```
(*LISTS/RECURSION EXAMPLES*)

List.map (fun elem -> (*do something to each elem*)) lst
List.reduce (fun acc elem -> accumulator plus elem) lst
List.fold (fun acc elem -> accumulator plus elem) 0.0 lst
List.filter (fun elem -> boolean) list (*Returns a new collection with elements that return true*)
let isAllZeroes list = List.forall (fun elem -> elem = 0.0) list

let rec trace lol =
  let first lst =
    match lst with
    | [x] -> x
    | x::xs -> x
  let removefirst lst =
    match lst with
    | [x] -> []
    | x::xs -> xs
  let rec helper lol acc =
    match lol with
    | [x] -> first x + acc
    | x::xs -> helper (List.map (fun elem -> removefirst elem) xs) ((first x) + acc)
  helper lol 0

let rec shuffle l1 l2 =
 let rec insertEach item lst= (*returns a list of lists, the item inserted in each position of the input list*)
  match lst with
  | [] -> [[item]]
  | x::xs -> (item::lst) :: (List.map (fun elem -> x::elem) (insertEach item xs))
 match (l1,l2) with
  | ([],[]) -> [[]]
  | ([x], y::ys) -> insertEach x l2
  | (x::xs, [y]) -> insertEach y l1
  | (x::xs, y::ys) -> (List.map (fun elem -> x::elem) (shuffle xs l2)) @ (List.map (fun elem -> y::elem) (shuffle l1 ys))

let psums lst =
 let rec helper accum l =
  match l with
  | [] -> [accum]
  | x::xs -> accum :: helper (accum+x) xs

let smash ll = List.fold (@) [] ll

let rec perms lst =
 match lst with
 | [] -> [[]]
 | x::xs -> smash (List.map (fun elem -> inter x elem)) (*inter is the same as insertEach above*)

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

let rec isort lst =
  match lst with
  | [] -> []
  | x :: xs -> insert x (isort xs)

let rec remove(a,l) =
  match l with
  |        [] -> []
  |   x :: xs -> if (x = a) then remove(a, xs) else x::(remove(a,xs))

let rec remDup lst =
  match lst with
  | []  -> []
  | x :: xs -> x::(remove(x,remDup(xs)))

(*HIGHER ORDER FUNCTIONS EXAMPLES*)

Church Numerals:
let zero = fun f -> (fun x -> x)
let one = fun f -> (fun x -> (f x))
let two = fun f -> (fun x -> (f (f x)))
let showcn cn = (cn (fun n -> n + 1)) 0
let r1 = showcn one
let r2 = showcn two
let succ cn =  (fun f -> (fun x -> f ((cn f) x)))
let r3 = showcn (succ two)
let add n m = fun f -> (fun x -> ((n f) ((m f) x)))
let times n m = fun f -> (fun x -> (n (m f) x))
let exp n m = fun f -> (fun x -> (m n) f x)

```

```
Other examples:

let deriv (f, dx:float) = fun x -> ((f(x + dx) - f(x))/dx)

let rec iter_sum(f, lo:float, hi:float, inc) =
  let rec helper(x:float, result:float) =
    if (x > hi) then result
    else helper(inc(x), f(x) + result)
  helper(lo,0.0);;

let integral(f,lo:float,hi:float,dx:float) =
  let delta (x:float) = x+dx
  dx * iter_sum(f,(lo + (dx/2.0)), hi, delta)

(*IMPERATIVE EXAMPLES*)

type transaction = Withdraw of int | Deposit of int | CheckBalance

let make_protected_account(opening_balance: int, password: string) =
  let balance = ref opening_balance
  fun (p, t: transaction) ->
      if p = password then
        match t with
          | Withdraw(m) ->  if (!balance > m)
                            then
                              balance := !balance - m
                              printfn "The new balance is %i\n" !balance
                            else
                              printfn "Insufficient funds.\n"
          | Deposit(m) -> (balance := !balance + m; (printf "The new balance is %i\n" !balance))
          | CheckBalance -> (printf "The balance is %i\n" !balance)
      else printfn "Incorrect password\n"

let morgoth = make_account(1000)
let sauron = make_account(500)

(*ENVIRONMENTS EXAMPLES*)

let result2 =
  let x = 1 in
  let f = fun u -> (printf "Inside f, x is %i\n" x);(u + x) in
  let x = 2 in
  (printf "x is %i\n" x);f x (* answer is 3 *)

val foo = fn f => (fn n => (if (n = 0) then 1 else (f (n - 1))));
foo (fn n => (2 * n)) 5; (* answer is 8 *)

let val y = 1 in
  let val f = fn y => (y + y) in
    let val y = 2 in
      f(y)
    end
  end
end (* answer is 4 *)

let x = 1 in
  let f = (let u = 3 in (fun y -> u + y + x)) in
    let x = 2 in
    f x;; (* answer is 6*)

(*SUBTYPES/OOP*)

class Foo {
    private int a;
    public Foo(int n){ a = n;};
    public int showiv(){return a;};
    public void setiv(int n){a = n;};
}
class Bar extends Foo{
    private int b;
    public Bar(int n){ super(n+1); b = n;};
    public int showiv(){return b;}; (*example of overriding*)
}
public static void main(String[] args){
    Bar Bat = new Bar(3); (*bat has a=4 and b=3*)
    Foo Fox = Bat; (*declared as type Foo but has actual type Bar*)
    System.out.println(Bat.showiv()); (*runs Bar's showiv, 3*)
    System.out.println(Fox.showiv()); (*compiles only because Foo has a method showiv. Actually runs Bar's, 3*)
    Bat.setiv(7); (*uses foo's setiv because bar doesn't have one, set's bat's a to 7*)
    System.out.println(Bat.showiv()); (*unchanged, 3*)
    System.out.println(Fox.showiv()); (*unchanged, 3*)
}
```

```
Assume that A << B.
Anything asking B will be happy w. A
(a) Is (A → B) → A a subtyp of (B → B) → A?
        +   -    +       <<     +   -    +  (*first part is changed which is +, so its covariant*)
(b) Is A → (B → A) a subtyp of B → (B → A)?
       -    -   +      >>      -    -   + (*first part changed, which is -, so contravariant*)



(*STREAMS*)

let nat = Seq.initInfinite (fun i -> i)
let cons x sigma = Seq.append (Seq.singleton x) sigma
let first sigma = Seq.nth 0 sigma
let rest sigma = Seq.skip 1 sigma
let rec prefix (n: int) sigma =
  if (n = 0) then []
  else (first sigma) :: (prefix (n - 1) (rest sigma))
let rec addFloatStreams (s1:seq<float>) s2 =
  Seq.delay (fun () -> cons ((first s1) + (first s2)) (addFloatStreams (rest s1) (rest s2)))
let power n =
  let rec helper exp= Seq.delay(fun () -> Seq.append (Seq.singleton (n*exp)) (helper(exp*n)))
  helper 1
let rec expSeries =
  Seq.delay (fun () -> (Seq.initInfinite (fun i -> Term((1.0/float(fact i)),i))))
let rec numsFrom n = cons n (Seq.delay (fun () -> (numsFrom (n + 1))))
let rec sieve sigma =
  Seq.delay (fun () ->
                let head = first sigma
                cons head (sieve (Seq.filter (fun n -> (n % head) <> 0) (rest sigma))))
let primes = sieve (numsFrom 2)
```