Esercizio A

1. Definire la funzione `twice` che, presa una funzione f, la applica al valore 1 e al risultato così ottenuto applica di nuovo f.
2. Quale è il tipo di `twice`?
3. Applicare `twice` alla funzione anonima che incrementa di 1 valori interi.
4. Applicare `twice` alla funzione anonima che moltiplica per 10 valori interi.

Soluzione

1. `let twice = fun f -> f (f 1)`

   oppure

   `let twice f = f (f 1)`
2. `(int -> int) -> int`

```
twice (fun x->x+1) = (fun f -> f (f 1)) (fun x->x+1)
                   = (fun x->x+1) ((fun x->x+1) 1)
                   = (fun x->x+1) (1+1)
                   = (fun x->x+1) 2
                   = 2+1 = 3
```

```
twice (fun x->x*10) = (fun f -> f (f 1)) (fun x->x*10)
                    = (fun x->x*10) ((fun x->x*10) 1)
                    = (fun x->x*10) (1*10)
                    = (fun x->x*10) 10
                    = 10*10 = 100
```

Esercizio B

1. Definire la funzione `scalar : int -> int * int -> int * int` che presi n e (x,y) restituisce (n*x,n*y).
   Esempio:
   ```
   assert (scalar 3 (2, 3) = (6, 9))
   ```

   Usare questa funzione per definire la funzione `doubleVec : int * int -> int * int` che raddoppia il vettore in input.
2. Definire la funzione `addVect : int * int -> int * int -> int * int` che presi (x1,y1) e (x2,y2) restituisce (x1+x2,y1+y2).
   Esempio:
   ```
   assert (addVect (1, 2) (3, 4) = (4, 6))
   ```

   Usare questa funzione per definire le funzioni `moveRight : in -> int * int -> int * int` e `moveUp : int -> int * int -> int * int` che prendono in input un intero n e un vettore (x,y) e traslano quest'ultimo di n unità rispettivamente verso destra e verso l'alto.
3. Definire la funzione `scalarProd : int * int -> int * int -> int` che presi (x1,y1) e (x2,y2) restituisce x1*x2+y1*y2.
   Esempio:
   ```
   assert (scalarProd (1, 2) (3, 4) = 11)
   ```

   Usare questa funzione per definire le funzioni `sumVec : int * int -> int` e `diffVec : int * int -> int` che calcolano rispettivamente la somma e la differenza delle componenti di un vettore. Usare poi una di queste funzioni per definire la funzione `isDiagonal : int * int -> bool` che controlla se il vettore in input sta sulla bisettrice del primo e terzo quadrante.
4. Definire le funzioni ai punti precedenti nelle loro versioni uncurried.

Soluzioni

```
let scalar n (x,y) = (n*x,n*y)
```

oppure
```
let scalar n (x,y) = n*x,n*y
```

```
let doubleVec = scalar 2
```

che è equivalente a

```
let doubleVec v = scalar 2 v
```

```
let addVect (x1,y1) (x2,y2) = (x1+x2,y1+y2)
```

oppure
```
let addVect (x1,y1) (x2,y2) = x1+x2,y1+y2
```

che è equivalente a

```
let doubleVec v = scalar 2 v

let addVect (x1,y1) (x2,y2) = (x1+x2,y1+y2)
```

oppure

```
let addVect (x1,y1) (x2,y2) = x1+x2,y1+y2


let moveRight n = addVect (n, 0)
let moveUp n = addVect (0, n)
```

oppure

```
let genMove v n = addVect (scalar n v)

let moveRight = genMove (1, 0)
let moveUp = genMove (0, 1)


let scalarProd (x1,y1) (x2,y2) = x1*x2+y1*y2

let sumVec = scalarProd (1,1)
let diffVec = scalarProd (1,-1)

let scalarUnc (n, (x, y)) = n * x, n * y
let doubleVecUnc v = scalarUnc (2,v)
let addVectUnc ((x1, y1), (x2, y2)) = x1 + x2, y1 + y2
let moveRightUnc (n, v) = addVectUnc ((n,0),v)
let moveUpUnc (n,v) = addVectUnc ((0,n),v)
let scalarProdUnc ((x1, y1), (x2, y2)) = x1 * x2 + y1 * y2
let sumVecUnc v = scalarProdUnc ((1,1),v)
let diffVecUnc v = scalarProdUnc ((1,-1),v)
```

notare che non si riesce ad usare l'applicazione parziale

Esercizio C

Nota: per definire funzioni ricorsive bisogna aggiungre la keyword `rec`, ad esempio `let rec f = ...`.

1. Definire la funzione generica `genSum : (int -> int) -> int -> int` tale che `genSum f n` calcola `f 0 + f 1 + ... + f n`.

   Una funzione è una specializzazione di `genSum` se ottenuta chiamando `genSum` e passando un'opportuna funzione come primo argomento.

   Definire come specializzazioni di `genSum` le funzioni `sumSquare` e `sumCube` che calcolano la somma dei quadrati e cubi dei numeri naturali da 0 a n inclusi.

   ```
   assert (sumSquare 3 = 14)
   ```

```
assert (sumSquare 3 = 14)

assert (sumCube 3 = 36)
```

2. Definire la funzione generica `genProd : (int -> int) -> int -> int` tale che `genProd f n` calcola `f 0 * f 1 * ... * f n`.

   Definire come specializzazioni di `genProd` le funzioni `fact` e `twoRaisedTo` che calcolano il fattoriale di n e 2 elevato alla n.

   ```
   assert (fact 5 = 120)

   assert (twoRaisedTo 10 = 1024)
   ```

Soluzioni

```
let rec genSum f n =
  if n<0 then 0 else f n + genSum f (n-1)
```

oppure

```
let genSum f =
  let rec aux n = if n < 0 then 0 else f n + aux (n - 1)
  aux

let sumSquare = genSum (fun x->x*x)
let sumCube = genSum (fun x->x*x*x)

let rec genProd f n =
  if n<0 then 1 else f n * genProd f (n-1)
```

oppure

```
let genProd f =
  let rec aux n = if n < 0 then 1 else f n * aux (n - 1)
  aux

let fact = genProd (fun x-> if x=0 then 1 else x)
let twoRaisedTo = genProd (fun x -> if x=0 then 1 else 2)
```

Esercizio D (difficile)

Considera la seguente definizione di funzione

```
let mapCollect (f1,f2) g (x,y) = g (f1 x) (f2 y)
```

Definire le funzioni dell'esercizio B come specializzazioni di `mapCollect`.

Soluzioni

```
let pair = fun x -> fun y -> x,y
let mult n = fun x -> n*x
let add n = fun x -> n+x
let scalar n = mapCollect (mult n, mult n) pair
let addVect (x,y) = mapCollect (add x, add y) pair
let scalarProd (x,y) = mapCollect (mult x, mult y) add
```