Variables, Bindings, and Functions: in a local binding, the new value to the variable applies only in that scope.

```
# let m = 3 in
  let n = m * m in
  let k = m * m in
    k*n
;;
```

- A new binding adds the same variable on another stack level, that stack level is deleted after use.
- Declaring functions -> let name (parameters) = …
  - When declaring a function, all variable values are put on the stack
  - Must redefine function to get new bindings (or else it looks at old bindings)

Recursive Functions: In a tail recursive function, the compiler doesn't consume additional stack space for each recursive call. Instead of creating new stack frames with each call, reuse current frame

```
let rec fact n =
  if n < 0 then raise Domain
  else if n = 0 then 1
  else n * fact(n-1);;
```

Lists: Append function exists: "@"

Tree functions: in a tree -> type 'a tree = Empty | Node of 'a * 'a tree * a' tree

Insert:

```
let rec insert ((x,d) as e) t = match t with
  | Empty                -> Node(e, Empty, Empty)
  | Node ((y,d'), l, r) ->
      if x = y then Node(e, l, r)
      else
  (if x < y then Node((y,d'), insert e l, r)
   else
     Node((y,d'), l, insert e r))
```

Lookup: We use option because we might Have None returned. so "Some(d)"

```
let rec lookup x t = match t with
  | Empty                -> None
  | Node ((y,d), l, r) ->
  if x = y then Some(d)
    else
      (if x < y then lookup x l
   else lookup x r)
```

Induction:

- Inductive definitions give us a way of thinking recursively

Example:

Induction proof of SUM:

For any list "l" thefunction sum "l" correctly computes the sum of it's elements.

BASE CASE:
Empty list []
Sum of [] = 0

INDUCTIVE STEP:
IH: assume that for list "l", sum "l" is correct
Now with function prove that with one additional element, "x::l" is correct.

Sum(x::l) = x + sum(l), we know sum(l) is correct, therefore we get what we expect.

Higher order functions:

- Takes as input a function and outputs a function
- Ex: ('a -> 'b) -> 'a list -> 'b list :::: Takes a function 'a -> 'b and an input of type 'a list

Returning a function:

- Example: let add x = fun y -> x+y
  - Let add_five = add 5;;
  - Let result = add_five 10;;
  - Result = 15

Impossible tail recursion:

```
let rec merge lst1 lst2 =
  match lst1, lst2 with
  | [], lst | lst, [] -> lst
  | x1 :: xs1, x2 :: xs2 ->
     if x1 < x2 then x1 :: merge xs1 lst2
     else x2 :: merge lst1 xs2
```

Turning a function into tail recursive:

- Identify recursive call
- Introduce accumulator
- Find initial value for accumulator

```
(* TODO: Implement combined_dist_table: float list list -> float list *)
let rec combined_dist_table (matrix: float list list) =
  if is_empty matrix then
    []  (* Return an empty list if the distribution matrix is empty *)
  else
    List.fold_left (fun acc row ->
        List.map2 (fun a b -> a *. b) acc row
      ) (List.hd matrix) (List.tl matrix)
```

```
let rec to_instr e =
  match e with
  |FLOAT(x) -> [Float x]
  |PLUS(x, y) -> to_instr x @ to_instr y @ [Plus]
  |MINUS(x, y) -> to_instr x @ to_instr y @ [Minus]
  |MULT(x, y) -> to_instr x @ to_instr y @ [Mult]
  |DIV(x, y) -> to_instr x @ to_instr y @ [Div]
  |SIN(x) -> to_instr x @ [Sin]
  |COS(x) -> to_instr x @ [Cos]
  |EXP(x) -> to_instr x @ [Exp]
```