settembre 2025

2. Sia `gen: (int -> 'a) -> int -> 'a list` la funzione tale che

   `gen f n = f n::f (n-1)::...::f 1::[]`

   Se $n \leq 0$, allora viene restituita la lista vuota. Esempi:

```
assert (gen (fun x -> x * x) 5 = 25 :: 16 :: 9 :: 4 :: 1 :: [])
assert (gen (fun x -> x - 1) 5 = 4 :: 3 :: 2 :: 1 :: 0 :: [])
assert (gen (fun x -> x - 1) 0 = [])
assert (gen (fun x -> x - 1) -5 = [])
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.
   Non è consentito usare funzionalità avanzate di F# o funzioni predefinite di `List`, inclusa `List.rev`.

luglio 2025

```
let gen f n =
    let rec aux i =
        if i > n then [] else f i :: aux (i + 1)

    aux 1

assert (gen (fun x -> x * x) 5 = 1 :: 4 :: 9 :: 16 :: 25 :: [])
assert (gen (fun x -> x - 1) 5 = 0 :: 1 :: 2 :: 3 :: 4 :: [])
assert (gen (fun x -> x - 1) 0 = [])
assert (gen (fun x -> x - 1) -5 = [])

let accGen f =
    let rec loop acc i =
        if i <= 0 then
            acc
        else
            loop (f i :: acc) (i - 1)

    loop []

assert (accGen (fun x -> x * x) 5 = 1 :: 4 :: 9 :: 16 :: 25 :: [])
assert (accGen (fun x -> x - 1) 5 = 0 :: 1 :: 2 :: 3 :: 4 :: [])
assert (accGen (fun x -> x - 1) 0 = [])
assert (accGen (fun x -> x - 1) -5 = [])
```

giugno 2025

2. Sia `swap: ('a -> 'a -> bool) -> ('a list -> 'a list)` la funzione tale che

   `swap f ls1 = ls2`

   dove `ls2` è ottenuta da `ls1` scambiando tutti e soli i suoi elementi contigui `el1` ed `el2` tali che `f el1 el2` restituisce `true`. Esempi:

```
assert (swap (<) (1 :: 2 :: 3 :: []) = 2 :: 3 :: 1 :: [])
assert (swap (<) (2 :: 3 :: 1 :: []) = 3 :: 2 :: 1 :: [])
assert (swap (<) (3 :: 2 :: 1 :: []) = 3 :: 2 :: 1 :: [])
assert (swap (>) (1 :: []) = 1 :: [])
assert (swap (>) (1 :: 3 :: 2 :: 4 :: 0 :: []) = 1 :: 2 :: 3 :: 0 :: 4 :: [])
```

   Notare che uno stesso elemento può essere scambiato di posto più volte, come accade nel primo esempio.

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.

Nota bene: **non è ammesso** usare funzionalità avanzate di F# o funzioni predefinite di `List` eccetto `List.rev`.

gennaio 2025 parziale

2. Sia `split: ('a -> 'b option) -> ('a list -> 'b list * 'a list)` la funzione tale che

   `split f ls = (ls1,ls2)`

   dove `ls1` e `ls2` rispettano l'ordine degli elementi di `ls`, `ls1` contiene tutti i valori v tali che `f e = Some v` per un elemento `e` di `ls` e `ls2` tutti gli elementi `e` di `ls` tali che `f e = None`.

   Esempio:

```
assert
  (split(fun x->if x<0.0 then None else Some(sqrt x))(1::-4::4::-1::[])=
     (1::2::[],-4::-1::[]))
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.

Nota bene: **non è ammesso** usare funzionalità avanzate di F# o funzioni predefinite di `List` eccetto `List.rev`.

```
let split f =
    let rec aux =
        function
        | hd :: tl ->
            let values, nones = aux tl

            match f hd with
            | Some v -> v :: values, nones
            | None -> values, hd :: nones
        | [] -> [], []

    aux

assert
    (split (fun x -> if x < 0.0 then None else Some(sqrt x)) (1 :: -4 :: 4 :: -1 :: []) = (1 :: 2 :: [],

let accSplit f =
    let rec loop values nones =
        function
        | hd :: tl ->
            match f hd with
            | Some v -> loop (v :: values) nones tl
            | None -> loop values (hd :: nones) tl
        | [] -> List.rev values, List.rev nones

    loop [] []

assert
    (accSplit (fun x -> if x < 0.0 then None else Some(sqrt x)) (1 :: -4 :: 4 :: -1 :: []) = (1 :: 2 ::
                                                                                              -4 :: -1 :
```

```
let gen f  =
    let rec aux i =
        if i <= 0 then
            []
        else
            f i :: aux (i - 1)

    aux

assert (gen (fun x -> x * x) 5 = 25 :: 16 :: 9 :: 4 :: 1 :: [])
assert (gen (fun x -> x - 1) 5 = 4 :: 3 :: 2 :: 1 :: 0 :: [])
assert (gen (fun x -> x - 1) 0 = [])
assert (gen (fun x -> x - 1) -5 = [])

let accGen f n =
    let rec loop acc i =
        if i > n then
            acc
        else
            loop (f i :: acc) (i + 1)

    loop [] 1

assert (accGen (fun x -> x * x) 5 = 25 :: 16 :: 9 :: 4 :: 1 :: [])
assert (accGen (fun x -> x - 1) 5 = 4 :: 3 :: 2 :: 1 :: 0 :: [])
assert (accGen (fun x -> x - 1) 0 = [])
assert (accGen (fun x -> x - 1) -5 = [])
```

2. Sia `gen: (int -> 'a) -> int -> 'a list` la funzione tale che

   `gen f n = f 1::f 2::...::f n::[]`

   Se $n \leq 0$, allora viene restituita la lista vuota. Esempi:

```
assert (gen (fun x -> x * x) 5 = 1 :: 4 :: 9 :: 16 :: 25 :: [])
assert (gen (fun x -> x - 1) 5 = 0 :: 1 :: 2 :: 3 :: 4 :: [])
assert (gen (fun x -> x - 1) 0 = [])
assert (gen (fun x -> x - 1) -5 = [])
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.

Nota bene: **non è ammesso** usare funzionalità avanzate di F# o funzioni predefinite di `List` eccetto `List.rev`.

```
let swap f =
    let rec aux =
        function
        | hd1 :: hd2 :: tl ->
            if f hd1 hd2 then
                hd2 :: aux (hd1 :: tl)
            else
                hd1 :: aux (hd2 :: tl)
        | l -> l

    aux

swap

assert (swap (<) (1 :: 2 :: 3 :: []) = 2 :: 3 :: 1 :: [])

assert (swap (<) (2 :: 3 :: 1 :: []) = 3 :: 2 :: 1 :: [])

assert (swap (<) (3 :: 2 :: 1 :: []) = 3 :: 2 :: 1 :: [])

assert (swap (>) (1 :: []) = 1 :: [])

assert (swap (>) (1 :: 3 :: 2 :: 4 :: 0 :: []) = 1 :: 2 :: 3 :: 0 :: 4 :: [])

let accSwap f =
    let rec loop acc =
        function
        | hd1 :: hd2 :: tl ->
            if f hd1 hd2 then
                loop (hd2 :: acc) (hd1 :: tl)
            else
                loop (hd1 :: acc) (hd2 :: tl)
        | hd :: [] -> loop (hd :: acc) []
        | [] -> List.rev acc

    loop []

assert (accSwap (<) (1 :: 2 :: 3 :: []) = 2 :: 3 :: 1 :: [])

assert (accSwap (>) (1 :: []) = 1 :: [])

assert (accSwap (>) (1 :: 3 :: 2 :: 4 :: 0 :: []) = 1 :: 2 :: 3 :: 0 :: 4 :: [])
```

febbraio 2025

2. Sia `unzip: ('a * 'b) list -> 'a list * 'b list` la funzione che divide, rispettando l'ordine degli elementi, una lista di coppie in due contenenti rispettivamente tutti gli elementi di sinistra e tutti quelli di destra contenuti nelle coppie della lista originaria.

   Esempio:

```
assert (unzip((1,"a")::(2,"b")::(3,"c")::[])=(1::2::3::[],"a"::"b"::"c"::[]))
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.

Nota bene: **non è ammesso** usare funzionalità avanzate di F# o funzioni predefinite di `List` eccetto `List.rev`.

```
let rec unzip =
    function
    | (left,right)::tl ->
        let llist,rlist = unzip tl
        left::llist,right::rlist
    | [] -> [],[]

assert (unzip ((1,"a")::(2,"b")::(3,"c")::[]) = (1::2::3::[],"a"::"b"::"c"::[]))

let accUnzip ls =
    let rec loop llist rlist =
        function
        | (left,right)::tl -> loop (left::llist) (right::rlist) tl
        | [] -> List.rev llist,List.rev rlist
    loop [] [] ls

assert (accUnzip ((1,"a")::(2,"b")::(3,"c")::[]) = (1::2::3::[],"a"::"b"::"c"::[]))
```

gennaio 2025

2. Sia `partition: ('a -> bool) -> ('a list -> 'a list * 'a list)` la funzione che partiziona, rispettando l'ordine degli elementi, una lista in due contentenenti rispettivamente tutti gli elementi che soddisfano o meno un dato predicato.

   Esempio:

```
assert (partition (fun x -> x % 2 = 0) (1::2::3::4::6::[]) =
  (2::4::6::[],1::3::[]))
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.

Nota bene: **non è ammesso** usare funzionalità avanzate di F# o funzioni predefinite di `List` eccetto `List.rev`.

```
let partition p =
    let rec aux =
        function
        | hd::tl ->
            let trues,falses = aux tl
            if p hd then hd::trues,falses else trues,hd::falses
        | [] -> [],[]
    aux

assert (partition (fun x -> x % 2 = 0) (1::2::3::4::6::[]) = (2::4::6::[],1::3::[]))

let accPartition p =
    let rec loop trues falses =
        function
        | hd::tl -> if p hd then loop (hd::trues) falses tl else loop trues (hd::falses) tl
        | [] -> List.rev trues,List.rev falses
    loop [] []

assert (accPartition (fun x -> x % 2 = 0) (1::2::3::4::6::[]) = (2::4::6::[],1::3::[]))
```

Settembre 2024

2. Sia `filterMap: ('a -> 'b option) -> 'a list -> 'b list` la funzione tale che `filterMap` f l restituisce la lista ottenuta nel seguente modo:

   - viene applicata f a ogni elemento x di l;
   - se $f x$=`None`, allora x viene scartato;
   - altrimenti, se $f x$=`Some` y, allora y viene inserito nella lista restituita come risultato.

   Esempi:

```
assert (filterMap (fun e -> if e = 0 then None else Some e)
  [ 1; 0; 0; 2; 0; 3; 0; 0 ] = [ 1; 2; 3 ])
assert (filterMap (fun e -> if e < 0.0 then None else Some(sqrt e))
  [ -2; 4; 9; -5; -7; 1 ] = [ 2; 3; 1 ])
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.
   Non è ammesso usare funzionalità avanzate di F# o funzioni predefinite di `List` eccetto `List.rev`.

```
let rec filterMap f =
    function
    | hd :: tl ->
        match f hd with
        | Some el -> el :: filterMap f tl
        | _ -> filterMap f tl
    | _ -> []

assert (filterMap (fun e -> if e = 0 then None else Some e) [ 1; 0; 0; 2; 0; 3; 0; 0 ] = [ 1; 2; 3 ])
assert (filterMap (fun e -> if e < 0.0 then None else Some(sqrt e)) [ -2; 4; 9; -5; -7; 1 ] = [ 2; 3; 1

let accFilterMap f =
    let rec loop acc =
        function
        | hd :: tl ->
            match f hd with
            | Some el -> loop (el :: acc) tl
            | _ -> loop acc tl
        | _ -> List.rev acc

    loop []

assert (accFilterMap (fun e -> if e = 0 then None else Some e) [ 1; 0; 0; 2; 0; 3; 0; 0 ] = [ 1; 2; 3 ])
assert (accFilterMap (fun e -> if e < 0.0 then None else Some(sqrt e)) [ -2; 4; 9; -5; -7; 1 ] = [ 2; 3;
```

Luglio 2024

2. Sia `merge: 'a list -> 'a list -> 'a list` la funzione che fonde assieme due liste ordinate in un'altra ordinata con i loro stessi elementi, eventualmente ripetuti, come nell'algoritmo di merge sort, assumendo che la variabile `'a` corrisponda a un tipo dove i valori possono essere confrontati con l'operatore predefinito `<`.

   Esempi:

```
assert (merge [ 1; 2; 5 ] [ 2; 4; 6; 8 ] = [ 1; 2; 2; 4; 5; 6; 8 ])
assert (merge [ 2; 4; 6; 8 ] [ 1; 2; 5 ] = [ 1; 2; 2; 4; 5; 6; 8 ])
assert (merge [] [ "ab"; "cde"; "fg" ] = [ "ab"; "cde"; "fg" ])
assert (merge [ "ab"; "cde"; "fg" ] [] = [ "ab"; "cde"; "fg" ])
```

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.
   Non è ammesso usare funzioni predefinite di `List` eccetto `List.rev` o altre funzionalità avanzate di F#.

```
  let rec merge l1 l2 =
      match l1, l2 with
      | h1 :: t1, h2 :: t2 -> if h1 < h2 then h1 :: merge t1 l2 else h2 :: merge l1 t2
      | [], _ -> l2
      | _ -> l1

  let accMerge l =
      let rec loop acc l1 l2 =
          match l1, l2 with
          | h1 :: t1, h2 :: t2 ->
              if h1 < h2 then
                  loop (h1 :: acc) t1 l2
              else
                  loop (h2 :: acc) l1 t2
          | [], _ -> List.rev acc @ l2
          | _ -> List.rev acc @ l1

      loop [] l

  assert (merge [ 1; 2; 5 ] [ 2; 4; 6; 8 ] = [ 1; 2; 2; 4; 5; 6; 8 ])
  assert (merge [ 2; 4; 6; 8 ] [ 1; 2; 5 ] = [ 1; 2; 2; 4; 5; 6; 8 ])
  assert (merge [] [ "ab"; "cde"; "fg" ] = [ "ab"; "cde"; "fg" ])
  assert (merge [ "ab"; "cde"; "fg" ] [] = [ "ab"; "cde"; "fg" ])

  assert (accMerge [ 1; 2; 5 ] [ 2; 4; 6; 8 ] = [ 1; 2; 2; 4; 5; 6; 8 ])
  assert (accMerge [ 2; 4; 6; 8 ] [ 1; 2; 5 ] = [ 1; 2; 2; 4; 5; 6; 8 ])
  assert (accMerge [] [ "ab"; "cde"; "fg" ] = [ "ab"; "cde"; "fg" ])
  assert (accMerge [ "ab"; "cde"; "fg" ] [] = [ "ab"; "cde"; "fg" ])
```

giugno 2024

```
let rec vectAdd l1 l2 =
    match l1, l2 with
    | h1 :: t1, h2 :: t2 -> h1 + h2 :: vectAdd t1 t2
    | [], [] -> []
    | _ -> failwith "Vectors have different size"

assert (vectAdd [ 1; 2; 3 ] [ 3; 2; 1 ] = [ 4; 4; 4 ])

assert (vectAdd [] [] = [])

assert
    try
        vectAdd [ 1 ] [ 1; 2 ] = []
    with Failure _ ->
        true

let accVectAdd l1 l2 =
    let rec loop acc =
        function
        | h1 :: t1, h2 :: t2 -> loop (h1 + h2 :: acc) (t1, t2)
        | [], [] -> List.rev acc
        | _ -> failwith "Vectors have different size"

    loop [] (l1, l2)

assert (accVectAdd [ 1; 2; 3 ] [ 3; 2; 1 ] = [ 4; 4; 4 ])

assert (accVectAdd [] [] = [])

assert
    try
        accVectAdd [ 1 ] [ 1; 2 ] = []
    with Failure _ ->
        true
```

2. Sia `vectAdd: int list -> int list -> int list` la funzione che calcola l'addizione di due vettori:
   `vectAdd [`x_1`; ... ;`x_k`] [`y_1`; ... ;`y_k`]` $= [x_1 + y_1;\ \ldots\ ; x_k + y_k]$, con $k \geq 0$.

   Se le liste non hanno la stessa lunghezza, allora `vectAdd` solleva un'eccezione.

   Suggerimento: per sollevare l'eccezione, usare `failwith "Vectors have different size"`

   (a) Definire la funzione in F# **senza uso** di parametri di accumulazione e di funzioni predefinite di `List`.

   (b) Definire la funzione in F# usando un parametro di accumulazione affinché la ricorsione sia di coda.
   Non è ammesso usare funzioni predefinite di `List` eccetto `List.rev` o altre funzionalità avanzate di F#.