Esercizio 1

Definire nel linguaggio funzionale *Scheme* la funzione **`reverse`**, che computa l'inverso di una lista **`A`** di atomi in ingresso. Ecco alcuni esempi:

A	**reverse**
`()`	`()`
`(a)`	`(a)`
`(a b c d)`	`(d c b a)`

Esercizio 1

Definire nel linguaggio funzionale *Scheme* la funzione **`reverse`**, che computa l'inverso di una lista **`A`** di atomi in ingresso. Ecco alcuni esempi:

A	**reverse**
`()`	`()`
`(a)`	`(a)`
`(a b c d)`	`(d c b a)`

```
(define (reverse A)
   (if (null? A) ()
       (append (reverse (cdr A)) (list (car A)))
    )
)
```

Esercizio 2

Definire nel linguaggio funzionale *Scheme* la funzione **`spec`**, avente in ingresso una lista **`L`**, che restituisce la lista speculare di **`L`**, come nei seguenti esempi:

`L`	**`spec L`**
`()`	`()`
`(A)`	`(A)`
`(A B)`	`(B A)`
`(A (B C))`	`((C B) A)`
`(A (B C) (D (E F)) G)`	`(G ((F E) D) (C B) A)`

Esercizio 2

Definire nel linguaggio funzionale *Scheme* la funzione **spec**, avente in ingresso una lista **L**, che restituisce la lista speculare di **L**, come nei seguenti esempi:

L	spec L
()	()
(A)	(A)
(A B)	(B A)
(A (B C))	((C B) A)
(A (B C) (D (E F)) G)	(G ((F E) D) (C B) A)

```
(define (spec L)
    (if (null? L) ()
        (append (spec (cdr L))(list (if (atom? (car L))
                                        (car L)
                                        (spec (car L))
                                    )
                                )
        )
     )
)
```

Esercizio 3

Definire nel linguaggio funzionale *Scheme* la funzione **`natoms`**, avente in ingresso una lista **`L`**, che computa il numero di elementi atomici (non liste) di **`L`**. Ecco alcuni esempi:

`L`	**`natoms`**
`()`	0
`(()()(A))`	1
`(A (B C))`	3
`(A (B C) (D ()))`	4

Esercizio 3

Definire nel linguaggio funzionale *Scheme* la funzione **`natoms`**, avente in ingresso una lista **`L`**, che computa il numero di elementi atomici (non liste) di **`L`**. Ecco alcuni esempi:

L	**natoms**
`()`	0
`(()()(A))`	1
`(A (B C))`	3
`(A (B C) (D ()))`	4

```
(define (natoms L)
   (if (null? L) 0
       (if (atom? (car L))
           (+ 1 (natoms (cdr L)))
           (+ (natoms (car L)) (natoms (cdr L)))
        )
    )
)
```

Esercizio 4

Definire nel linguaggio *Scheme* la funzione **`atomi`**, avente in ingresso una **`lista`**, che computa la lista degli elementi atomici (non liste) di lista. Ecco alcuni esempi:

`lista`	**`atomi`**
`(()())`	`()`
`(A (B C))`	`(A B C)`
`((A B) C (D (E F) G))`	`(A B C D E F G)`

Esercizio 4

Definire nel linguaggio *Scheme* la funzione **`atomi`**, avente in ingresso una **`lista`**, che computa la lista degli elementi atomici (non liste) di lista. Ecco alcuni esempi:

lista	**atomi**
`(()())`	`()`
`(A (B C))`	`(A B C)`
`((A B) C (D (E F) G))`	`(A B C D E F G)`

```
(define (atomi lista)
   (if (null? lista) ()
       (if (atom? (car lista))
           (cons (car lista) (atomi (cdr lista)))
           (append (atomi (car lista)) (atomi (cdr lista)))
        )
    )
)
```

Esercizio 5

Definire nel linguaggio funzionale *Scheme* la funzione **`occ`**, avente in ingresso un elemento **`x`** ed una **`lista`**, che restituisce il numero di occorrenze di **`x`** in **`lista`**:

lista	**x**	**occ x lista**
`()`	`a`	0
`(a)`	`a`	1
`(b a b)`	`b`	2
`(c (c d) a c)`	`c`	2
`(a (b c) (c b) (d (b c)) b (b c) (b c) a)`	`(b c)`	3
`(a () (b ()) () c)`	`()`	2

Esercizio 5

Definire nel linguaggio funzionale *Scheme* la funzione **`occ`**, avente in ingresso un elemento **`x`** ed una **`lista`**, che restituisce il numero di occorrenze di **`x`** in **`lista`**:

lista	x	occ x lista
( )	a	0
(a)	a	1
(b a b)	b	2
(c (c d) a c)	c	2
(a (b c) (c b) (d (b c)) b (b c) (b c) a)	(b c)	3
(a () (b ()) () c)	()	2

```
(define (occ x lista)
    (if (null? lista) 0
        (if (equal? x (car lista))
            (+ 1 (occ x (cdr lista)))
            (occ x (cdr lista))
        )
     )
)
```

Esercizio 6

Definire nel linguaggio funzionale *Scheme* la funzione **`select`**, avente in ingresso una funzione **`fun`** ed una sequenza **`lista`**. Si assume che la funzione **`fun`** sia unaria e computi un valore booleano. La funzione **`select`** restituisce la sotto-sequenza (eventualmente vuota) di **`lista`** comprendente gli elementi che rendono la funzione **`fun`** vera, come nei seguenti esempi:

`lista`	**`fun`**	**`(select fun lista)`**
`()`	`number?`	`()`
`(a 3 125 (x y))`	`number?`	`(3 125)`
`(a 3 () x (2 d 12) 125)`	`atom?`	`(a 3 x 125)`
`(c (c d) a c (2 3 4))`	`list?`	`((c d) (2 3 4))`
`(1 (2 (3 4 x) z) 34 () y)`	`list?`	`((2 (3 4 x) z) ())`
`(1 2 3 x y z)`	`list?`	`()`

Esercizio 6

Definire nel linguaggio funzionale *Scheme* la funzione **`select`**, avente in ingresso una funzione **`fun`** ed una sequenza **`lista`**. Si assume che la funzione **`fun`** sia unaria e computi un valore booleano. La funzione **`select`** restituisce la sotto-sequenza (eventualmente vuota) di **`lista`** comprendente gli elementi che rendono la funzione **`fun`** vera, come nei seguenti esempi:

lista	**fun**	**(select fun lista)**
`()`	`number?`	`()`
`(a 3 125 (x y))`	`number?`	`(3 125)`
`(a 3 () x (2 d 12) 125)`	`atom?`	`(a 3 x 125)`
`(c (c d) a c (2 3 4))`	`list?`	`((c d) (2 3 4))`
`(1 (2 (3 4 x) z) 34 () y)`	`list?`	`((2 (3 4 x) z) ())`
`(1 2 3 x y z)`	`list?`	`()`

```
(define (select fun lista)
    (if (null? lista) ()
        (if (fun (car lista))
            (cons (car lista) (select fun (cdr lista)))
            (select fun (cdr lista))
         )
    )
)
```

Esercizio 7

Definire nel linguaggio funzionale *Scheme* la funzione **`cancella`**, avente in ingresso un **`elemento`** ed una **`lista`**, che computa la lista risultante dalla cancellazione di tutte le istanze di **`elemento`** in **`lista`**, come nei seguenti esempi:

`elemento`	**`lista`**	**`(cancella elemento lista)`**
`3`	`()`	`()`
`3`	`(3)`	`()`
`3`	`(1 2 3)`	`(1 2)`
`3`	`(1 2 3 4 3 5 3)`	`(1 2 4 5)`
`(1 2)`	`((1 2) (3 4) 5)`	`((3 4) 5)`
`()`	`(1 2 3 () (4 5) () 6)`	`(1 2 3 (4 5) 6)`

Esercizio 7

Definire nel linguaggio funzionale *Scheme* la funzione **`cancella`**, avente in ingresso un **`elemento`** ed una **`lista`**, che computa la lista risultante dalla cancellazione di tutte le istanze di **`elemento`** in **`lista`**, come nei seguenti esempi:

`elemento`	**`lista`**	**`(cancella elemento lista)`**
`3`	`()`	`()`
`3`	`(3)`	`()`
`3`	`(1 2 3)`	`(1 2)`
`3`	`(1 2 3 4 3 5 3)`	`(1 2 4 5)`
`(1 2)`	`((1 2) (3 4) 5)`	`((3 4) 5)`
`()`	`(1 2 3 () (4 5) () 6)`	`(1 2 3 (4 5) 6)`

```
(define (cancella elemento lista)
   (if (null? lista) ()
       (if (equal? (car lista) elemento)
           (cancella elemento (cdr lista))
           (cons (car lista) (cancella elemento (cdr lista))))))
```

Esercizio 8

Definire nel linguaggio funzionale *Scheme* la funzione **`zip`**, avente in ingresso due liste, **`lista1`** e **`lista2`**, che computa la lista di coppie di elementi che sono nella stessa posizione nelle rispettive liste, come nei seguenti esempi:

`lista1`	**`lista2`**	**`(zip lista1 lista2)`**
`()`	`()`	`()`
`()`	`(1 2)`	`()`
`(1 2 3)`	`()`	`()`
`(a b c)`	`(1 2 3 4)`	`((a 1)(b 2)(c 3))`
`(() 1 (a b))`	`(() (3 4) 5 zeta)`	`((()())(1 (3 4))((a b) 5))`

Esercizio 8

Definire nel linguaggio funzionale *Scheme* la funzione **`zip`**, avente in ingresso due liste, **`lista1`** e **`lista2`**, che computa la lista di coppie di elementi che sono nella stessa posizione nelle rispettive liste, come nei seguenti esempi:

lista1	**lista2**	**(zip lista1 lista2)**
()	()	()
()	(1 2)	()
(1 2 3)	()	()
(a b c)	(1 2 3 4)	((a 1)(b 2)(c 3))
(() 1 (a b))	(() (3 4) 5 zeta)	((()())(1 (3 4))((a b) 5))

```
(define (zip lista1 lista2)
  (if (or (null? lista1)(null? lista2))
      ()
      (cons (list (car lista1) (car lista2))
            (zip (cdr lista1) (cdr lista2)))))
```

Esercizio 9

Definire nel linguaggio funzionale *Scheme* la funzione **`shrink`**, avente in ingresso una lista **`L`**, che computa la lista degli elementi di **`L`** che si trovano in posizione dispari, come nei seguenti esempi:

`L`	**`(shrink L)`**
`()`	`()`
`(a)`	`(a)`
`(a b)`	`(a)`
`(a b c)`	`(a c)`
`(a (1 2 3) (4 5 (6 7)) 8 ())`	`(a (4 5 (6 7))())`

Esercizio 9

Definire nel linguaggio funzionale *Scheme* la funzione **`shrink`**, avente in ingresso una lista **`L`**, che computa la lista degli elementi di **`L`** che si trovano in posizione dispari, come nei seguenti esempi:

`L`	**`(shrink L)`**
`()`	`()`
`(a)`	`(a)`
`(a b)`	`(a)`
`(a b c)`	`(a c)`
`(a (1 2 3) (4 5 (6 7)) 8 ())`	`(a (4 5 (6 7))())`

```
(define (shrink L)
  (if (null? L)()
      (if (null? (cdr L))L
          (cons(car L)(shrink (cddr L))))))
```

Esercizio 10

Definire nel linguaggio funzionale *Scheme* la funzione **`remdup`**, avente in ingresso una lista **`L`**, che computa la lista degli elementi di **`L`** senza duplicazioni, come nei seguenti esempi:

`L`	**`(remdup L)`**
`()`	`()`
`(a)`	`(a)`
`(a b)`	`(a b)`
`(a b a)`	`(b a)`
`(a b a b a c d a)`	`(b c d a)`
`((a b)(a b c)(a b) 3 4 ())`	`((a b c) (a b) 3 4 ())`
`((a b)(2 (a b)) c (a b) 10)`	`((2 (a b)) c (a b) 10)`

Esercizio 10

Definire nel linguaggio funzionale *Scheme* la funzione **`remdup`**, avente in ingresso una lista **`L`**, che computa la lista degli elementi di **`L`** senza duplicazioni, come nei seguenti esempi:

L	**(remdup L)**
`()`	`()`
`(a)`	`(a)`
`(a b)`	`(a b)`
`(a b a)`	`(b a)`
`(a b a b a c d a)`	`(b c d a)`
`((a b)(a b c)(a b) 3 4 ())`	`((a b c) (a b) 3 4 ())`
`((a b)(2 (a b)) c (a b) 10)`	`((2 (a b)) c (a b) 10)`

```
(define (remdup L)
  (if (null? L)()
      (if (member (car L)(cdr L))
          (remdup (cdr L))
          (cons (car L)(remdup (cdr L))))))
```

Esercizio 11

Definire nel linguaggio *Scheme* la funzione **`remove`**, avente in ingresso una funzione **`f`** ed una **`lista`**. Si assume che la funzione **`f`** sia unaria e computi un valore booleano. La funzione **`remove`** restituisce la sotto-lista (eventualmente vuota) di **`lista`** comprendente gli elementi che rendono la funzione **`f`** falsa, come nei seguenti esempi:

`lista`	**`f`**	**`(remove f lista)`**
`()`	`number?`	`()`
`(x 2 4 (y z))`	`number?`	`(x (y z))`
`(x 10 () y (1 x 20) 82)`	`atom?`	`(() (1 x 20))`
`(a (b c) 10 (1 2 x))`	`list?`	`(a 10)`
`(a (b (1 2 y) 3) 25 () z w)`	`list?`	`(a 25 z w)`
`(a b 20)`	`list?`	`(a b 20)`

Esercizio 11

Definire nel linguaggio *Scheme* la funzione **`remove`**, avente in ingresso una funzione **`f`** ed una **`lista`**. Si assume che la funzione **`f`** sia unaria e computi un valore booleano. La funzione **`remove`** restituisce la sotto-lista (eventualmente vuota) di **`lista`** comprendente gli elementi che rendono la funzione **`f`** falsa, come nei seguenti esempi:

lista	**f**	**(remove f lista)**
`()`	`number?`	`()`
`(x 2 4 (y z))`	`number?`	`(x (y z))`
`(x 10 () y (1 x 20) 82)`	`atom?`	`(() (1 x 20))`
`(a (b c) 10 (1 2 x))`	`list?`	`(a 10)`
`(a (b (1 2 y) 3) 25 () z w)`	`list?`	`(a 25 z w)`
`(a b 20)`	`list?`	`(a b 20)`

```
(define (remove f lista)
   (if (null? lista) ()
       (if (f (car lista))
           (remove f (cdr lista))
           (cons (car lista) (remove f (cdr lista)))
        )
   )
)
```

Esercizio 12

Definire nel linguaggio *Scheme* la funzione **`serie`**, avente in ingresso un intero **`n`** >= 0, che computa la lista dei numeri interi da 1 a **`n`**, come nei seguenti esempi:

`n`	**`(serie n)`**
0	`()`
1	`(1)`
3	`(1 2 3)`
7	`(1 2 3 4 5 6 7)`

Esercizio 12

Definire nel linguaggio *Scheme* la funzione **`serie`**, avente in ingresso un intero **`n`** >= 0, che computa la lista dei numeri interi da 1 a **`n`**, come nei seguenti esempi:

n	**(serie n)**
0	()
1	(1)
3	(1 2 3)
7	(1 2 3 4 5 6 7)

```
(define (serie n)
   (if (= n 0) ()
       (append (serie (- n 1)) (list n))
   )
)
```

Esercizio 13

Definire nel linguaggio *Scheme* la funzione **`minimo`**, avente in ingresso una lista (non vuota) **`interi`**, che computa il numero più piccolo in **`interi`**, come nei seguenti esempi:

`interi`	**`(minimo interi)`**
`(2)`	`2`
`(4 6 3)`	`3`
`(10 8 20 40)`	`8`

Esercizio 13

Definire nel linguaggio *Scheme* la funzione **`minimo`**, avente in ingresso una lista (non vuota) **`interi`**, che computa il numero più piccolo in **`interi`**, come nei seguenti esempi:

`interi`	**`(minimo interi)`**
`(2)`	`2`
`(4 6 3)`	`3`
`(10 8 20 40)`	`8`

```
(define (minimo interi)
   (if (null? (cdr interi))(car interi)
       (if (< (car interi) (minimo (cdr interi)))
           (car interi)
           (minimo (cdr interi))
       )
    )
)
```

Esercizio 14

Definire nel linguaggio *Scheme* la funzione **`apply`**, avente in ingresso una funzione unaria **`f`** ed una **`lista`**. Si assume che **`lista`** rappresenti una espressione *Scheme*. La funzione **`apply`** computa l'applicazione di **`f`** al valore della espressione rappresentata da **`lista`**, come nei seguenti esempi:

`f`	**`lista`**	**`(apply f lista)`**
`number?`	`(length '(1 2 3))`	`#t`
`list?`	`(+ 1 2 3)`	`#f`
`fact`	`(+ 1 2 3)`	`720`
`quadrato`	`(length '(1 2 3))`	`9`
`reverse`	`(subst 'x 100 '(x 100 y))`	`(y x x)`

Esercizio 14

Definire nel linguaggio *Scheme* la funzione **`apply`**, avente in ingresso una funzione unaria **`f`** ed una **`lista`**. Si assume che **`lista`** rappresenti una espressione *Scheme*. La funzione **`apply`** computa l'applicazione di **`f`** al valore della espressione rappresentata da **`lista`**, come nei seguenti esempi:

`f`	**`lista`**	**`(apply f lista)`**
`number?`	`(length '(1 2 3))`	`#t`
`list?`	`(+ 1 2 3)`	`#f`
`fact`	`(+ 1 2 3)`	`720`
`quadrato`	`(length '(1 2 3))`	`9`
`reverse`	`(subst 'x 100 '(x 100 y))`	`(y x x)`

```
(define (apply f lista)
  (f (eval lista)))
```

Esercizio 15

Definire nel linguaggio *Scheme* la funzione booleana **`prefix?`**, avente in ingresso due liste, **`L1`** ed **`L2`**, la quale risulta vera se e solo se **`L1`** è un prefisso di **`L2`**, come nei seguenti esempi:

L1	**L2**	**(prefix? L1 L2)**
`()`	`()`	`#t`
`()`	`(a b c)`	`#t`
`(a)`	`()`	`#f`
`(a)`	`(a b c)`	`#t`
`(a b)`	`(a b c)`	`#t`
`(a b)`	`(b a c)`	`#f`
`((a b) c)`	`((a b) c (d e))`	`#t`
`((a b))`	`(a b c)`	`#f`
`(a c)`	`(a b c d)`	`#f`

Esercizio 15

Definire nel linguaggio *Scheme* la funzione booleana **`prefix?`**, avente in ingresso due liste, **`L1`** ed **`L2`**, la quale risulta vera se e solo se **`L1`** è un prefisso di **`L2`**, come nei seguenti esempi:

L1	L2	(prefix? L1 L2)
()	()	#t
()	(a b c)	#t
(a)	()	#f
(a)	(a b c)	#t
(a b)	(a b c)	#t
(a b)	(b a c)	#f
((a b) c)	((a b) c (d e))	#t
((a b))	(a b c)	#f
(a c)	(a b c d)	#f

```
(define (prefix? L1 L2)
  (if (null? L1) #t
      (if (null? L2) #f
          (if (equal? (car L1) (car L2))
              (prefix?(cdr L1) (cdr L2))
              #f))))
```

Esercizio 16

Dopo aver definito in *Scheme* la funzione booleana **`appartiene`**, che stabilisce se **`x`** appartiene alla lista **`set`**, come nei seguenti esempi,

x	**set**	**(appartiene x set)**
1	(1 2 3)	#t
4	(a b c)	#f
(a)	(a b c)	#f
(b)	(a (b) c)	#t
()	()	#f
()	(1 2 ())	#t

definire la funzione **`intersezione`**, avente in ingresso due liste (senza duplicati), **`set1`** e **`set2`**, che computa l'intersezione insiemistica **`set1`** $\cap$ **`set2`**.

Esercizio 16

x	set	(appartiene x set)
1	(1 2 3)	#t
4	(a b c)	#f
(a)	(a b c)	#f
(b)	(a (b) c)	#t
()	()	#f
()	(1 2 ())	#t

```
(define (appartiene x set)
  (if (null? set)
      #f
      (if (equal? x (car set))
          #t
         (appartiene x (cdr set)))))
```

```
(define (intersezione set1 set2)
  (if (null? set1)
      ()
      (if (appartiene (car set1) set2)
          (cons (car set1) (intersezione (cdr set1) set2))
          (intersezione (cdr set1) set2))))
```

Esercizio 17

Dopo aver definito in *Scheme* la funzione booleana **`manca`**, che stabilisce se **`x`** non è incluso nella lista **`L`**, come nei seguenti esempi,

x	**L**	**(manca x L)**
1	(1 2 3)	#f
4	(a b c)	#t
(a)	(a b c)	#t
(b)	(a (b) c)	#f
()	()	#t
()	(1 2 ())	#f

definire la funzione **`unione`**, avente in ingresso due liste (senza duplicati), **`L1`** e **`L2`**, che computa l'unione insiemistica (quindi, senza duplicati) **`L1`** $\cup$ **`L2`**.

Esercizio 17

x	L	(manca x L)
1	(1 2 3)	#f
4	(a b c)	#t
(a)	(a b c)	#t
(b)	(a (b) c)	#f
()	()	#t
()	(1 2 ())	#f

```
(define (manca x L)
  (if (null? L)
      #t
      (if (equal? x (car L))
          #f
          (manca x (cdr L)))))

(define (unione L1 L2)
  (if (null? L1)
      L2
      (if (manca (car L1) L2)
          (cons (car L1) (unione (cdr L1) L2))
          (unione (cdr L1) L2))))
```

Esercizio 18

Definire nel linguaggio *Scheme* la funzione **`take`** che, ricevendo un intero **`n`** ≥ 0 ed una **`lista`**, restituisce i primi **`n`** elementi di **`lista`**. Nel caso in cui **`n`** sia maggiore della lunghezza di **`lista`**, **`take`** restituisce **`lista`**. Ecco alcuni esempi:

`n`	**`lista`**	**`(take n lista)`**
3	`(1 2 3 4 5)`	`(1 2 3)`
3	`(a b)`	`(a b)`
0	`(a b c)`	`()`
2	`(a (b) c ())`	`(a (b))`
0	`()`	`()`

Esercizio 18

Definire nel linguaggio *Scheme* la funzione **`take`** che, ricevendo un intero **`n`** ≥ 0 ed una **`lista`**, restituisce i primi **`n`** elementi di **`lista`**. Nel caso in cui **`n`** sia maggiore della lunghezza di **`lista`**, **`take`** restituisce **`lista`**. Ecco alcuni esempi:

n	**lista**	**(take n lista)**
3	(1 2 3 4 5)	(1 2 3)
3	(a b)	(a b)
0	(a b c)	()
2	(a (b) c ())	(a (b))
0	()	()

```
(define (take n lista)
  (if (or (equal? n 0)(null? lista))
      ()
      (cons (car lista)(take (- n 1) (cdr lista)))))
```

Esercizio 19

Definire nel linguaggio *Scheme* la funzione **`domino`** che, ricevendo una **`lista`** di coppie di numeri, stabilisce se tali coppie sono disposte come i tasselli nel gioco del domino (uguaglianza del secondo numero di ogni coppia con il primo numero della successiva coppia). Ecco alcuni esempi:

`lista`	**`(domino lista)`**
`()`	`#t`
`((1 2))`	`#t`
`((1 2)(2 6))`	`#t`
`((1 2)(2 6)(6 34)(34 0))`	`#t`
`((1 3)(4 5)(5 7))`	`#f`

Esercizio 19

Definire nel linguaggio *Scheme* la funzione **`domino`** che, ricevendo una **`lista`** di coppie di numeri, stabilisce se tali coppie sono disposte come i tasselli nel gioco del domino (uguaglianza del secondo numero di ogni coppia con il primo numero della successiva coppia). Ecco alcuni esempi:

`lista`	**`(domino lista)`**
`()`	`#t`
`((1 2))`	`#t`
`((1 2)(2 6))`	`#t`
`((1 2)(2 6)(6 34)(34 0))`	`#t`
`((1 3)(4 5)(5 7))`	`#f`

```
(define (domino lista)
  (if (or (null? lista) (null? (cdr lista))) #t
      (if (not (equal? (cadar lista) (caadr lista))) #f
          (domino (cdr lista)))))
```

Esercizio 20

Nel linguaggio *Scheme*, si assume di avere a disposizione la funzione booleana binaria **`prefix?`** (di cui non è richiesta la specifica), che stabilisce se una lista (primo argomento) è un prefisso di un'altra lista (secondo argomento). Si chiede di definire la funzione booleana binaria **`sublist?`** che, ricevendo le liste **`S`** ed **`L`**, stabilisce se **`S`** è una sottolista di **`L`**. Ecco alcuni esempi:

`S`	**`L`**	**`(sublist? S L)`**
`()`	`(1 2)`	`#t`
`(1 2)`	`()`	`#f`
`(1 2)`	`(1 3 2)`	`#f`
`(1 2)`	`(1 2)`	`#t`
`(1 2)`	`(2 3 1 2 3 4 5)`	`#t`
`((a b) c)`	`(() (a b) c () d)`	`#t`
`(() a)`	`(1 b () a c)`	`#t`
`(())`	`()`	`#f`

Esercizio 20

Nel linguaggio *Scheme*, si assume di avere a disposizione la funzione booleana binaria **`prefix?`** (di cui non è richiesta la specifica), che stabilisce se una lista (primo argomento) è un prefisso di un'altra lista (secondo argomento). Si chiede di definire la funzione booleana binaria **`sublist?`** che, ricevendo le liste **`S`** ed **`L`**, stabilisce se **`S`** è una sottolista di **`L`**. Ecco alcuni esempi:

`S`	**`L`**	**`(sublist? S L)`**
`()`	`(1 2)`	`#t`
`(1 2)`	`()`	`#f`
`(1 2)`	`(1 3 2)`	`#f`
`(1 2)`	`(1 2)`	`#t`
`(1 2)`	`(2 3 1 2 3 4 5)`	`#t`
`((a b) c)`	`(() (a b) c () d)`	`#t`
`(() a)`	`(1 b () a c)`	`#t`
`(())`	`()`	`#f`

```
(define (sublist? S L)
      (if (> (length S)(length L)) #f
          (if (prefix? S  L) #t
              (sublist? S (cdr L)))))
```

Esercizio 21

Specificare nel linguaggio *Scheme* la funzione binaria **`indexing`** che, ricevendo in ingresso un array associativo **`A`** (rappresentato da una lista di coppie chiave-valore) ed una chiave di accesso **`key`**, computa il valore corrispondente alla chiave **`key`** nell'array associativo **`A`**. Ogni chiave di accesso è una stringa di caratteri. Nel caso di chiave inesistente, la funzione restituisce l'atomo **`error`**. Ecco alcuni esempi:

A	**key**	**(indexing A key)**
`((alfa 3)(beta 4)(gamma 5))`	`beta`	`4`
`((x (1 2))(y (3 4))(z (5 6)) (w ()))`	`y`	`(3 4)`
`((alfa 3)(beta 4)(gamma 5))`	`delta`	`error`

Esercizio 21

Specificare nel linguaggio *Scheme* la funzione binaria **`indexing`** che, ricevendo in ingresso un array associativo **`A`** (rappresentato da una lista di coppie chiave-valore) ed una chiave di accesso **`key`**, computa il valore corrispondente alla chiave **`key`** nell'array associativo **`A`**. Ogni chiave di accesso è una stringa di caratteri. Nel caso di chiave inesistente, la funzione restituisce l'atomo **`error`**. Ecco alcuni esempi:

A	**key**	**(indexing A key)**
((alfa 3)(beta 4)(gamma 5))	beta	4
((x (1 2))(y (3 4))(z (5 6)) (w ()))	y	(3 4)
((alfa 3)(beta 4)(gamma 5))	delta	error

```
(define (indexing A key)
  (if (null? A)
      'errore
      (if (equal? key (car (car A)))
          (car (cdr (car A)))
          (indexing (cdr A) key))))
```

Esercizio 22

Specificare nel linguaggio *Scheme* la funzione unaria **`duplica`** che, ricevendo in ingresso una **`lista`**, computa una nuova lista, in cui ogni elemento di **`lista`** viene duplicato in una coppia di elementi uguali. Ecco alcuni esempi:

`lista`	**`(duplica lista)`**
`()`	`()`
`(a)`	`((a a))`
`(a b c)`	`((a a)(b b)(c c))`
`(1 () (2 3) (4))`	`((1 1)(())((2 3)(2 3))((4)(4)))`

Esercizio 22

Specificare nel linguaggio *Scheme* la funzione unaria **`duplica`** che, ricevendo in ingresso una **`lista`**, computa una nuova lista, in cui ogni elemento di **`lista`** viene duplicato in una coppia di elementi uguali. Ecco alcuni esempi:

`lista`	**`(duplica lista)`**
`()`	`()`
`(a)`	`((a a))`
`(a b c)`	`((a a)(b b)(c c))`
`(1 () (2 3) (4))`	`((1 1)(())((2 3)(2 3))((4)(4)))`

```
(define (duplica lista)
  (if (null? lista)
      ()
      (cons (list (car lista) (car lista)) (duplica (cdr lista)))))
```

Esercizio 23

Dopo aver specificato in *Scheme* la funzione di Fibonacci **`(fib n)`**, specificare la funzione **`(listafib n)`** che, ricevendo in ingresso un intero **`n`** $\geq$ **`0`**, computa la lista dei numeri di Fibonacci, da **`0`** ad **`n`**, come nei seguenti esempi:

`n`	**`(listafib n)`**
0	`(0)`
1	`(0 1)`
2	`(0 1 1)`
3	`(0 1 1 2)`
4	`(0 1 1 2 3)`
9	`(0 1 1 2 3 5 8 13 21 34)`

(Si ricorda che, per **`n >= 2`**, il numero di Fibonacci relativo ad **`n`** è la somma dei numeri di Fibonacci relativi ad **`n-1`** ed **`n-2`**.)

Esercizio 23

Dopo aver specificato in *Scheme* la funzione di Fibonacci **`(fib n)`**, specificare la funzione **`(listafib n)`** che, ricevendo in ingresso un intero **`n`** ≥ **`0`**, computa la lista dei numeri di Fibonacci, da **`0`** ad **`n`**, come nei seguenti esempi:

`n`	**`(listafib n)`**
0	`(0)`
1	`(0 1)`
2	`(0 1 1)`
3	`(0 1 1 2)`
4	`(0 1 1 2 3)`
9	`(0 1 1 2 3 5 8 13 21 34)`

(Si ricorda che, per **`n >= 2`**, il numero di Fibonacci relativo ad **`n`** è la somma dei numeri di Fibonacci relativi ad **`n-1`** ed **`n-2`**.)

```
(define (fib n)
  (if (= n 0) 0
      (if (= n 1) 1
          (+ (fib (- n 1))
             (fib (- n 2))))))


(define (listafib n)
  (if (= n 0) '(0)
      (append (listafib (- n 1))
              (list (fib n)))))
```

Esercizio 24

Definire nel linguaggio Scheme la funzione **`iniziale`**, la quale, ricevendo in ingresso una **`lista`**, computa la lista ottenuta eliminando l'ultimo elemento di **`lista`**, come nei seguenti esempi:

`lista`	**`(iniziale lista)`**
`(a b c)`	`(a b)`
`(() 1 (2 3 4) (3 a))`	`(() 1 (2 3 4))`
`(())`	`()`

Esercizio 24

Definire nel linguaggio Scheme la funzione **`iniziale`**, la quale, ricevendo in ingresso una **`lista`**, computa la lista ottenuta eliminando l'ultimo elemento di **`lista`**, come nei seguenti esempi:

`lista`	**`(iniziale lista)`**
`(a b c)`	`(a b)`
`(() 1 (2 3 4) (3 a))`	`(() 1 (2 3 4))`
`(())`	`()`

```
(define (iniziale lista)
  (if (null? (cdr lista))
      ()
      (cons (car lista)(iniziale (cdr lista)))))
```

Esercizio 25

Definire nel linguaggio *Scheme* la funzione **`inserisci`**, la quale, ricevendo in ingresso un intero **`n`** ed una **`lista`** ordinata (in modo ascendente) di numeri, genera la lista ordinata ottenuta inserendo **`n`** in **`lista`**, come nei seguenti esempi:

`n`	**`lista`**	**`(inserisci n lista)`**
`4`	`(2 3 6 7 9)`	`(2 3 4 6 7 9)`
`5`	`(2 3 5 6 7)`	`(2 3 5 5 6 7)`
`6`	`()`	`(6)`
`8`	`(2 3 5 6 7)`	`(2 3 5 6 7 8)`
`1`	`(2 3 5 6 7)`	`(1 2 3 5 6 7)`

Quindi, definire la funzione **`ordina`** che, ricevendo una **`lista`** (anche vuota) di interi, genera la lista ordinata.

Esercizio 25

Definire nel linguaggio *Scheme* la funzione **`inserisci`**, la quale, ricevendo in ingresso un intero **`n`** ed una **`lista`** ordinata (in modo ascendente) di numeri, genera la lista ordinata ottenuta inserendo **`n`** in **`lista`**, come nei seguenti esempi:

`n`	**`lista`**	**`(inserisci n lista)`**
4	(2 3 6 7 9)	(2 3 4 6 7 9)
5	(2 3 5 6 7)	(2 3 5 5 6 7)
6	()	(6)
8	(2 3 5 6 7)	(2 3 5 6 7 8)
1	(2 3 5 6 7)	(1 2 3 5 6 7)

Quindi, definire la funzione **`ordina`** che, ricevendo una **`lista`** (anche vuota) di interi, genera la lista ordinata.

```
(define (inserisci n lista)
  (if (null? lista)
      (list n)
      (if (<= n (car lista))
          (cons n lista)
          (cons (car lista) (inserisci n (cdr lista))))))

(define (ordina lista)
  (if (null? lista) ()
      (inserisci (car lista) (ordina (cdr lista)))))
```

Esercizio 26

Dopo aver specificato la funzione fattoriale di un intero n ≥ 0 (fact n), specificare nel linguaggio *Scheme* la funzione **`(fattoriali n)`** che, ricevendo in ingresso un intero **n** ≥ 0, genera la lista dei fattoriali degli interi compresi tra 0 ed **n**, come nei seguenti esempi:

n	**(fattoriali n)**
0	(1)
1	(1 1)
2	(1 1 2)
3	(1 1 2 6)
5	(1 1 2 6 24 120)

Esercizio 26

Dopo aver specificato la funzione fattoriale di un intero n ≥ 0 (fact n), specificare nel linguaggio *Scheme* la funzione **`(fattoriali n)`** che, ricevendo in ingresso un intero **n** ≥ 0, genera la lista dei fattoriali degli interi compresi tra 0 ed **n**, come nei seguenti esempi:

n	**(fattoriali n)**
0	(1)
1	(1 1)
2	(1 1 2)
3	(1 1 2 6)
5	(1 1 2 6 24 120)

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))

(define (fattoriali n)
  (if (= n 0) '(1)
      (append (fattoriali (- n 1)) (list (fact n)))))
```

Esercizio 27

Definire nel linguaggio *Scheme* la funzione **`(revatoms L)`** che, ricevendo in ingresso una generica lista **`L`**, computa la lista speculare degli atomi di **`L`**, come nei seguenti esempi:

L	**(revatoms L)**
`()`	`()`
`(a b)`	`(b a)`
`(x () y (1 2 (c d)) z)`	`(z y x)`
`((a b c))`	`()`

Esercizio 27

Definire nel linguaggio *Scheme* la funzione **`(revatoms L)`** che, ricevendo in ingresso una generica lista **`L`**, computa la lista speculare degli atomi di **`L`**, come nei seguenti esempi:

L	**(revatoms L)**
`()`	`()`
`(a b)`	`(b a)`
`(x () y (1 2 (c d)) z)`	`(z y x)`
`((a b c))`	`()`

```
(define (revatoms L)
  (if (null? L) ()
      (if (atom? (car L))
          (append (revatoms (cdr L)) (list (car L)))
          (revatoms (cdr L)))))
```

Esercizio 28

Definire nel linguaggio *Scheme* la funzione **`media`** che, ricevendo in ingresso una **`lista`** (non vuota) di numeri, ne computa la media.

Esercizio 28

Definire nel linguaggio *Scheme* la funzione **`media`** che, ricevendo in ingresso una **`lista`** (non vuota) di numeri, ne computa la media.

```
(define (sum lista)
  (if (null? lista) 0
      (+ (car lista) (sum (cdr lista)))))

(define (media lista)
  (/ (sum lista) (length lista)))
```

Esercizio 29

Definire nel linguaggio *Scheme* la funzione **`itera`**, la quale, ricevendo in ingresso una funzione unaria **`f`**, un valore **`x`** ed un numero naturale **`n`**, computa la lista di **`n+1`** elementi, ottenuta partendo da **`x`** ed applicando **`f`** ripetutamente al valore precedente, come nei seguenti esempi:

f	**x**	**n**	**(itera f x n)**
`doppio`	`2`	`0`	`(2)`
`doppio`	`2`	`1`	`(4 2)`
`doppio`	`2`	`5`	`(64 32 16 8 4 2)`
`cdr`	`(1 2 3 4 5)`	`2`	`((3 4 5) (2 3 4 5) (1 2 3 4 5))`
`not`	`#t`	`3`	`(#f #t #f #t)`

Esercizio 29

Definire nel linguaggio *Scheme* la funzione **`itera`**, la quale, ricevendo in ingresso una funzione unaria **`f`**, un valore **`x`** ed un numero naturale **`n`**, computa la lista di **`n+1`** elementi, ottenuta partendo da **`x`** ed applicando **`f`** ripetutamente al valore precedente, come nei seguenti esempi:

f	x	n	(itera f x n)
doppio	2	0	(2)
doppio	2	1	(4 2)
doppio	2	5	(64 32 16 8 4 2)
cdr	(1 2 3 4 5)	2	((3 4 5) (2 3 4 5) (1 2 3 4 5))
not	#t	3	(#f #t #f #t)

```
(define (itera f x n)
    (if (= n 0) (list x)
        (append (itera f (f x) (- n 1)) (list x))))
```

Esercizio 30

Definire nel linguaggio *Scheme* la funzione **`valori`**, la quale, ricevendo in ingresso un numero naturale **`n`** ed una funzione unaria **`f`** di numeri naturali, computa la lista dei valori **`f`**`(0)`, **`f`**`(1)`, ..., **`f`**`(n)`**.**

Esercizio 30

Definire nel linguaggio *Scheme* la funzione **`valori`**, la quale, ricevendo in ingresso un numero naturale **`n`** ed una funzione unaria **`f`** di numeri naturali, computa la lista dei valori **`f`**`(0)`, **`f`**`(1)`, ..., **`f`**`(n)`.

```
(define (valori n f)
  (if (= n 0) (list (f 0))
      (append (valori (- n 1) f) (list (f n)))))
```

Esercizio 31

Codificare nel linguaggio *Scheme* la funzione booleana **`crescente`**, la quale, ricevendo in ingresso una lista di coppie di numeri, stabilisce se la somma di ogni coppia di numeri sia minore o uguale alla somma della coppia di numeri successiva, come nei seguenti esempi:

`lista`	**`(crescente lista)`**
`()`	`true`
`((2 3))`	`true`
`((3 2)(1 4)(3 3)(4 5))`	`true`
`((2 3)(1 4)(1 2)(4 5))`	`false`

Esercizio 31

Codificare nel linguaggio *Scheme* la funzione booleana **`crescente`**, la quale, ricevendo in ingresso una lista di coppie di numeri, stabilisce se la somma di ogni coppia di numeri sia minore o uguale alla somma della coppia di numeri successiva, come nei seguenti esempi:

lista	**(crescente lista)**
`()`	`true`
`((2 3))`	`true`
`((3 2)(1 4)(3 3)(4 5))`	`true`
`((2 3)(1 4)(1 2)(4 5))`	`false`

```
(define (crescente lista)
  (if (or (null? lista)(null? (cdr lista))) #t
      (if (> (+ (caar lista)(cadar lista))
             (+ (caadr lista)(cadadr lista))) #f
          (crescente (cdr lista)))))
```

Esercizio 32

Codificare nel linguaggio *Scheme* la funzione **`somma2`** che, ricevendo in ingresso una lista **`numeri`**, genera la lista delle somme delle coppie di numeri consecutivi (con in coda l'eventuale numero spaiato), come nei seguenti esempi:

`numeri`	**`(somma2 numeri)`**
`()`	`()`
`(2)`	`(2)`
`(2 3)`	`(5)`
`(2 3 4 5)`	`(5 9)`
`(2 3 4 5 1)`	`(5 9 1)`
`(2 3 4 5 1 6 7)`	`(5 9 7 7)`

Esercizio 32

Codificare nel linguaggio *Scheme* la funzione **`somma2`** che, ricevendo in ingresso una lista **`numeri`**, genera la lista delle somme delle coppie di numeri consecutivi (con in coda l'eventuale numero spaiato), come nei seguenti esempi:

numeri	**(somma2 numeri)**
()	()
(2)	(2)
(2 3)	(5)
(2 3 4 5)	(5 9)
(2 3 4 5 1)	(5 9 1)
(2 3 4 5 1 6 7)	(5 9 7 7)

```
(define (somma2 numeri)
  (if (or (null? numeri)
          (null? (cdr numeri)))
      numeri
      (cons (+ (car numeri)(cadr numeri))
            (somma2 (cddr numeri)))))
```

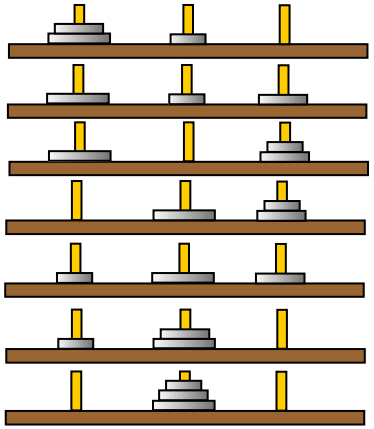
Esercizio 33

Definire nel linguaggio *Scheme* la funzione **`hanoi`**, avente in ingresso un intero **`n > 0`**, che restituisce la lista di mosse che spostano la torre di **`n`** dischi dal piolo sorgente al piolo di destinazione, in modo che vengano rispettate le seguenti regole:

- Solo il disco superiore di una torre può essere rimosso ad ogni spostamento;
- Il disco rimosso da una torre non può essere spostato su un disco più piccolo di un'altra torre.

Ogni mossa è rappresentata dalla coppia (*da a*).
Ecco un esempio (`n=3`):

```
(hanoi 3) = ((sinistra centro)
             (sinistra destra)
             (centro destra)
             (sinistra centro)
             (destra sinistra)
             (destra centro)
             (sinistra centro))
```



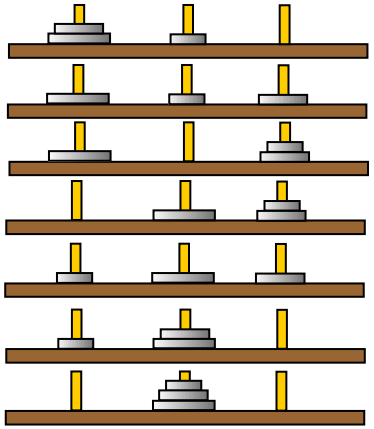
Esercizio 33

Definire nel linguaggio *Scheme* la funzione **`hanoi`**, avente in ingresso un intero **`n > 0`**, che restituisce la lista di mosse che spostano la torre di **`n`** dischi dal piolo sorgente al piolo di destinazione, in modo che vengano rispettate le seguenti regole:

- Solo il disco superiore di una torre può essere rimosso ad ogni spostamento;
- Il disco rimosso da una torre non può essere spostato su un disco più piccolo di un'altra torre.

Ogni mossa è rappresentata dalla coppia (*da a*).
Ecco un esempio (`n=3`):

```
(hanoi 3) = ((sinistra centro)
             (sinistra destra)
             (centro destra)
             (sinistra centro)
             (destra sinistra)
             (destra centro)
             (sinistra centro))
```



```
(define (hanoi n)
  (sposta n 'sinistra 'centro 'destra))

(define (sposta n sorgente destinazione appoggio)
  (if (= n 0) '()
      (append (append (sposta (- n 1) sorgente appoggio destinazione)
                      (list (list sorgente destinazione)))
              (sposta (- n 1) appoggio destinazione sorgente))))
```

Esercizio 34

Definire nel linguaggio *Scheme* la funzione **`maxval`**, avente in ingresso una funzione unaria **`f`** (che computa un valore intero) ed una lista (non vuota) di valori denominata `dominio`. La funzione **`maxval`** restituisce il massimo tra i valori computati da **`f`** quando viene applicata agli elementi di **`dominio`**.

Esercizio 34

Definire nel linguaggio *Scheme* la funzione **`maxval`**, avente in ingresso una funzione unaria **`f`** (che computa un valore intero) ed una lista (non vuota) di valori denominata `dominio`. La funzione **`maxval`** restituisce il massimo tra i valori computati da **`f`** quando viene applicata agli elementi di **`dominio`**.

```
(define (maxval f dominio)
  (if (null? (cdr dominio))
      (f (car dominio))
      (if (> (f (car dominio))(maxval f (cdr dominio)))
          (f (car dominio))
          (maxval f (cdr dominio)))))
```

Esercizio 35

Definire nel linguaggio *Scheme* la funzione **`clear`**, avente in ingresso una **`lista`**, che restituisce la lista in ingresso privata di tutti i suoi atomi (ad ogni livello). Ad esempio:

```
(clear '(x (y 10 (z w h)) (1) (a b)))
((()) () ())
```

Esercizio 35

Definire nel linguaggio *Scheme* la funzione **`clear`**, avente in ingresso una **`lista`**, che restituisce la lista in ingresso privata di tutti i suoi atomi (ad ogni livello). Ad esempio:

```
(clear '(x (y 10 (z w h)) (1) (a b)))
((()) () ())
```

```
(define (clear lista)
  (if (null? lista) lista
      (if (list? (car lista))
          (cons (clear (car lista)) (clear (cdr lista)))
          (clear (cdr lista)))))
```

Esercizio 36

Definire nel linguaggio *Scheme* la funzione `sumfun`, avente in ingresso un numero naturale `n` ed una una funzione unaria `f` (che computa un valore intero). La funzione `sumfun` computa `f(0) + f(1) + ... + f(n)`.

Esercizio 36

Definire nel linguaggio *Scheme* la funzione `sumfun`, avente in ingresso un numero naturale `n` ed una una funzione unaria `f` (che computa un valore intero). La funzione `sumfun` computa `f(0) + f(1) + ... + f(n)`.

```
(define (sumfun n f)
  (if (= n 0) (f 0)
      (+ (sumfun (- n 1) f) (f n))))
```

Esercizio 37

Definire nel linguaggio *Scheme* la funzione `ordinate`, avente in ingresso una `lista` (anche vuota) di coppie di numeri, che seleziona da `lista` le coppie strettamente ordinate, quelle cioè in cui il primo elemento è maggiore del secondo.

Esercizio 37

Definire nel linguaggio *Scheme* la funzione `ordinate`, avente in ingresso una `lista` (anche vuota) di coppie di numeri, che seleziona da `lista` le coppie strettamente ordinate, quelle cioè in cui il primo elemento è maggiore del secondo.

```
(define (ordinate lista)
  (if (null? lista) ()
      (if (> (caar lista)(cadar lista))
          (cons (car lista) (ordinate (cdr lista)))
          (ordinate (cdr lista)))))
```

Esercizio 38

Definire nel linguaggio *Scheme* la funzione `combina`, avente in ingresso una funzione binaria, una `lista1` e una `lista2`, che genera la lista i cui elementi sono il risultato dell'applicazione di `funzione` agli elementi di `lista1` e `lista2` nella stessa posizione. La lunghezza della lista generata coincide con la lunghezza della lista più corta. Ecco alcuni esempi:

funzione	**lista1**	**lista2**	**(combina funzione lista1 lista2)**
+	(1 2 3 4)	(10 20 30 40 50)	(11 22 33 44)
*	(1 2 3 4)	(10 20 30 40 50)	(10 40 90 160)
append	((1 2)(3 4 5))	((a b)(g d e)(z)))	((1 2 a b) (3 4 5 g d e))

Esercizio 38

Definire nel linguaggio *Scheme* la funzione `combina`, avente in ingresso una funzione binaria, una `lista1` e una `lista2`, che genera la lista i cui elementi sono il risultato dell'applicazione di `funzione` agli elementi di `lista1` e `lista2` nella stessa posizione. La lunghezza della lista generata coincide con la lunghezza della lista più corta. Ecco alcuni esempi:

funzione	**lista1**	**lista2**	**(combina funzione lista1 lista2)**
+	(1 2 3 4)	(10 20 30 40 50)	(11 22 33 44)
*	(1 2 3 4)	(10 20 30 40 50)	(10 40 90 160)
append	((1 2)(3 4 5))	((a b)(g d e)(z)))	((1 2 a b) (3 4 5 g d e))

```
(define (combina funzione lista1 lista2)
  (if (or (null? lista1) (null? lista2)) ()
      (cons (funzione (car lista1) (car lista2))
            (combina funzione (cdr lista1) (cdr lista2)))))
```

Esercizio 39

Specificare nel linguaggio *Scheme* la funzione `sommapotenze`, avente in ingresso una lista di interi, così definita:

$$sommapotenze([x_1, x_2, \ldots, x_n]) = \Sigma_{i=1}^{n}(x_i^i)$$

Nel caso limite di lista vuota, `sommapotenze` vale 0.

Esercizio 39

Specificare nel linguaggio *Scheme* la funzione `sommapotenze`, avente in ingresso una lista di interi, così definita:

$$sommapotenze([x_1, x_2, \ldots, x_n]) = \Sigma_{i=1}^{n}(x_i^i)$$

Nel caso limite di lista vuota, `sommapotenze` vale 0.

```
(define (init lista)
  (reverse (cdr (reverse lista))))

(define (last lista)
  (car (reverse lista)))

(define (power x n)
  (if (= n 0) 1
      (* x (power x (- n 1)))))

(define (sommapotenze numeri)
  (if (empty? numeri) 0
      (+ (sommapotenze (init numeri)) (power (last numeri) (length numeri)))))
```

oppure:

```
(define (sommapotenze numeri)
   (revsommapot (reverse numeri)))

(define (revsommapot numeri)
  (if (empty? numeri) 0
      (+ (power (car numeri) (length numeri)) (revsommapot (cdr numeri)))))
```

Esercizio 40

Specificare nel linguaggio *Scheme* la funzione `somme`, avente in ingresso una lista di interi, così definita:

$$somme([x_1, x_2, \ldots, x_n]) = [y_i, y_2, \ldots, y_n], where\ y_i = \Sigma_{j=1}^{i}(x_j), i \in [1..n]$$

Nel caso di lista vuota, `somme` restituisce la lista vuota.

Esercizio 40

Specificare nel linguaggio *Scheme* la funzione `somme`, avente in ingresso una lista di interi, così definita:

$$somme([x_1, x_2, \ldots, x_n]) = [y_i, y_2, \ldots, y_n], where\ y_i = \Sigma_{j=1}^{i}(x_j), i \in [1..n]$$

Nel caso di lista vuota, `somme` restituisce la lista vuota.

```
(define (somme numeri)
  (sommeaux 0 numeri))

(define (sommeaux n numeri)
  (if (null? numeri) '()
      (let ((testa (+ (car numeri) n)))
         (cons testa (sommeaux testa (cdr numeri))))))
```

Esercizio 41

Specificare nel linguaggio *Scheme* la funzione `funpair`, avente in ingresso una funzione binaria `f` e due liste, `list1` e `list2`, la quale restituisce la lista dei valori risultanti dall'applicazione di `f` agli elementi di `list1` e `list2` che si trovano nella stessa posizione nella rispettiva lista, come nel seguente esempio:

```
(funpair + '(1 2 3) '(4 5 6 7 8)) =  (5 7 9)
```

Esercizio 41

Specificare nel linguaggio *Scheme* la funzione `funpair`, avente in ingresso una funzione binaria `f` e due liste, `list1` e `list2`, la quale restituisce la lista dei valori risultanti dall'applicazione di `f` agli elementi di `list1` e `list2` che si trovano nella stessa posizione nella rispettiva lista, come nel seguente esempio:

```
(funpair + '(1 2 3) '(4 5 6 7 8)) =  (5 7 9)
```

```
(define (funpair f list1 list2)
  (if (or (null? list1) (null? list2))
      ()
      (cons (f (car list1) (car list2))
            (funpair f (cdr list1) (cdr list2)))))
```

Esercizio 42

Specificare nel linguaggio *Scheme* la funzione `medie`, avente in ingresso una lista di numeri, la quale restituisce una lista di lunghezza pari a quella in ingresso, in cui ogni elemento alla posizione *i*-esima è costituito dalla media dei numeri della lista di ingresso fino alla posizione *i*-esima, come nel seguente esempio:

```
(medie '(1 2 3 4 5)) = (1 1.5 2 2.5 3)
```

Nel caso di lista vuota, medie restituisce la lista vuota.

Esercizio 42

Specificare nel linguaggio *Scheme* la funzione `medie`, avente in ingresso una lista di numeri, la quale restituisce una lista di lunghezza pari a quella in ingresso, in cui ogni elemento alla posizione *i*-esima è costituito dalla media dei numeri della lista di ingresso fino alla posizione *i*-esima, come nel seguente esempio:

```
(medie '(1 2 3 4 5)) = (1 1.5 2 2.5 3)
```

Nel caso di lista vuota, medie restituisce la lista vuota.

```
(define (medie numeri)
  (reverse (medaux (reverse numeri))))

(define (medaux lista)
  (if (null? lista) '()
  (cons (media lista)(medaux (cdr lista)))))

(define (media lista)
  (/ (eval (cons '+ lista)) (length lista)))
```

Esercizio 43

Specificare nel linguaggio *Scheme* la funzione `cartesiano`, che computa il prodotto cartesiano di due liste in ingresso, come nel seguente esempio:

```
(cartesiano '(1 2) '(a b c)) = ((1 a)(1 b)(1 c)(2 a)(2 b)(2 c))
```

Esercizio 43

Specificare nel linguaggio *Scheme* la funzione `cartesiano`, che computa il prodotto cartesiano di due liste in ingresso, come nel seguente esempio:

```
(cartesiano '(1 2) '(a b c)) = ((1 a)(1 b)(1 c)(2 a)(2 b)(2 c))
```

```
(define (cartesiano lista1 lista2)
  (if (or (null? lista1) (null? lista2)) '()
      (append (combina (car lista1) lista2)
              (cartesiano (cdr lista1) lista2))))

(define (combina elemento lista)
  (if (null? lista) '()
  (cons (list elemento (car lista))
        (combina elemento (cdr lista)))))
```

Esercizio 44

Specificare nel linguaggio *Scheme* la funzione `revpairs`, la quale riceve in ingresso una lista di coppie (anche vuota) e computa la lista delle coppie invertite, come nel seguente esempio:

```
(revpairs '((1 2)(3 4)(alfa beta)) = ((2 1)(4 3)(beta alfa))
```

Esercizio 44

Specificare nel linguaggio *Scheme* la funzione `revpairs`, la quale riceve in ingresso una lista di coppie (anche vuota) e computa la lista delle coppie invertite, come nel seguente esempio:

```
(revpairs '((1 2)(3 4)(alfa beta)) = ((2 1)(4 3)(beta alfa))
```

```
(define (revpairs coppie)
  (if (null? coppie) '()
      (cons (list (cadar coppie) (caar coppie))
            (revpairs (cdr coppie)))))
```

Esercizio 45

Definire nel linguaggio *Scheme* la funzione `drop` (protocollo incluso) che, ricevendo un intero `n` ≥ 0 ed una `lista`, restituisce la lista in ingresso privata degli ultimi `n` elementi. Nel caso in cui `n` sia maggiore della lunghezza di lista, `drop` restituisce la lista vuota. Ecco alcuni esempi:

n	lista	(drop n lista)
2	(a b c d e)	(a b c)
2	(() (a b c) ())	(())
5	(a b c d e)	()
10	(a b c d e)	()
0	(a b c d e)	(a b c d e)

Esercizio 45

Definire nel linguaggio *Scheme* la funzione `drop` (protocollo incluso) che, ricevendo un intero `n` ≥ 0 ed una `lista`, restituisce la lista in ingresso privata degli ultimi `n` elementi. Nel caso in cui `n` sia maggiore della lunghezza di lista, `drop` restituisce la lista vuota. Ecco alcuni esempi:

n	lista	(drop n lista)
2	(a b c d e)	(a b c)
2	(() (a b c) ())	(())
5	(a b c d e)	()
10	(a b c d e)	()
0	(a b c d e)	(a b c d e)

```
(define (drop n lista)
  (take (- (length lista) n) lista))

(define (take n lista)
  (if (or (< n 1) (null? lista))
      ()
      (cons (car lista)(take (- n 1)(cdr lista)))))
```

Esercizio 46

Specificare nel linguaggio *Scheme* la funzione `singletons`, avente in ingresso una lista (anche vuota), la quale computa la lista di liste di un elemento, come nei seguenti esempi:

```
(singletons '()) = ().
(singletons '(5)) = ((5)).
(singletons '(1 2 3 4)) = ((1) (2) (3) (4)).
(singletons '(a (2 3 4) b () 10)) = ((a) ((2 3 4)) (b) (()) (10)).
```

Esercizio 46

Specificare nel linguaggio *Scheme* la funzione `singletons`, avente in ingresso una lista (anche vuota), la quale computa la lista di liste di un elemento, come nei seguenti esempi:

```
(singletons '()) = ().
(singletons '(5)) = ((5)).
(singletons '(1 2 3 4)) = ((1) (2) (3) (4)).
(singletons '(a (2 3 4) b () 10)) = ((a) ((2 3 4)) (b) (()) (10)).
```

```
(define (singletons lista)
  (if (null? lista) '()
      (cons (list (car lista)) (singletons (cdr lista)))))
```

Esercizio 47

Specificare nel linguaggio *Scheme* la funzione `swapairs`, avente in ingresso una lista (anche vuota) di un numero pari di elementi, la quale computa la lista con gli elementi scambiati a coppie, come nei seguenti esempi:

```
(swapairs '()) = ()
(swapairs '(2 5)) = (5 2)
(swapairs '(a b c d e f) = (b a d c f e)
(swapairs '(1 (2 3) (4 5 6) 7 () 8) = ((2 3) 1 7 (4 5 6) 8 ())
```

Esercizio 47

Specificare nel linguaggio *Scheme* la funzione `swapairs`, avente in ingresso una lista (anche vuota) di un numero pari di elementi, la quale computa la lista con gli elementi scambiati a coppie, come nei seguenti esempi:

```
(swapairs '()) = ()
(swapairs '(2 5)) = (5 2)
(swapairs '(a b c d e f) = (b a d c f e)
(swapairs '(1 (2 3) (4 5 6) 7 () 8) = ((2 3) 1 7 (4 5 6) 8 ())
```

```
(define (swapairs lista)
  (if (null? lista) '()
      (append (list (cadr lista) (car lista)) (swapairs (cddr lista)))))
```

Esercizio 48

Specificare nel linguaggio *Scheme* la funzione `sumpairs`, avente in ingresso una lista (anche vuota) di numeri, la quale computa la lista delle somme delle coppie, come nei seguenti esempi (se disaccoppiato, l'ultimo numero viene trascritto nel risultato):

```
(sumpairs '()) = ()
(sumpairs '(1)) = (1)
(sumpairs '(1 2 5) = (3 5)
(sumpairs '(1 2 3 4 8 12) = (3 7 20)
```

Esercizio 48

Specificare nel linguaggio *Scheme* la funzione `sumpairs`, avente in ingresso una lista (anche vuota) di numeri, la quale computa la lista delle somme delle coppie, come nei seguenti esempi (se disaccoppiato, l'ultimo numero viene trascritto nel risultato):

```
(sumpairs '()) = ()
(sumpairs '(1)) = (1)
(sumpairs '(1 2 5) = (3 5)
(sumpairs '(1 2 3 4 8 12) = (3 7 20)
```

```
(define (sumpairs lista)
  (if (null? lista) '()
      (if (null? (cdr lista)) (list (car lista))
       (cons (+ (car lista)(cadr lista)) (sumpairs (cddr lista))))))
```