
** an assignment destroys an old value, programmer has control over the lifetime of data (decides whether a value is needed any more and makes a choice to reuse a storage cell)
→ functional programming CANNOT do this

Records with multiple mutable fields

```
# type point = {mutable x: int; mutable y: int}
# let p = {x=3; y=4}
# p.x              - : int = 3
# p.x <- 5         - : unit = ()
# let move (p:point) (a,b) =
      (p.x <- p.x + a);(p.y <- p.y + b);;
val move : point -> int * int -> unit = <fun>
# move p (2,5)     - : unit = ()
# p                - : point = {x = 7; y = 9}
```

`** Do not put ref inside of the function **`

- the local bindings are thrown away so any changes to the ref variable is invisible → we need to see the side effect: `unit()`
- we need the binding to be trapped and not re-executed as a fresh binding everytime → the variable must be outside of the `fun()` and inside of the method name for it to be trapped inside of the environment

```
let flip =
 let c = ref 0 in
  fun () -> (c := 1 - !c); (Printf.printf "%i\n" !c)
```

<u>wrong way</u>

```
let flop = fun () ->
  let c = ref 0 in (c := 1 - !c);
```

- sequential composition and is done with one semicolon
- a sequence of commands ending with an expression, the last value is returned
- commands do not return a value (apart from ())

Imperative Banking:

```
let make_account(opening_balance: int) =
    let balance = ref opening_balance in
    fun (t: transaction) ->
      match t with
        | Withdraw(m) ->
            if (!balance > m)
            then ((balance := !balance - m);
    (Printf.printf "Balance is %i" !balance))
            else
            print_string "Insufficient funds."
        | Deposit(m) ->
           ((balance := !balance + m);
     (Printf.printf "Balance is %i\n" !balance))
        | Checkbalance -> (Printf.printf
"Balance is %i\n" !balance);;
```

Reversing a Linked List:

```
let reverse (lst: rlist) =
  let rec helper ((l: rlist), (acc: rlist)) =
    match !l with
      | None -> acc
      | Some c when !(c.next) = None ->
         (c.next := !acc; acc := (Some c); acc)
      | Some c ->
         (l := !(c.next); c.next := !acc; acc :=
(Some c); helper(l,acc))
  in (helper(lst, ref None));;
```

Inserts an element into a sorted linked list:

```
let rec insert comp (item: int) (list: rlist) =
  match !list with
  | Some {data = d; next = l} when comp (item,
d) = true ->
      list := Some {data = item; next =
cell2rlist {data = d; next = l}}
  | Some {data = d; next = l} when !l = None ->
      list :=  Some {data = d; next = cell2rlist
{data = item; next = ref None}}
  | Some {data = d; next = l} when comp (item,
d) = false -> insert comp item l
  | None -> list := Some {data = item; next =
ref None}
```

Insertion Sort:

```
let rec insert (n, l) =
  match l with
    | [] -> [n]
    | x::xs -> if (n < x) then n::(x::xs)
            else x::(insert(n, xs))

let rec in_sort l =
   match l with
    | [] -> []
    | x::xs -> insert (x, in_sort(xs))
```

Streams:

```
type 'a stream = Eos | StrCons of 'a * (unit ->
'a stream);;

let hdStr (s: 'a stream) : 'a =
  match s with
  | Eos -> failwith "headless stream"
  | StrCons (x,_) -> x;;

let tlStr (s : 'a stream) : 'a stream =
  match s with
  | Eos -> failwith "empty stream"
  | StrCons (x, t) -> t ();;

(* convert first n elements of a stream into a
list, useful to display part of a stream. *)
let rec listify (s : 'a stream) (n: int) : 'a
list =
  if n <= 0 then []
  else
    match s with
    | Eos -> []
    | _ -> (hdStr s) :: listify (tlStr s) (n -
1);;

(* n-th element of a stream *)
let rec nthStr (s : 'a stream) (n : int) : 'a =
  if n = 0 then hdStr s else nthStr (tlStr s) (n
- 1);;

(* make a stream from a list *)
let from_list (l : 'a list) : 'a stream =
  List.fold_right (fun x s -> StrCons (x, fun ()
-> s)) l Eos;;

let rec ones = StrCons (1, fun () -> ones);;

let rec nums_from n = StrCons(n, fun () ->
nums_from (n + 1));;

let nats = nums_from 0;;
```

Type Inference Examples:

Midterm Solution:

```
(* q1 *)
let rec repeated (f,n) =
  if (n = 0) then fun x -> x
  else
    fun x -> f ((repeated (f,n-1)) x);;

(* Rewritten q1 *)
let rec repeated (f,n) x =
  if (n = 0) then x
  else
    f ((repeated (f,n-1)) x);;

(* q2 *)
let square m =
  match m with
  | [] -> true
  | _ -> List.for_all (fun r -> (List.length m)
= List.length r) m;;
```

Quiz Solutions:

```
(Quiz 2)
let fst = fun x -> fun y -> x;;
let snd = fun x -> fun y -> y;;
```

- `snd 1 snd 2 "foo"`
    - `# -: string = "foo"`
- `fst 1 snd`
    - `# -: int = 1`

Higher-Order Functions:

```
let twice f = fun x -> f (f x);;
# twice twice
-: (fun f -> fun x -> f(f x)) twice
-: fun x -> twice (twice x)
```

which would give:

```
let fourtimes f = (twice twice) f;;
```

```
let double (x : int) : int = 2 * x
let square (x : int) : int = x * x
let twice f = fun x -> f (f x);;

# let quad (x : int) : int = twice (double, x)
      is equivalent to
# let quad (x : int) : int = double (double x)

# let fourth (x : int) : int = twice (square, x)
      is equivalent to
# let fourth (x : int) : int = square (square x)
```