**List Library Functions**

**List.fold**: Applies a function f to each element of the collection, thread an accumulator argument. Returns the final result.   
let data = [(“cats”, 4); (“dogs”, 4); (“mice”, 3)]

let count = List.fold (fun acc (nm, x) -> acc + x) 0 data

(nm, x): format of each element in the data

val count = 12

**List.map:** creates a new collection whose elements are the results of applying the given function to each of the elements of the collection.

let data = [1;2;3;4]

let r1 = data |> List.map(fun x -> x+ 1)

val r1 = [2;3;4;5]

**List.filter:** Returns a new collection containing only the elements of the collection for which the given predicate returns **true**.

let evenOnlyList = List.filter (fun x -> x %2 = 0) [1; 2; 3; 4; 5; 6]

val: it = [2; 4; 6]

**List.forall:** Tests if all elements of the collection satisfy the given predicate.

let matrixcheck m = List.forall(fun x -> List.length x = List.length m) m;; => output: true/false

**List.iter**: applies the given function to each element of the collection. (no accumulator, apply the function on each element)

**List.Length** lst and **List.nth** lst 0 / **List.item** 0 lst

**Extra Library Functions**:

**List.fold2**: Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes.

**List.map2**: Creates a new collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise.

**List.reduce**: If the input function is f and the elements are i0...iN, then this function computes f i0 (...(f iN-1 iN)). Accumulator is the last element of the list. (reverse if List.fold)

List.reduceBack (fun acc elem -> acc + elem) list

let list1 = [1; 2; 3], let list2 = [4; 5; 6]

let sumList = List.map**2** (fun x y -> x + y) list1 list2

=> [5; 7; 9]

let rec shuffle (l1, l2) =

match l1, l2 with

| [], [] -> [[]]

| [], y::ys -> [l2]

| x::xs, [] -> [l1]

| x::xs, y::ys -> (List.map(fun l -> x::l) (shuffle(xs, l2)))

@ List.map(fun l -> y:: l) (shuffle(l1, ys))

shuffle([1;2;3], [4;5])

[[**1**,2,3,4,5],[**1**,2,4,3,5],[1,2,4,5,3],[1,4,2,3,5],[1,4,2,5,3],[1,4,5,2,3],  
   [**4**,1,2,3,5],[**4**,1,2,5,3],[**4**,1,5,2,3],[**4**,5,1,2,3]]

**List Examples:**

let bit n =

if (n = 0) then [[]]

elif (n =1) then [[0]; [1]]

else

List.fold(fun acc el -> acc @ [0::el, 1::el ])[] bits(n-1)

n = 1 🡺 [0]; [1]

n = 2 🡺 0**0** 0**1**  [0::bit(1); 1::bit(1)] for bit(1)

0**1** 1**1** [[0; 0]; [1;0]; [0;1]; [1;1]]

Another approach:

let rec bits n =

if n = 0 then [[]]

elif = 1 then [[0]; [1]]

else

let prev = bits(n-1)

let l1 = List.map(fun l -> 0::l) prev

let l2 = List.map(fun l -> 1::l) prev

l1 @ l2

🡺 [ (l1) [0; 0]; [0;1]; (l2) [1;0];[1;1]] for n =2

let smash ll = List.fold(@) \*[] ll \*for every element in ll, append it to the empty list [].

[[1;2;3]; [4;5]; [6]] 🡺 [1;2;3;4;5;6]

let rec inter item lst =

match lst with

| [] -> [[item]]

| x::xs -> ­­­­­­1(item::lst) :: (List.map(fun u -> (2x::u)) (inter item xs))

1 : always put the item in front of the list

2 : after inserting the item, put the orginal head in front of the list

inter 0 [1;2;3] => [[0; 1; 2; 3]; [1; 0; 2; 3]; [1; 2; 0; 3]; [1; 2; 3; 0]]

let rec perms l =

match l with

| [] -> [[]]

| x::xs -> smash (List.map(fun u -> (inter x u)) (perms xs))

perms [1;2;3] 🡺 [[1;2;3]; [2;1;3]; [2;3;1]; [1;3;2]; [3;1;2]; [3;2;1]]

1st recursion, take take 2 take 3   
 1 and place it everywhere

let trace m =

let rec helper lst acc =

match lst with

| [] -> acc

| (1**x::xs**)::rest -> helper(List.map(fun (1y::ys) -> ys) rest) (x+acc)

helper m 0

1: pattern on the nested list.

trace [[1;2;3]; [4;5;6]; [7;8;9]]

* 1st recursion: take the first element of first list: acc = 1
* 2nd recursion: [4;5]; [8;9] : acc = 1 + 4 = 5
* 3rd recursion: [9]: acc = 5 + 9 = 14

let rec pascalTriangle n =

if n = 0 then [1]

else

let rrow = pascal(n-1) @ [0]

let rprime = 0::pascal(n-1)

List.fold**2**(fun acc rr rp -> (rr+rp)::acc) [] rrow rprime

//Determined if its sorted in descending order

let rec descendCheck l =   
    match l with   
    | [] -> true  
    | [x] -> true  
    | x::y::xs -> if (x < y) then false  
                  else  
                      descendCheck(y::xs)

// the bad way of doing the trace

let trace (matrix: int list list) =  
    let rec helper(m, index, sum) =   
        match m with  
        |[] -> sum  
        |[[]] -> sum   
        | x::xs -> let element = List.item index x  
                   helper(xs, index+1, sum + element)  
    helper(matrix, 0, 0)

**Higher Order Function**

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 r0 = shownch zero 🡺 0

= ((fun f -> (fun x -> x)) ((fun n -> n+1))

= (fun f -> (fun x -> x)) 0 => 0

let r1 = showncn one 🡺 1

= (fun f -> fun x -> f x) ( fun n -> n +1)) 0

= (fun f -> fun x -> fun n -> n + 1) 0

=> fun f -> fun x -> 1 => 1

val repeat: f: (‘a -> ‘a)-> n: int ->(‘a -> ‘a)

let repeat f n =

if n = 0 then fun x -> x

else 1fun x -> f ((repeat f (n-1)) x )

let inc n = n + 1

let zero = repeat inc 0 => zero 3 => 3 (fun x -> x) 3 => 3

let plustwo = repeat inc 2

plustwo 3 => 5

fun x -> i+1 (repeat i+1 1) x

-> repeat i+1 0

-> fun x -> x

(fun x -> (i+1)+1) 3 => fun 3 -> 3+1+1 => 5

//sum up the intervals

*Don’t do this!*

let x = ref 1

let y = x

let incr n = ( n := !n +1)

incr x

so, !x is 2 and !y is also 2 because y is pointing to x

Do this:

let x = ref 1

let z = ref (!x)

incr z

> !z => 2

> !x => 1

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 max\_so\_far =   
    let 1mutable max = 0  
    let 1mutable counter = 0  
    fun 2**n** -> if (counter = 0) then   
            max <- n

counter <- counter + 1

max  
             else  
                if (max < n) then  
                    max <- n  
                    max  
                else max

let max\_so\_far =   
    let 1max = ref 0  
    let 1counter = ref 0  
  fun 2**n** -> if (!counter = 0) then   
         max := n

counter := !counter + 1

!max   
             else  
                if (!max < n) then  
                    max := n  
                    !max  
                else !max

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)

//width \* f(x)

E.g

let r\_sq(x:float):float = x \* x

calling: integral (r\_sq, 0.0, 1.0, 0.001)

let conv f g =

(1fun x -> integrate ((fun a -> f (x-a) \* g(a)) 0 x 0.0001))

1: create the *new* function “convolution” from the params f and g.

let rec apply\_list l = //apply the list of functions

match l with

| [] -> (fun x -> x)

| f::fs -> (fun x -> f ((apply\_list(fs) x)))

//sum up of the function values from a to b

let rec sum\_inc (f, a, b, inc) =

if (a > b) then 0

else (f a) + sum\_inc(f, (inc a), b, inc)

val: (f: (‘a -> ‘a) \* a: ‘a \* b: ‘a \* inc: (‘a -> ‘a) ) -> int

**Mutable variables**

let mutable x = 1

x <- x + 1

<- : assign new values to mutable variables by using the <- operator.

Reference Cells

You use the **ref** operator before a value to create a **new reference** cell that **encapsulates the value**. You can then **change the underlying value because it is mutable**.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Definition** |
| **!** (dereference operator) | Returns the underlying value. | let (!) r = r.contents |
| **:=** (assignment operator) | Changes the underlying value. | let (:=) r x = r.contents <- x |
| **ref** (operator) | Encapsulates a value into a new reference cell. | let ref x = { contents = x } |
| **Value** (property) | Gets or sets the underlying value. | member x.Value = x.contents |
| **contents** (record field) | underlying value of the reference cell | let ref x = { contents = x } |

// Declare a reference.

let refVar = ref 6

// Change the value referred to by the reference.

refVar := 50

// Dereference by using the ! operator.

printfn "%d" !refVar // !refVar is the same as saying refVar.contents

1: the variable is **trapped** inside the function, because they are the **closure** for the function body. The closure re,e,bers the environment that exists when the function is defined.

2n: argument

let make\_protected\_account(opening\_balance: int,password: string) =  
    let balance = ref opening\_balance  
    let passcode = ref password  
    fun (pass: string, t: transaction) ->  
        if (pass <> !passcode) then  
            printfn "Password is incorrect"  
        else  
            match t with  
            | Withdraw(m)-> if (!balance >= m) then  
 balance := !balance - m  
 printfn "Balance is %i" !balance  
 else  
 printfn "Insufficient funds. You have only %i" !balance  
| Deposit(m)  > (balance := !balance + m; (printf "Balance is %i\n" !balance))  
| CheckBalance -> (printf "Balance is %i\n" !balance)

**OOP: Subtyping and Inheritance**

Rules for method lookup and Type checking

Two phases:

compile time: when the type checking is done

run time: when method lookup happens

Compile time is before run time

* The type checker has to say that a method is OK at compile time
* All type checking is done based on what the declared type of a reference to an objet is.
* Subtyping is an integral part of type checking. This means if B is a subtype of A and there is a context that gets a B where A was expected, there will be no type error
* [Overriding] Method lookup is based on the actual type (type associated at run time) of the object and not the declared type of the reference.
* When there is overloading (as opposed to overriding), this is resolved by type checking

super: - use the constructor for the super class

* Invoke the super class constructor.

**Overriding**: - The show() method in myInt is no use for guassInt class. So you create another show() in guassInt.

- The show() in myInt is *hidden*

**Overloading**: In myInt: add(myInt N)

In gaussInt: add(gaussInt z)

* Type arguments for the two methods are different
* With different signature, we get two different methods with the same name.
* Both methods are available from the subclass
* The system looks at the types of the actual arguments and decides which one to use.

Example:

gaussIntz = new gaussInt(3,4)

gaussInt w; // object has been created

myInt b = z // b and z are the name for the same object (run-time)

z.show() and b.show(): both show the same thing.

Compile time: “b” is declared myInt, and myInt has the method show(), so it passes the type checking.

Run time: b’s actual type at run time is gaussInt, so it uses the show() in gaussInt.

myInt d = b.add(b) //type check “b” is myInt

- At compile time: b has the method add(), so it passes the type check. The system thinks that b is of type myInt.

When overloading: the type checker tells b to use the add() in myInt class because at compile time, the system thinks “b” is the type myInt.

d.show() : // d has type myInt => 6

w = z.add(b) 🡺 //not type checked 🡺 overloading

1. At compile time, z can use 2 method add()
2. One accepts gaussInt, another accepts myInt type
3. So z will take the add() from myInt because “b” (the argument) is the type myInt . The add() will return myInt
4. w = gaussInt and z.add(b) = myInt 🡺 error

w = b.add(z) //not type check

1. Overloading: the system thinks b has type myInt, so “b” will use add() from myInt, which returns myInt
2. “z” can be passed into add(myInt) because gaussInt is a subtype of myInt.
3. w -> gaussInt and b.add(z) -> myInt 🡺 error

w = ((gaussInt) b).add(z) //not type check

1. “b” is type gaussInt. “b” can choose which add() method to use. Since the argument “z” has type guassInt, system will tell “b” to use the add() in gaussInt.
2. w is declared gaussInt, so w can be associated to the right hand side.

w.show() // real: 6 imaginary part: 8

myInt a = new myInt(3)

myInt c = z.add(a)

1. Overloading: z is type gaussInt and it can use 2 add() functions. Since the passing argument “a” is type myInt, z will call the add() in myInt and return myInt.

c.show() // 6

**Subtyping**

Subtype: A <| B (A is a subtype of B)

If my computation wants B value, it will be happy with an A value.

Contract: a type declaration signs a contract

Given: int <| float

**Covariant (output)**: These rules say that these type constructors *preserve* the subtyping relation.

int --> **int** <| int --> **float**

If I want to get a float from my contract, it is also ok to get an int from another contract (replacement).

**Contravariant (input)**: it looks like the relation has gotten *flipped.*

**float** --> int <| **int** --> int

Pass in an int does not bother the fact that the gadget deals with float-> int. I want the function to have type A, but additional fields in B should not matter

Violation: int --> int <| float --> int

You don’t know how to deal with the float input

**Function** types are contracts.

F: T1 🡪 T2: calling context promises to supply values of type T1 and expects values of type T2 in return

Subtype polymorphism:

Subtyping is a relation between types (interfaces in OOP parlance)

Inheritance is a relation between implementation steeming from a language feature that allows new objects to be created from existing ones.

The actual plotting code does not care how y was obtained => interface plottable.

Plot method is declared to accept an object of type plottable.

(A🡪 B) 🡪 A <| (B🡪B) 🡪 A? True

(B🡪B) <| A 🡪B? Yes, it satisfies the condition of contravariant.

(B 🡪 B) = C and A 🡪B = D 🡺 C <| D 🡺 D 🡪 A <| C 🡪 A

A 🡪 (B 🡪 A) <| B 🡪 (B 🡪 A)? False

It’s a violation for the input. A contract handles B, but its replacement can only handle A, which is a subtype of B.

**Subtype Formal Proof**

T1 <| S1 S2 <| T2

(S1 -> S2) <| (T1 -> T2)

a) (A🡪 B) 🡪 A <| (B🡪B) 🡪 A?

(B -> B) <| (A->B) A <| A

(A ->B)->A <| (B->B)->A

A <| B B <| B => Contravariant

(B -> B) <| (A->B)

=> True.

**Type Derivation (Informal)**

Function type = input -> output

🡺let S = fun x -> (fun y -> (fun z -> (x z) (y z)))   
x: α y: β z: γ (x z) (y z): Θ

(x z): z is the input type for x. Then we can assume that α = γ -> δ, where δ is the return type for (x z)

(y z): z is also the input for y, then we can assume that β = γ -> ϕ, where ϕ is the return type for (y z)

(x z) (y z): the return type for (y z) becomes the input for (x z). Hence we have δ = ϕ -> Θ

So, we have S = α -> β -> γ -> Θ

= (γ -> δ) -> (γ -> ϕ) -> γ -> Θ

S = (γ -> ϕ -> Θ) -> (γ -> ϕ) -> γ -> Θ

🡺 let rec apply\_list l =

match l with

|[] -> fun x -> x

| f::fs -> (fun x -> apply\_list(fs) (f x))

l: α apply\_list: α -> β

Base case: we know that *l* is a list. We call it α = δ-list. The return type is a function (fun x-> x): γ -> γ. Hence, β = γ -> γ

Second clause: from (f x), we see that f is a function and x is the input. Then, we can deduce f: γ -> Θ, where Θ is the return type for (f x). After, Θ is fed in to function type being returned by apply\_list, which has type γ -> γ. This means, Θ= γ. Hence, f = γ-> γ = δ.

Thus, we get α -> β = δ-list -> γ -> γ = = (γ-> γ)-list -> γ-> γ.

🡺 let rec map = fun f -> fun x -> if ( x= []) then [] else f (head(x))::(map f ( tail(x)))

f: α x: β

from x = [ ], we see β = γ-list. From f (head(x)), we see f is a function and head(x): γ so α = γ -> δ. δ is fresh. Since f (head(x)) = δ so

f (head(x))::… => δ-list

map: α-> β = (γ -> δ) -> γ-list -> δ-list

🡺 let rec append (l1, l2) =

match l1 with

| [] -> l2

| x::xs -> x: (append )