Formulario F#

1 Funzioni

```
let nomeFunzione arg1 arg2 = <corpo funzione>
let rec nomeFunzioneRicorsiva arg1 arg2 = <corpo funzione>
fun arg1 arg2 -> <corpo funzione> // funzione anonima
```

2 Liste

1;2;3 $\rightarrow$ head::tail, head = 1, tail = [2;3]

- Usiamo @ per appendere una **lista** in coda ad un'altra, usiamo :: per concatenare un **elemento** in testa ad una lista

3 Pattern Matching

```
match x with
| 1 -> "Uno"
| 2 -> "Due"
| _ -> "Altro"
```

Si può usare il `when` per aggiungere condizioni

```
match x with
| x when x > 0 -> "Positivo"
| x when x < 0 -> "Negativo"
| _ -> "Zero"
```

4 Eccezioni

4.1 Per lanciarle

```
[altro codice] failwith "Messaggio di errore"
```

4.2 Per gestirle

```
try
    [codice che potrebbe lanciare eccezioni]
with
    | e -> printfn "Errore: %s" e.Message
```

5 Tipi

```
type nomeTipo = | Costruttore1 [of tipo1 * tipo2 * ...] | ...
```

6 Esempi di funzioni ricorsive e con accumulatore

Accoppiamento elemento indice

```
// Senza accumulatore
let rec pairWithIndex list index=
    match list with
    | [] -> []
    | head :: tail -> (head, index) :: pairWithIndex tail (index+1)
// Con accumulatore
let pairWithIndexAcc list =
    let rec loop list acc =
        match list with
        | [] -> []
        | head :: tail -> (head, acc) :: loop tail (acc + 1)
    loop list 1

let indexed l =
    let rec aux i l =
        match l with
        | hd::tl -> (i, hd) :: aux (i + 1) tl
        | _ -> []
    aux 0 l

let mapi_indexed l = List.mapi (fun i h -> (i, h)) l
```

Somma elemento per elemento

```
// Senza accumulatore
let rec sum_wise list1 list2 =
    match list1, list2 with
    | [], [] -> []
    | head1 :: tail1, head2 :: tail2 -> (head1 + head2) :: sum_wise tail1 tail2
    | _, _ -> failwith "Lists must have the same length"
// Con accumulatore
let sum_wise_acc list1 list2 =
    let rec loop list1 list2 acc =
        match list1, list2 with
        | [], [] -> List.rev acc
        | head1 :: tail1, head2 :: tail2 -> loop tail1 tail2 ((head1 + head2) :: acc)
        | _, _ -> failwith "Lists must have the same length"
    loop list1 list2 []
```

Selezione elementi in base a condizione

```
// Senza accumulatore
let rec select comparer list1 list2 =
    match list1, list2 with
    | [], [] -> []
    | head1 :: tail1, head2 :: tail2 ->
        if comparer head1 head2 then
            head1 :: select comparer tail1 tail2
        else
            head2 :: select comparer tail1 tail2
    | _, _ -> failwith "Lists must have the same length"
// Con accumulatore
let select_acc comparer list1 list2 =
    let rec loop list1 list2 acc =
        match list1, list2 with
        | [], [] -> List.rev acc
        | head1 :: tail1, head2 :: tail2 ->
            if comparer head1 head2 then
                loop tail1 tail2 (head1 :: acc)
            else
                loop tail1 tail2 (head2 :: acc)
        | _, _ -> failwith "Lists must have the same length"
    loop list1 list2 []
```

Unione di liste applicando funzione f

```
// Senza accumulatore
let rec combine f l1 l2 =
    match l1, l2 with
    | h1::t1, h2::t2 -> (f h1 h2)::(combine f t1 t2)
    | [], [] -> []
    | _ -> raise (System.ArgumentException("combine"))
// Con accumulatore
let accCombine f =
    let rec aux acc l1 l2 =
        match l1, l2 with
        | h1::t1, h2::t2 -> aux (f h1 h2 :: acc) t1 t2
        | [], [] -> List.rev acc
        | _ -> raise (System.ArgumentException("combine"))
    aux []
```

Somma elementi lista (due elementi alla volta)

```
// Senza accumulatore
let rec agg = function
    | h1::h2::t -> h1+h2::agg t
    | l -> l
// Con accumulatore
let acc_agg =
    let rec aux acc = function
        | h1::h2::t -> aux (h1+h2::acc) t
        | [h] -> List.rev (h::acc)
        | _ -> List.rev acc
    in aux []
```

Duplicazione elementi lista

```
// Senza accumulatore
let rec dup = function
| h::t -> h::h::dup t
| _ -> []
// Con accumulatore
let accDup list =
    let rec aux acc = function
        | h::t -> aux (h::h::acc) t
        | _ -> acc
    List.rev (aux [] list)
```

Rimuovi in posizione i

```
// Senza accumulatore
let rec remove i l =
    match l with
    | hd :: tl -> if i = 0 then tl else hd :: remove (i - 1) tl
    | _ -> []
// Con accumulatore
let accRemove i l =
    let rec loop acc i l =
        match l with
        | hd :: tl -> loop (if i = 0 then acc else hd :: acc) (i - 1) tl
        | _ -> List.rev acc

    loop [] i l

// Versione nostra
let rec remove l i =
    match i,l with
    | i, _ when i < 0 || i >= List.length l -> l
    | 0, _ :: tl -> tl
    | _, hd :: tl -> hd :: remove tl (i - 1)

let removeAcc l i =
    if i < 0 || i >= List.length l then l
    else
        let rec loop l i acc =
        match i,l with
        | 0, _ :: tl -> acc @ tl
        | _, hd :: tl -> loop tl (i - 1) (hd :: acc)
    loop l i []
```

Inserisci in posizione i (con e senza eccezione)

```
// Senza accumulatore
let rec insert el index list =
    match index, list with
    | 0, _ -> el::list
    | _, [] -> failwith "Index out of bounds"
    | _, hd::tl -> hd::insert el (index-1) tl
// Con accumulatore
let insert list item pos =
    if pos < 0 || pos > List.length list then failwith "Index out of bounds"
    else
        let rec loop list p acc =
            match p, list with
            | 0, _ -> acc@(item::list)
            | p, hd::tl -> loop tl (p-1) (acc@[hd])
        loop list pos []
```

Applica una funzione f agli elementi di due liste

```
let rec map2 f l1 l2 =
    match l1, l2 with
    | hd1::tl1, hd2::tl2 -> f hd1 hd2 :: map2 f tl1 tl2
    | [], [] -> []
    | _ -> invalidArg "l2" "List lengths do not match"

let map2der f l1 l2 = List.map (fun (x,y) -> f x y) (List.zip l1 l2)
```

Pairwise

```
let rec pairwise lst =
    match lst with
    | x::y::tl -> (x, y)::pairwise tl
    | _ -> []

let acc_pairwise lst =
    let rec aux acc lst =
        match lst with
        | x::y::tl -> aux ((x, y)::acc) (y::tl)
        | _ -> List.rev acc
    aux [] lst
```

Lista di lunghezza i con stesso elemento el

```
let repl (el: 'a) (i: int) =
    let rec aux i =
        if i <= 0 then []
        else el :: aux (i - 1)
    aux i

let acc_repl (el: 'a) (i: int) =
    let rec aux acc i =
        if i <= 0 then acc
        else aux (el :: acc) (i - 1)
    aux [] i
```

Prodotto scalare

```
let rec scalar (l1: int list) (l2: int list) =
    match l1, l2 with
    | hd1::tl1, hd2::tl2 -> hd1 * hd2 + scalar tl1 tl2
    | [], [] -> 0
    | _, _ -> failwith "scalar"

let acc_scalar (l1: int list) (l2: int list) =
    let rec aux acc l1 l2 =
        match l1, l2 with
        | hd1::tl1, hd2::tl2 -> aux (hd1 * hd2 + acc) tl1 tl2
        | [], [] -> acc
        | _, _ -> failwith "scalar"
    aux 0 l1 l2
```

Somma elementi lista con applicazione di una funzione a tutti gli elementi

```
let rec sumBy f l=
    match l with
    | hd::tl -> f hd + sumBy tl
    | _ -> 0

let accSumBy f l=
    let rec aux acc l=
        match l with
        | hd::tl -> aux (f hd + acc) tl
        | _ -> acc
    aux 0 l
```