Esercizio 1

Dopo aver specificato la funzione **`member`** in *Haskell* (appartenenza di un elemento ad una lista), definire la funzione booleana **`is_square`**, che stabilisce se un numero naturale in ingresso è il quadrato di un altro numero naturale.

Esercizio 1

Dopo aver specificato la funzione **`member`** in *Haskell* (appartenenza di un elemento ad una lista), definire la funzione booleana **`is_square`**, che stabilisce se un numero naturale in ingresso è il quadrato di un altro numero naturale.

```
member :: Ord a => a -> [a] -> Bool
member n ordlist
    | ordlist == []          = False
    | head ordlist < n       = member n (tail ordlist)
    | head ordlist == n      = True
    | otherwise              = False
```

```
quadrati :: [Integer]
quadrati = [ n^2 | n <- [ 0.. ] ]

is_square :: Integer -> Bool
is_square n =
  member n quadrati
```

Esercizio 2

Definire nel linguaggio *Haskell* la funzione **`reverse`**, avente in ingresso una lista, che restituisce la lista invertita, come nei seguenti esempi:

lista	**reverse**
`[]`	`[]`
`[ "alfa" ]`	`[ "alfa" ]`
`[ 'A', 'B' ]`	`['B', 'A' ]`
`[ 1, 2, 3 ]`	`[ 3, 2, 1 ]`

Esercizio 2

Definire nel linguaggio *Haskell* la funzione **`reverse`**, avente in ingresso una lista, che restituisce la lista invertita, come nei seguenti esempi:

lista	reverse
`[]`	`[]`
`[ "alfa" ]`	`[ "alfa" ]`
`[ 'A', 'B' ]`	`['B', 'A' ]`
`[ 1, 2, 3 ]`	`[ 3, 2, 1 ]`

```
reverse :: [a] -> [a]
reverse []           =    []
reverse (testa:coda) =    (reverse coda) ++ [testa]
```

Esercizio 3

Definire nel linguaggio *Haskell* una struttura tabellare

```
Prodotti(Codice, Nome, Prezzo)
```

in cui ogni riga rappresenta un prodotto. Quindi, codificare la funzione `selezione` che, ricevendo in ingresso due interi, `min` e `max`, computa la lista dei nomi dei prodotti il cui prezzo è compreso tra `min` e `max`.

Esercizio 3

Definire nel linguaggio *Haskell* una struttura tabellare

```
Prodotti(Codice, Nome, Prezzo)
```

in cui ogni riga rappresenta un prodotto. Quindi, codificare la funzione `selezione` che, ricevendo in ingresso due interi, `min` e `max`, computa la lista dei nomi dei prodotti il cui prezzo è compreso tra `min` e `max`.

```
type Codice = Integer
type Nome = String
type Prezzo = Integer

type Prodotti = [(Codice, Nome, Prezzo)]

selezione :: Prodotti -> Prezzo -> Prezzo -> [ Nome ]
selezione prodotti min max = [ n | (c, n, p) <- prodotti, p >= min, p <= max ]
```

Esercizio 4

Definire nel linguaggio *Haskell* le seguenti due strutture tabellari

```
Studente(Matricola, Anno, Corso)
Docente(Nome, Corso)
```

Quindi, codificare la funzione `docenti_dello_studente` che, ricevendo in ingresso le tabelle `stud` e `doc` e la matricola `mat` di uno studente, computa la lista dei nomi dei docenti relativi ai corsi di tale studente.

Esercizio 4

Definire nel linguaggio *Haskell* le seguenti due strutture tabellari

```
Studente(Matricola, Anno, Corso)
Docente(Nome, Corso)
```

Quindi, codificare la funzione `docenti_dello_studente` che, ricevendo in ingresso le tabelle `stud` e `doc` e la matricola `mat` di uno studente, computa la lista dei nomi dei docenti relativi ai corsi di tale studente.

```
type Matricola = Integer
type Anno = Integer
type Corso = String
type Nome = String

type Studenti = [(Matricola, Anno, Corso)]
type Docenti = [(Nome, Corso)]

docenti_dello_studente :: Studenti -> Docenti -> Matricola -> [ Nome ]
docenti_dello_studente stud doc mat =
   [ n | (m, a, c) <- stud, (n, c') <- doc, c==c',m==mat ]
```

Esercizio 5

Definire nel linguaggio *Haskell* una struttura tabellare

```
Esami (Matricola, Corso, Voto)
```

in cui ogni riga rappresenta rispettivamente uno studente, il corso, ed il relativo voto. Quindi, codificare in *Haskell* la funzione `esamiSostenuti` che, ricevendo in ingresso una tabella `esami` ed una certa `matricola`, computa la lista degli esami superati dal relativo studente (senza l'indicazione del voto).

Esercizio 5

Definire nel linguaggio *Haskell* una struttura tabellare

```
Esami (Matricola, Corso, Voto)
```

in cui ogni riga rappresenta rispettivamente uno studente, il corso, ed il relativo voto. Quindi, codificare in *Haskell* la funzione `esamiSostenuti` che, ricevendo in ingresso una tabella `esami` ed una certa `matricola`, computa la lista degli esami superati dal relativo studente (senza l'indicazione del voto).

```
type Matricola = Integer
type Corso = String
type Voto = Integer
type Esami = [(Matricola, Corso, Voto)]

esamiSostenuti :: Esami -> Matricola -> [ Corso ]
esamiSostenuti esami matricola = [ c | (m, c, v) <- esami, m==matricola ]
```

Esercizio 6

Definire nel linguaggio *Haskell* una struttura tabellare

```
Orario (Corso, Giorno, Ora1, Ora2, Aula)
```

in cui ogni tupla rappresenta l'allocazione di un corso in un certo giorno, nella fascia oraria limitata da ora iniziale ed ora finale, in una certa aula. Quindi, codificare in *Haskell* la funzione `auleOccupate` che, ricevendo in ingresso una tabella `orario`, un giorno ed una ora, computa la lista delle aule occupate in quella fascia oraria.

Esercizio 6

Definire nel linguaggio *Haskell* una struttura tabellare

```
Orario (Corso, Giorno, Ora1, Ora2, Aula)
```

in cui ogni tupla rappresenta l'allocazione di un corso in un certo giorno, nella fascia oraria limitata da ora iniziale ed ora finale, in una certa aula. Quindi, codificare in *Haskell* la funzione `auleOccupate` che, ricevendo in ingresso una tabella `orario`, un giorno ed una ora, computa la lista delle aule occupate in quella fascia oraria.

```
type Corso = String
type Giorno = String
type Ora = Int
type Aula = String
type Lezione = (Corso, Giorno, Ora, Ora, Aula)
type Orario = [Lezione]

auleOccupate :: Orario -> Giorno -> Ora -> [Aula]
auleOccupate orario giorno ora = [ a | (c, g, o1, o2, a) <- orario,
                                       g == giorno,
                                       ora >= o1, ora <= o2 ]
```

Esercizio 7

Definire nel linguaggio *Haskell* le seguenti strutture tabellari:

```
Libri(Titolo, Autore, Edizione)
Corsi(Nome, Docente, Libro)
Studenti(Cognome, Corso)
```

Quindi, codificare la funzione `studenti_autori` che, ricevendo in ingresso le tabelle `libri`, `corsi` e `studenti`, computa la lista degli studenti che sono autori di testi adottati nei loro corsi.

Esercizio 7

Definire nel linguaggio *Haskell* le seguenti strutture tabellari:

```
Libri(Titolo, Autore, Edizione)
Corsi(Nome, Docente, Libro)
Studenti(Cognome, Corso)
```

Quindi, codificare la funzione `studenti_autori` che, ricevendo in ingresso le tabelle `libri`, `corsi` e `studenti`, computa la lista degli studenti che sono autori di testi adottati nei loro corsi.

```
type Titolo = String
type Autore = String
type Edizione = String
type Nome = String
type Docente = String
type Libro = String
type Cognome = String
type Corso = String
type Libri = [(Titolo, Autore, Edizione)]
type Corsi = [(Nome, Docente, Libro)]
type Studenti = [(Cognome, Corso)]

studenti_autori :: Libri -> Corsi -> Studenti -> [ Cognome ]
studenti_autori libri corsi studenti = [ cg | (cg, c) <- studenti,
                                              (n, d, l) <- corsi,
                                              (t, a, e) <- libri,
                                               c == n, l == t, a == cg ]
```

Esercizio 8

Definire nel linguaggio *Haskell* le seguenti strutture tabellari:

```
Citta(NomeCitta, NumAbitanti)
Fiumi(NomeFiume, Lunghezza)
Attraversamenti(NomeFiume, NomeCitta)
```

Quindi, codificare la funzione `grosse_citta_con_lunghi_fiumi` che, ricevendo in ingresso le tabelle `citta`, `fiumi` ed `attraversamenti`, computa la lista delle città che hanno più di 1.000.000 di abitanti e sono attraversate da un fiume lungo più di 500 chilometri.

Esercizio 8

Definire nel linguaggio *Haskell* le seguenti strutture tabellari:

```
Citta(NomeCitta, NumAbitanti)
Fiumi(NomeFiume, Lunghezza)
Attraversamenti(NomeFiume, NomeCitta)
```

Quindi, codificare la funzione `grosse_citta_con_lunghi_fiumi` che, ricevendo in ingresso le tabelle `citta`, `fiumi` ed `attraversamenti`, computa la lista delle città che hanno più di 1.000.000 di abitanti e sono attraversate da un fiume lungo più di 500 chilometri.

```
type NomeCitta = String
type NumAbitanti = Int
type NomeFiume = String
type Lunghezza = Int

type Citta = [(NomeCitta, NumAbitanti)]
type Fiumi = [(NomeFiume, Lunghezza)]
type Attraversamenti = [(NomeFiume, NomeCitta)]

grosse_citta_con_lunghi_fiumi :: Citta -> Fiumi -> Attraversamenti -> [ NomeCitta ]
grosse_citta_con_lunghi_fiumi citta fiumi attraversamenti =
  [ c | (c, n) <- citta, (f, l) <- fiumi, (nf, nc) <- attraversamenti,
        c == nc, f == nf, n > 1000000, l > 500 ]
```

Esercizio 9

Definire nel linguaggio *Haskell* la forma funzionale **`computa`**, avente in ingresso una funzione **`f`**, una funzione **`g`** ed una lista di **`Integer`**, che restituisce la lista ottenuta sommando l'applicazione di **`f`** e **`g`** ad ogni elemento di **`lista`**, come nei seguenti esempi:

`f`	**`g`**	**`lista`**	**`computa f g lista`**
`quadrato`	`cubo`	`[1,2,3,4]`	`[2,12,36,80]`
`fattoriale`	`fibonacci`	`[2,4,3,0]`	`[3,27,8,1]`

Esercizio 9

Definire nel linguaggio *Haskell* la forma funzionale **`computa`**, avente in ingresso una funzione **`f`**, una funzione **`g`** ed una lista di **`Integer`**, che restituisce la lista ottenuta sommando l'applicazione di **`f`** e **`g`** ad ogni elemento di **`lista`**, come nei seguenti esempi:

f	g	lista	computa f g lista
quadrato	cubo	[1,2,3,4]	[2,12,36,80]
fattoriale	fibonacci	[2,4,3,0]	[3,27,8,1]

```
computa :: (Integer -> Integer) -> (Integer -> Integer) -> [Integer] -> [Integer]
computa f g [] = []
computa f g (p:coda) = (f p + g p):(computa f g coda)
```

oppure:

```
computa' :: (Integer -> Integer) -> (Integer -> Integer) -> [Integer] -> [Integer]
computa' f g lista = [f n + g n | n <- lista]
```

Esercizio 10

Definire nel linguaggio *Haskell* la funzione **`last`**, che restituisce l'ultimo elemento di una **`lista`** in ingresso, come nei seguenti esempi:

`lista`	**`last lista`**
`[]`	*non definita*
`[1,2,3]`	`3`
`["alfa"]`	`"alfa"`
`[(1,True),(2,False),(3,True)]`	`(3,True)`

Esercizio 10

Definire nel linguaggio *Haskell* la funzione **`last`**, che restituisce l'ultimo elemento di una **`lista`** in ingresso, come nei seguenti esempi:

`lista`	**`last lista`**
`[]`	*non definita*
`[1,2,3]`	`3`
`["alfa"]`	`"alfa"`
`[(1,True),(2,False),(3,True)]`	`(3,True)`

```
last :: [a] -> a
last (testa:[])  = testa
last(testa:coda) = last coda
```

Esercizio 11

Definire nel linguaggio *Haskell* la funzione **`shrink`** mediante pattern matching, sulla base del seguente protocollo: **`shrink :: [a] -> [a]`**. Tale funzione ha in ingresso una lista **`L`** e computa la lista degli elementi di **`L`** che si trovano in posizione dispari, come nei seguenti esempi:

`L`	**`shrink L`**
`[]`	`[]`
`[25]`	`[25]`
`[(1,True),(2,False)]`	`[(1,True)]`
`[10,20,30]`	`[10,30]`
`[fac,fib,fastFib,square]`	`[fac,fastFib]`

Esercizio 11

Definire nel linguaggio *Haskell* la funzione **`shrink`** mediante pattern matching, sulla base del seguente protocollo: **`shrink :: [a] -> [a]`**. Tale funzione ha in ingresso una lista **`L`** e computa la lista degli elementi di **`L`** che si trovano in posizione dispari, come nei seguenti esempi:

L	**shrink L**
`[]`	`[]`
`[25]`	`[25]`
`[(1,True),(2,False)]`	`[(1,True)]`
`[10,20,30]`	`[10,30]`
`[fac,fib,fastFib,square]`	`[fac,fastFib]`

```
shrink :: [a] -> [a]
shrink [] = []
shrink [x] = [x]
shrink (testa:(testa2:coda2)) = testa:(shrink coda2)
```

Esercizio 12

Definire nel linguaggio *Haskell* la funzione **`campionato`**, avente in ingresso una lista di squadre di calcio, che computa la lista di tutte le possibili partite del campionato, come nel seguente esempio:

```
campionato ["inter", "milan", "iuve"] =
    [("inter","milan"),("inter","iuve"),("milan","inter"),("milan","iuve"),("iuve","inter"),("iuve","milan")]
```

Esercizio 12

Definire nel linguaggio *Haskell* la funzione **`campionato`**, avente in ingresso una lista di squadre di calcio, che computa la lista di tutte le possibili partite del campionato, come nel seguente esempio:

campionato ["inter", "milan", "iuve"] =

[("inter","milan"),("inter","iuve"),("milan","inter"),("milan","iuve"),("iuve","inter"),("iuve","milan")]

```
campionato :: [String] -> [(String, String)]
campionato squadre =
    [(x,y) | x <- squadre, y <- squadre, x /= y ]
```

Esercizio 13

Definire nel linguaggio *Haskell* la funzione **`unzip :: [(a,b)] -> ([a],[b])`**, avente in ingresso una **`lista`** (non vuota) di coppie, che genera la corrispondente coppia di liste, come nei seguenti esempi:

lista	**unzip lista**
`[(1,'a')]`	`([1],['a'])`
`[(1,'a'),(2,'b')]`	`([1,2],['a','b'])`
`[(1,'a'),(2,'b'),(3,'c')]`	`([1,2,3],['a','b','c'])`

Esercizio 13

Definire nel linguaggio *Haskell* la funzione **`unzip :: [(a,b)] -> ([a],[b])`**, avente in ingresso una **`lista`** (non vuota) di coppie, che genera la corrispondente coppia di liste, come nei seguenti esempi:

lista	**unzip lista**
`[(1,'a')]`	`([1],['a'])`
`[(1,'a'),(2,'b')]`	`([1,2],['a','b'])`
`[(1,'a'),(2,'b'),(3,'c')]`	`([1,2,3],['a','b','c'])`

```
unzip :: [(a,b)] -> ([a], [b])
unzip lista = ([x | (x,_) <- lista], [y | (_,y) <- lista])
```

oppure:

```
unzip lista = (partex lista, partey lista)
              where
               partex []           = []
               partex ((x,_):coda) = x:(partex coda)
               partey []           = []
               partey ((_,y):coda) = y:(partey coda)
```

oppure:

```
unzip []           = ([],[])
unzip ((x,y):coda) = (x:listax, y:listay)
                     where
                      (listax, listay) = unzip coda
```

Esercizio 14

Mediante la notazione di specifica basata sul pattern-matching, definire nel linguaggio *Haskell* la funzione **`take`** che, ricevendo una 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]`
4	`"alfabeto"`	`"alfa"`
1	`[(25,"dicembre"),(1,"gennaio")]`	`[(25,"dicembre")]`
7	`[1,2,3]`	`[1,2,3]`
0	`[1,2,3]`	`[]`
5	`[]`	`[]`
0	`[]`	`[]`
0	`"alfa"`	`""`

Esercizio 14

Mediante la notazione di specifica basata sul pattern-matching, definire nel linguaggio *Haskell* la funzione **`take`** che, ricevendo una 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]`
4	`"alfabeto"`	`"alfa"`
1	`[(25,"dicembre"),(1,"gennaio")]`	`[(25,"dicembre")]`
7	`[1,2,3]`	`[1,2,3]`
0	`[1,2,3]`	`[]`
5	`[]`	`[]`
0	`[]`	`[]`
0	`"alfa"`	`""`

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (testa:coda) = testa:(take (n-1) coda)
```

Esercizio 15

Mediante la notazione di specifica basata sul pattern-matching, definire nel linguaggio *Haskell* la funzione **`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**
`[]`	`[]`	`True`
`[]`	`[1,2,3]`	`True`
`[1]`	`[]`	`False`
`[1]`	`[1,2,3]`	`True`
`[1,2]`	`[1,2,3]`	`True`
`[1,2]`	`[2,1,3]`	`False`
`[1,3]`	`[1,2,3,4]`	`False`

Esercizio 15

Mediante la notazione di specifica basata sul pattern-matching, definire nel linguaggio *Haskell* la funzione **`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**
`[]`	`[]`	`True`
`[]`	`[1,2,3]`	`True`
`[1]`	`[]`	`False`
`[1]`	`[1,2,3]`	`True`
`[1,2]`	`[1,2,3]`	`True`
`[1,2]`	`[2,1,3]`	`False`
`[1,3]`	`[1,2,3,4]`	`False`

```
prefix :: Eq a => [a] -> [a] -> Bool
prefix [] _                          =   True
prefix _ []                          =   False
prefix (testa1:coda1) (testa2:coda2)   =   testa1 == testa2 && prefix coda1 coda2
```

Esercizio 16

Definire nel linguaggio *Haskell* la funzione **`all`**, che riceve una **`lista`** di elementi ed una funzione booleana **`f`** avente come dominio lo stesso tipo degli elementi di **`lista`**. La funzione **`all`** restituisce **`True`** se tutti gli elementi di **`lista`** rendono vera la **`f`**, altrimenti restituisce **`False`**. (Nel caso in cui **`lista`** è vuota, **`all`** retituisce **`True`**).

Esercizio 16

Definire nel linguaggio *Haskell* la funzione **`all`**, che riceve una **`lista`** di elementi ed una funzione booleana **`f`** avente come dominio lo stesso tipo degli elementi di **`lista`**. La funzione **`all`** restituisce **`True`** se tutti gli elementi di **`lista`** rendono vera la **`f`**, altrimenti restituisce **`False`**. (Nel caso in cui **`lista`** è vuota, **`all`** retituisce **`True`**).

```
all :: Eq a => [a] -> (a -> Bool) -> Bool
all lista f = ([x | x <- lista, f x ] == lista)
```

```
all' [] f = True
all' (testa:coda) f = (f testa) && (all' coda f)
```

Esercizio 17

Dopo aver definito in *Haskell* 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]	False
4	[1,2,3]	True
2	[]	True

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

Dopo aver definito in *Haskell* 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]	False
4	[1,2,3]	True
2	[]	True

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

```
manca :: Eq a => a -> [a] -> Bool
manca _ [] = True
manca x (testa:coda) = x /= testa && manca x coda

unione :: Eq a => [a] -> [a] -> [a]
unione [] lista2 = lista2
unione (testa:coda) lista2 = if manca testa lista2 then
                                    testa:(unione coda lista2)
                             else
                                    unione coda lista2
```

Esercizio 18

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`naturals`**, avente in ingresso un intero **`n`** ≥ 0, che computa la lista dei primi **`n`** numeri naturali (partendo da zero), come nei seguenti esempi:

`n`	**`naturals n`**
0	`[]`
1	`[0]`
2	`[0,1]`
5	`[0,1,2,3,4]`

Esercizio 18

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`naturals`**, avente in ingresso un intero **`n`** ≥ 0, che computa la lista dei primi **`n`** numeri naturali (partendo da zero), come nei seguenti esempi:

`n`	**`naturals n`**
0	`[]`
1	`[0]`
2	`[0,1]`
5	`[0,1,2,3,4]`

```
naturals ::  Int -> [Int]
naturals 0 = []
naturals n = naturals(n-1)++[n-1]
```

Esercizio 19

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`catena`** (protocollo incluso), avente in ingresso una lista **`funzioni`** di funzioni unarie (che mappano un **`Int`** in un **`Int`**) ed un **`valore`** di tipo **`Int`**, la quale computa la composizione di tutte le funzioni (nella lista) applicata a **`valore`**. Formalmente, se f_1, f_2, …, f_n sono le funzioni nella lista, **`catena`** computa $f_1(f_2(\ldots(f_n($**valore**$))\ \ldots))$. Se la lista **funzioni** è vuota, **`catena`** restituisce **`valore`**. Ecco alcuni esempi:

funzioni	**`valore`**	**`catena funzioni valore`**
`[]`	3	3
`[quad, cube, fact]`	3	`quad(cube(fact(3))) = 46656`
`[quad, fib, fib]`	6	`quad(fib(fib(6))) = 441`
`[fib, quad]`	5	`fib(quad(5)) = 75025`

Esercizio 19

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`catena`** (protocollo incluso), avente in ingresso una lista **`funzioni`** di funzioni unarie (che mappano un **`Int`** in un **`Int`**) ed un **`valore`** di tipo **`Int`**, la quale computa la composizione di tutte le funzioni (nella lista) applicata a **`valore`**. Formalmente, se f_1, f_2, …, f_n sono le funzioni nella lista, **`catena`** computa $f_1(f_2(\ldots(f_n(\textbf{valore}))\ \ldots))$. Se la lista **funzioni** è vuota, **`catena`** restituisce **`valore`**. Ecco alcuni esempi:

funzioni	**`valore`**	**`catena funzioni valore`**
`[]`	3	3
`[quad, cube, fact]`	3	`quad(cube(fact(3))) = 46656`
`[quad, fib, fib]`	6	`quad(fib(fib(6))) = 441`
`[fib, quad]`	5	`fib(quad(5)) = 75025`

```
catena :: [Integer -> Integer] -> Integer -> Integer
catena [] valore = valore
catena (f:coda) valore = f(catena coda valore)
```

```
catena' :: [Integer -> Integer] -> Integer -> Integer
catena' funzioni valore = ((foldr (.) id) funzioni) valore
```

Esercizio 20

E’ data una struttura tabellare che rappresenta un insieme di famiglie, come nel seguente esempio:

Cognome	Madre	Padre	Figli
rossi	anna	andrea	alessio angela
bianchi	rosa	rino	rino renata rachele
neri	monica	mino	mina mauro
gialli	serena	sergio	sonia sofia serena

Si chiede di specificare nel linguaggio *Haskell* la funzione **`omonimie`** (protocollo e corpo) che, ricevendo in ingresso una lista di famiglie (come nell’esempio), computa la lista dei cognomi delle famiglie in cui almeno uno dei figli ha lo stesso nome di uno dei genitori (nell’esempio, **`[bianchi, gialli]`**).

Esercizio 20

E' data una struttura tabellare che rappresenta un insieme di famiglie, come nel seguente esempio:

Cognome	Madre	Padre	Figli
rossi	anna	andrea	alessio angela
bianchi	rosa	rino	rino renata rachele
neri	monica	mino	mina mauro
gialli	serena	sergio	sonia sofia serena

Si chiede di specificare nel linguaggio *Haskell* la funzione **`omonimie`** (protocollo e corpo) che, ricevendo in ingresso una lista di famiglie (come nell'esempio), computa la lista dei cognomi delle famiglie in cui almeno uno dei figli ha lo stesso nome di uno dei genitori (nell'esempio, **`[bianchi, gialli]`**).

```
omonimie :: Famiglie -> [ Cognome ]
omonimie famiglie =
  [ c | (c, m, p, f) <- famiglie, n <- f, n==m || n==p ]
```

Esercizio 21

Sono date le seguenti due strutture tabellari:

```
Cliente(nomecliente, indirizzo, cittacliente)
Filiale(nomefiliale, cittafiliale, deposito)
```

Si chiede di specificare nel linguaggio *Haskell* protocollo e corpo della funzione `stessacitta` che, ricevendo in ingresso una lista di clienti ed una lista di filiali, computa la lista delle coppie `(nomecliente, nomefiliale)` per le quali il cliente `nomecliente` vive nella stessa cittàEsercizio 31 in cui si trova la filiale `nomefiliale`.

Esercizio 21

Sono date le seguenti due strutture tabellari:

```
Cliente(nomecliente, indirizzo, cittacliente)
Filiale(nomefiliale, cittafiliale, deposito)
```

Si chiede di specificare nel linguaggio *Haskell* protocollo e corpo della funzione `stessacitta` che, ricevendo in ingresso una lista di clienti ed una lista di filiali, computa la lista delle coppie `(nomecliente, nomefiliale)` per le quali il cliente `nomecliente` vive nella stessa cittàEsercizio 31 in cui si trova la filiale `nomefiliale`.

```
stessa_citta :: Clienti -> Filiali -> [(String, String)]
stessa_citta clienti filiali =
  [ (c,f) | (c, i, cc) <- clienti, (f, cf, d) <- filiali, cc==cf ]
```

Esercizio 22

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`applica`** (protocollo incluso), avente in ingresso due funzioni ed una lista di interi. Le due funzioni mappano un intero in un altro intero. La funzione **`applica`** genera la lista di interi in cui ogni elemento rappresenta il valore della applicazione della prima funzione al risultato della seconda funzione applicata ad un elemento della lista in ingresso. Ecco alcuni esempi:

- `applica quadrato cubo [1,2,3] = [1,64,729]`
- `applica doppio fattoriale [1,3,5] = [2,12,240]`
- `applica fattoriale doppio [1,3,5] = [2,720,3628800]`

Esercizio 22

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`applica`** (protocollo incluso), avente in ingresso due funzioni ed una lista di interi. Le due funzioni mappano un intero in un altro intero. La funzione **`applica`** genera la lista di interi in cui ogni elemento rappresenta il valore della applicazione della prima funzione al risultato della seconda funzione applicata ad un elemento della lista in ingresso. Ecco alcuni esempi:

- `applica quadrato cubo [1,2,3] = [1,64,729]`
- `applica doppio fattoriale [1,3,5] = [2,12,240]`
- `applica fattoriale doppio [1,3,5] = [2,720,3628800]`

```
applica :: (Integer->Integer) -> (Integer->Integer) -> [Integer] -> [Integer]
applica f g [] = []
applica f g (testa:coda) = (f  (g testa)):(applica f g coda)
```

```
applica' f g lista = map (f . g) lista
```

Esercizio 23

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`dispari`** (protocollo incluso) che, avente in ingresso un intero **`n`** ≥ 0, computa la lista dei primi **n** numeri dispari, come nei seguenti esempi:

n	**dispari n**
0	`[]`
1	`[1]`
2	`[1,3]`
3	`[1,3,5]`
10	`[1,3,5,7,9,11,13,15,17,19]`

Esercizio 23

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`dispari`** (protocollo incluso) che, avente in ingresso un intero **`n`** ≥ 0, computa la lista dei primi **n** numeri dispari, come nei seguenti esempi:

n	**dispari n**
0	`[]`
1	`[1]`
2	`[1,3]`
3	`[1,3,5]`
10	`[1,3,5,7,9,11,13,15,17,19]`

```
dispari :: Int -> [Int]
dispari 0 = []
dispari n = dispari(n-1) ++ [n*2-1]
```

Esercizio 24

Definire nel linguaggio *Haskell* la funzione **`coppie`** (protocollo incluso) che, avente in ingresso una **`lista`** **(**senza duplicati) ed una funzione booleana **`f`** applicabile a due elementi di **`lista`**, computa la lista delle coppie di elementi di **`lista`** per le quali la funzione **`f`** risulta vera, come nei seguenti esempi:

`lista`	**`f`**	**`coppie lista f`**
`[1..10]`	`(==)`	`[(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10)]`
`"tre"`	`(/=)`	`[('t','r'),('t','e'),('r','t'),('r','e'),('e','t'),('e','r')]`

Esercizio 24

Definire nel linguaggio *Haskell* la funzione **`coppie`** (protocollo incluso) che, avente in ingresso una **`lista`** (senza duplicati) ed una funzione booleana **`f`** applicabile a due elementi di **`lista`**, computa la lista delle coppie di elementi di **`lista`** per le quali la funzione **`f`** risulta vera, come nei seguenti esempi:

lista	f	coppie lista f
`[1..10]`	`(==)`	`[(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10)]`
`"tre"`	`(/=)`	`[('t','r'),('t','e'),('r','t'),('r','e'),('e','t'),('e','r')]`

```
coppie :: [a] -> (a -> a -> Bool) -> [(a,a)]
coppie lista f = [ (x,y) | x <- lista, y <- lista, f x y ]
```

Esercizio 25

Definire nel linguaggio *Haskell* la funzione **`discendenti`** (protocollo incluso) che, avente in ingresso una **`genealogia`** rappresentata da una lista di genitori associati alla rispettiva lista di figli ed una **`persona`**, computa la lista dei discendenti di **`persona`**, come nel seguente esempio:

Genitore	*Figli*
guido	luisa, franco, elena
luisa	andrea, dario, gino
franco	giovanni, paola, letizia, sofia
elena	emma, zeno
andrea	rino, anna

`genealogia`

Esercizio 25

Definire nel linguaggio *Haskell* la funzione **`discendenti`** (protocollo incluso) che, avente in ingresso una **`genealogia`** rappresentata da una lista di genitori associati alla rispettiva lista di figli ed una **`persona`**, computa la lista dei discendenti di **`persona`**, come nel seguente esempio:

Genitore	*Figli*
guido	luisa, franco, elena
luisa	andrea, dario, gino
franco	giovanni, paola, letizia, sofia
elena	emma, zeno
andrea	rino, anna

`genealogia`

```
discendenti :: [(String,[String])] -> String -> [ String ]

discendenti genealogia persona =
  [ f | (genitore,figli) <- genealogia, genitore == persona, f <- figli ] ++
  [ d | (genitore,figli) <- genealogia, genitore == persona,
                        f <- figli, d <- (discendenti genealogia f) ]
```

Esercizio 26

Dopo aver definito nel linguaggio *Haskell* le seguenti tre strutture tabellari,

- `Libri(titolo, autore, anno_pubblicazione)`
- `Prestiti(libro, lettore)`
- `Lettori(nome, anno_nascita, citta)`

specificare la funzione **`autolettori`** che, ricevendo in ingresso tali strutture tabellari, una **`citta`** ed un **`anno`**, resituisce la lista dei nomi dei lettori di **`citta`** nati prima dell'**`anno`**, che hanno in prestito un libro di cui sono loro stessi autori.

Esercizio 26

Dopo aver definito nel linguaggio *Haskell* le seguenti tre strutture tabellari,

- `Libri(titolo, autore, anno_pubblicazione)`
- `Prestiti(libro, lettore)`
- `Lettori(nome, anno_nascita, citta)`

specificare la funzione **`autolettori`** che, ricevendo in ingresso tali strutture tabellari, una **`citta`** ed un **`anno`**, resituisce la lista dei nomi dei lettori di **`citta`** nati prima dell'**`anno`**, che hanno in prestito un libro di cui sono loro stessi autori.

```
type Libri = [(String, String, Int)]
type Prestiti = [(String, String)]
type Lettori = [(String, Int, String)]

autolettori :: Libri -> Prestiti -> Lettori -> String -> Int -> [String]

autolettori libri prestiti lettori citta anno =
  [n | (t,a,_) <- libri, (l,le) <- prestiti, (n,an,c) <- lettori,
       t == l, le == n, a == n, an < anno, c == citta ]]
```

Esercizio 27

Definire nel linguaggio *Haskell* mediante pattern-matching la funzione **`medie`** (protocollo incluso), avente in ingresso una lista **`F`** di funzioni, una lista **`G`** di funzioni ed una lista **`R`** di numeri reali. Le tre liste hanno lo stesso numero $n \geq 0$ di elementi. La funzione **`medie`** genera una lista di n numeri reali m_i così definita:

$$\forall i \in [1..n] \;\; (m_i = (\mathbf{F}[i](\mathbf{R}[i]) + \mathbf{G}[i](\mathbf{R}[i])) / 2)$$

Esercizio 27

Definire nel linguaggio *Haskell* mediante pattern-matching la funzione **medie** (protocollo incluso), avente in ingresso una lista **F** di funzioni, una lista **G** di funzioni ed una lista **R** di numeri reali. Le tre liste hanno lo stesso numero $n \geq 0$ di elementi. La funzione **medie** genera una lista di n numeri reali m_i così definita:

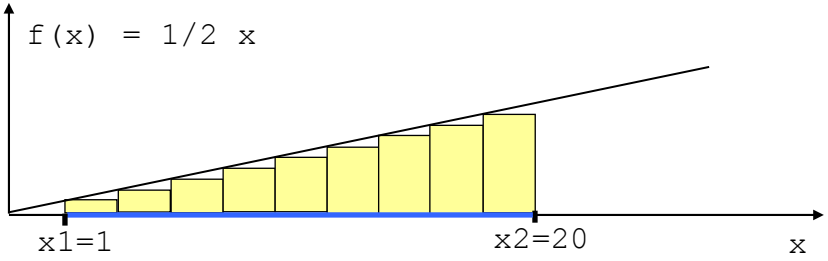
$$\forall i \in [1..n] \;\; (m_i = (\mathbf{F}[i](\mathbf{R}[i]) + \mathbf{G}[i](\mathbf{R}[i])) / 2)$$

```
medie :: [Float->Float] -> [Float->Float] -> [Float] -> [Float]

medie [] [] [] = []
medie (f:codaF) (g:codaG) (r:codaR) =
                                  (((f r)+(g r))/2):(medie codaF codaG codaR)
```

Esercizio 28

Definire nel linguaggio *Haskell* la funzione **`integrale`** (protocollo incluso), avente in ingresso una funzione **`f`** (che mappa un numero reale in un numero reale), i due estremi di integrazione **`x1`** e **`x2`** (**`x1`** < **`x2`**) e un intero **`n`** che rappresenta il numero di segmenti in cui dividere l'intervallo di integrazione [**`x1`**, **`x2`**]. Tale funzione computa l'integrale (approssimato, come sommatoria dell'area dei rettangoli) di **`f`** in [**`x1`**, **`x2`**], come nel seguente esempio:

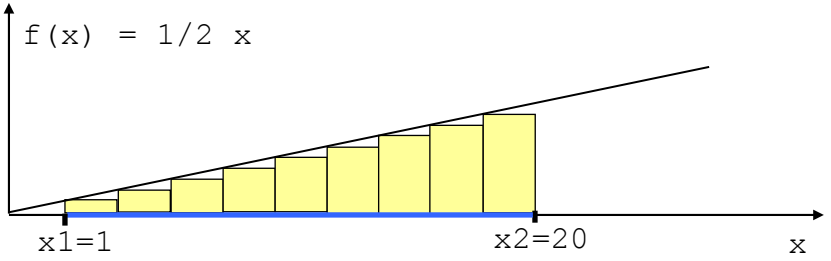


in cui: **`integrale`** `f 1 20 9 = 89.72222` (sommatoria di 9 aree di rettangoli con uguale base).

Si noti che la precisione del risultato aumenta con **n**, ad esempio: **`integrale`** `f 1 20 10000 = 99.75184`

Esercizio 28

Definire nel linguaggio *Haskell* la funzione **`integrale`** (protocollo incluso), avente in ingresso una funzione **`f`** (che mappa un numero reale in un numero reale), i due estremi di integrazione **`x1`** e **`x2`** (**`x1`** < **`x2`**) e un intero **`n`** che rappresenta il numero di segmenti in cui dividere l'intervallo di integrazione [**`x1`**, **`x2`**]. Tale funzione computa l'integrale (approssimato, come sommatoria dell'area dei rettangoli) di **`f`** in [**`x1`**, **`x2`**], come nel seguente esempio:



in cui: **`integrale`** `f 1 20 9 = 89.72222` (sommatoria di 9 aree di rettangoli con uguale base).

Si noti che la precisione del risultato aumenta con **n**, ad esempio: **`integrale`** `f 1 20 10000 = 99.75184`

```
integrale :: (Float -> Float) -> Float -> Float -> Int -> Float
integrale f x1 x2 0 = 0
integrale f x1 x2 n = (f x1) * dx + (integrale f (x1 + dx) x2 (n-1))
                        where
                          dx = (x2 - x1) / fromIntegral n
```

Esercizio 29

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`uguali`** che, avente in ingresso una lista, stabilisce se tutti gli elementi di tale lista siano uguali fra loro (per definizione, nel caso di lista vuota, **`uguali`** risulta vera). Si richiede anche la specifica completa del protocollo di tale funzione.

Esercizio 29

Definire nel linguaggio *Haskell*, mediante la notazione di pattern-matching, la funzione **`uguali`** che, avente in ingresso una lista, stabilisce se tutti gli elementi di tale lista siano uguali fra loro (per definizione, nel caso di lista vuota, **`uguali`** risulta vera). Si richiede anche la specifica completa del protocollo di tale funzione.

```
uguali :: Eq a => [a] -> Bool
uguali [] = True
uguali [_] = True
uguali(testa:(testa2:coda2)) = testa == testa2 && uguali(testa2:coda2)
```

Esercizio 30

Definire nel linguaggio Haskell la funzione **`affetta`** (protocollo incluso), la quale, avente in ingresso un numero intero **`n`**>0 ed una **`lista`**, genera la lista di liste ottenuta prelevando ripetutamente **`n`** elementi da **`lista`**, come nei seguenti esempi:

n	**lista**	**(affetta n lista)**
2	`[1,2,3,4,5,6,7,8,9,10]`	`[[1,2],[3,4],[5,6],[7,8],[9,10]]`
3	`[True,False,True,False,True]`	`[[True,False,True],[False,True]]`
1	`"alfabeto"`	`["a","l","f","a","b","e","t","o"]`

Esercizio 30

Definire nel linguaggio Haskell la funzione **affetta** (protocollo incluso), la quale, avente in ingresso un numero intero **n**>0 ed una **lista**, genera la lista di liste ottenuta prelevando ripetutamente **n** elementi da **lista**, come nei seguenti esempi:

n	lista	(affetta n lista)
2	[1,2,3,4,5,6,7,8,9,10]	[[1,2],[3,4],[5,6],[7,8],[9,10]]
3	[True,False,True,False,True]	[[True,False,True],[False,True]]
1	"alfabeto"	["a","l","f","a","b","e","t","o"]

```
affetta :: Int -> [a] -> [[a]]
affetta _ [] = []
affetta n lista = (take n lista):(affetta n (drop n lista))
```

Esercizio 31

Codificare nel linguaggio *Haskell* la funzione **`compaesaniCoscritti`** (protocollo incluso), la quale, avente in ingresso una lista **`anagrafe`** di triple (**`nome`**, **`anno`**, **`citta`**) e una **`persona`**, computa la lista dei nomi delle persone in anagrafe (escludendo **`persona`**) che sono nate nello stesso anno di **`persona`** ed abitano nella stessa città di **`persona`**.

Esercizio 31

Codificare nel linguaggio *Haskell* la funzione **`compaesaniCoscritti`** (protocollo incluso), la quale, avente in ingresso una lista **`anagrafe`** di triple (**`nome`**, **`anno`**, **`citta`**) e una **`persona`**, computa la lista dei nomi delle persone in anagrafe (escludendo **`persona`**) che sono nate nello stesso anno di **`persona`** ed abitano nella stessa città di **`persona`**.

```
type Anagrafe = [(String, Integer, String)]

compaesaniCoscritti :: Anagrafe -> String -> [String]

compaesaniCoscritti anagrafe persona =
  [ p' | (p, a, c) <- anagrafe,
         (p', a', c') <- anagrafe,
          p == persona, p' /= persona, a' == a, c == c' ]
```

Esercizio 32

Definire nel linguaggio *Haskell* una tabella in cui ogni riga associa al nome di uno studente la lista dei suoi esami. Ogni esame è rappresentato dal nome del corso e dal voto conseguito. Quindi, codificare la funzione **`superato`** (protocollo incluso) che, ricevendo in ingresso un **`esame`** (nome del corso) e la tabella degli studenti con i relativi esami, genera la lista dei nomi degli studenti che hanno superto l'**`esame`**.

Esercizio 32

Definire nel linguaggio *Haskell* una tabella in cui ogni riga associa al nome di uno studente la lista dei suoi esami. Ogni esame è rappresentato dal nome del corso e dal voto conseguito. Quindi, codificare la funzione **`superato`** (protocollo incluso) che, ricevendo in ingresso un **`esame`** (nome del corso) e la tabella degli studenti con i relativi esami, genera la lista dei nomi degli studenti che hanno superto l'**`esame`**.

```
type Studenti = [(String, [(String, Integer)])]

superato :: String -> Studenti -> [String]

superato esame studenti =
  [ s | (s, e) <- studenti, (c, v) <- e, esame == c ]
```

Esercizio 33

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni di numeri interi:

```
data Expr = Const Int
          | Var String
          | Plus Expr Expr
          | Minus Expr Expr
          | Mult Expr Expr

type Stato = [(String, Int)]
```

in cui `Const`, `Var`, `Plus`, `Minus` e `Mult` si riferiscono, rispettivamente, a una costante, una variabile, una somma, una differenza e una moltiplicazione, mentre Stato si riferisce alla associazione tra le variabili e i corrispondenti valori. Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione `valexpr` (protocollo incluso) che, ricevendo in ingresso una espressione `e` ed uno stato `s`, genera il valore di `e` nello stato `s`.

Esercizio 33

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni di numeri interi:

```
data Expr = Const Int
          | Var String
          | Plus Expr Expr
          | Minus Expr Expr
          | Mult Expr Expr

type Stato = [(String, Int)]
```

in cui `Const`, `Var`, `Plus`, `Minus` e `Mult` si riferiscono, rispettivamente, a una costante, una variabile, una somma, una differenza e una moltiplicazione, mentre Stato si riferisce alla associazione tra le variabili e i corrispondenti valori. Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione `valexpr` (protocollo incluso) che, ricevendo in ingresso una espressione `e` ed uno stato `s`, genera il valore di `e` nello stato `s`.

```
valore :: String -> Stato -> Int
valore i s = head [v | (nome,v) <- s, nome == i ]

valexpr :: Expr -> Stato -> Int
valexpr (Const n) _ = n
valexpr (Var i) s = valore i s
valexpr (Plus x y) s = (valexpr x s) + (valexpr y s)
valexpr (Minus x y) s = (valexpr x s) - (valexpr y s)
valexpr (Mult x y) s = (valexpr x s) * (valexpr y s)
```

Esercizio 34

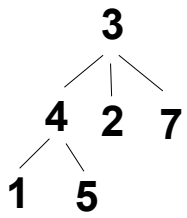
È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad alberi *n*-ari di interi:

```
data Albero = Nodo Int [Albero]
```

Ecco un esempio di valore di `Albero`:

```
frassino :: Albero
frassino = (Nodo 3 [(Nodo 4 [(Nodo 1 []),(Nodo 5 [])]),(Nodo 2 []),(Nodo 7 [])])
```

che corrisponde al seguente albero:



Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione `sommalbero` (protocollo incluso) che, ricevendo in ingresso un `Albero`, genera la somma di tutti i valori contenuti nell'albero. Nel nostro esempio avremmo:

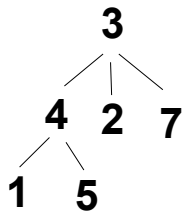
```
> sommalbero frassino
22
```

Esercizio 34

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad alberi *n*-ari di interi:

```
data Albero = Nodo Int [Albero]
```

```
frassino :: Albero
frassino = (Nodo 3 [(Nodo 4 [(Nodo 1 []),(Nodo 5 [])]),(Nodo 2 []),(Nodo 7 [])])
```



Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione `sommalbero` (protocollo incluso) che, ricevendo in ingresso un `Albero`, genera la somma di tutti i valori contenuti nell'albero.

```
sommalbero :: Albero -> Int
sommalbero(Nodo n figli) = n + (somma figli)

somma :: [Albero] -> Int
somma [] = 0
somma (x:xs) = (sommalbero x) + (somma xs)
```

oppure:

```
sommalbero :: Albero -> Int
sommalbero (Nodo n figli) = n + sum [ sommalbero f | f <- figli ]
```

Esercizio 35

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni di liste di interi:

```
data Lexpr = List [Int]
           | Var String
           | Cat Lexpr Lexpr
           | Rev Lexpr

type State = [(String, [Int])]
```

in cui `List`, `Var`, `Cat` e `Rev` si riferiscono, rispettivamente, a una lista di interi, una variabile di tipo lista di interi, una concatenazione di liste di interi e una inversione di lista di interi, mentre `State` si riferisce alla associazione tra le variabili e le corrispondenti liste di interi. Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione `eval` (protocollo incluso) che, ricevendo in ingresso una espressione di liste di interi `e` ed uno stato `s`, genera il valore di `e` nello stato `s`. Si può fare uso delle funzioni della libreria standard di manipolazione di liste (di cui non è richiesta la specifica).

Esercizio 35

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni di liste di interi:

```
data Lexpr = List [Int]
           | Var String
           | Cat Lexpr Lexpr
           | Rev Lexpr

type State = [(String, [Int])]
```

in cui `List`, `Var`, `Cat` e `Rev` si riferiscono, rispettivamente, a una lista di interi, una variabile di tipo lista di interi, una concatenazione di liste di interi e una inversione di lista di interi, mentre `State` si riferisce alla associazione tra le variabili e le corrispondenti liste di interi. Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione `eval` (protocollo incluso) che, ricevendo in ingresso una espressione di liste di interi `e` ed uno stato `s`, genera il valore di `e` nello stato `s`. Si può fare uso delle funzioni della libreria standard di manipolazione di liste (di cui non è richiesta la specifica).

```
valore :: String -> State -> [Int]
valore n s = head [lista | (nome,lista) <- s, nome == n ]

eval :: Lexpr -> State -> [Int]
eval (List lista) _ = lista
eval (Var n) s = valore n s
eval (Cat x y) s = (eval x s) ++ (eval y s)
eval (Rev x) s = reverse (eval x s)
```

Esercizio 36

Specificare nel linguaggio *Haskell* la classe di tipi `AddNeg`, nella quale sono definite due funzioni:

- `add`: somma di due elementi dello stesso tipo;
- `neg`: negazione di un elemento.

Quindi, istanziare tale classe mediante i seguenti tipi:

- `Int`: in cui `add` è la somma aritmetica, mentre `neg` è il cambiamento di segno.
- `Bool`: in cui `add` è `False` se e solo se entrambi gli operandi sono `False` (altrimenti è `True`), mentre `neg` è la negazione logica.
- `[Int]`: in cui `add` genera la lista (di lunghezza minima tra le due) in cui ogni elemento è la somma dei rispettivi elementi nelle due liste, mentre `neg` è la lista in cui tutti i numeri (rispetto alla lista operando) sono cambiati di segno.

Esercizio 36

Specificare nel linguaggio *Haskell* la classe di tipi `AddNeg`, nella quale sono definite due funzioni:

- `add`: somma di due elementi dello stesso tipo;
- `neg`: negazione di un elemento.

```
class AddNeg a where
  add :: a -> a -> a
  neg :: a -> a

instance AddNeg Int where
  add n m = n + m
  neg n = -n

instance AddNeg Bool where
  add b1 b2 = b1 || b2
  neg b =  not b

instance AddNeg [Int] where
  add [] _ = []
  add _ [] = []
  add  (x:xs) (y:ys) = (x+y):(add xs ys)
  neg [] = []
  neg (x:xs) = (-x):(neg xs)
```

Esercizio 37

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni su insiemi di interi:

```
data Expr = Set [Int]
          | Var String
          | Union Expr Expr
          | Inter Expr Expr
          | Select (Int -> Bool) Expr

type State = [(String, [Int])]
```

in cui `Set`, `Var`, `Union`, `Inter` e `Select` si riferiscono, rispettivamente, ad una istanza, una variabile, una unione insiemistica, una intersezione insiemistica e una selezione (il cui primo argomento è la funzione filtro), mentre `State` si riferisce alla associazione tra le variabili e le corrispondenti istanze (stato). Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, la funzione computa (protocollo incluso) che, ricevendo in ingresso una espressione di insiemi di interi ed uno stato, genera il valore della espressione in quello stato. Sono disponibili le seguenti funzioni ausiliarie (di cui non è richiesta la codifica): `head`, `tail` ed `elem` (appartenenza di un elemento ad una lista).

Esercizio 37

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni su insiemi di interi:

```
data Expr = Set [Int]
          | Var String
          | Union Expr Expr
          | Inter Expr Expr
          | Select (Int -> Bool) Expr

type State = [(String, [Int])]
```

```
computa :: Expr -> State -> [Int]
computa (Set xs) _ = xs
computa (Var nome) stato = head [ l | (n,l) <- stato, n == nome ]
computa (Union e1 e2) s = unione (computa e1 s) (computa e2 s)
computa (Inter e1 e2) s = intersezione (computa e1 s) (computa e2 s)
computa (Select p e) s = seleziona p (computa e s)

unione [] ys = ys
unione (x:xs) ys = if elem x ys then unione xs ys else x:(unione xs ys)

intersezione [] ys = []
intersezione (x:xs) ys = if elem x ys then x:(intersezione xs ys)
                         else intersezione xs ys
seleziona _ [] = []
seleziona p (x:xs) = if p x then x:(seleziona p xs) else seleziona p xs
```

Esercizio 38

È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Tree = Node Int [Tree]
```

Si chiede di codificare la funzione `somma` che, ricevendo in ingresso un albero, computa la somma dei numeri memorizzati nell'albero.

Esercizio 38

È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Tree = Node Int [Tree]
```

Si chiede di codificare la funzione `somma` che, ricevendo in ingresso un albero, computa la somma dei numeri memorizzati nell'albero.

```
somma :: Tree -> Int
somma' :: [Tree] -> Int

somma' [] = 0
somma' ((Node n figli):coda) = n + (somma' figli) + (somma' coda)

somma t = somma' [t]
```

oppure:

```
somma :: Tree -> Int
somma (Node n figli) = n + sum (map somma figli)
```

Esercizio 39

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni di vettori di numeri reali:

```
data Expr = Vec [Float]
          | Var String
          | Sum Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr

type State = [(String, [Float])]
```

in cui `Vec`, `Var`, `Sum`, `Sub`, `Mul`, e `Div` si riferiscono, rispettivamente, ad una istanza, una variabile, una somma di vettori, una differenza di vettori, una moltiplicazione di vettori e una divisione di vettori. Ogni operazione aritmetica su due vettori genera un vettore (di dimensione minima far i due vettori), in cui ogni numero *i*-esimo del vettore generato corrisponde al risultato dell'operazione aritmetica applicata agli elementi *i*-esimi dei due vettori operando.

Si chiede di definire in *Haskell*, mediante la notazione di pattern-matching, le seguenti funzioni (protocollo incluso):

- `matVec`: riceve in ingresso un vettore, un operatore aritmetico (+, -, *, /) ed un altro vettore; computa l'operazione aritmetica sui due vettori corrispondente all'operatore aritmetico;
- `eval`: riceve in ingresso una espressione di vettori e uno stato; computa il valore della espressione nello stato.

Esercizio 39

È data la seguente dichiarazione nel linguaggio *Haskell*, relativa ad espressioni di vettori di numeri reali:

```
data Expr = Vec [Float]
          | Var String
          | Sum Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr

type State = [(String, [Float])]
```

```
matVec :: [Float] -> (Float -> Float -> Float) -> [Float] -> [Float]
matVec [] _ _ = []
matVec _ _ [] = []
matVec (x:xs) op (y:ys) = (op x y):(matVec xs op ys)

eval :: Expr -> State -> [Float]
eval (Vec xs) _ = xs
eval (Var nome) stato = head [ l | (n,l) <- stato, n == nome ]
eval (Sum e1 e2) s = matVec (eval e1 s) (+) (eval e2 s)
eval (Sub e1 e2) s = matVec (eval e1 s) (-) (eval e2 s)
eval (Mul e1 e2) s = matVec (eval e1 s) (*) (eval e2 s)
eval (Div e1 e2) s = matVec (eval e1 s) (/) (eval e2 s)
```

Esercizio 40

È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Albero = Nodo Int [Albero]
```

Si chiede di codificare le seguenti funzioni (protocollo incluso):

- `concatena`: riceve una lista di liste di interi e restituisce la concatenazione delle liste di interi;
- `appiattisci`: riceve un albero di interi e restituisce la lista di interi contenuta nell'albero;
- `complementa`: riceve un albero di interi e restituisce un albero di interi isomorfo all'albero in ingresso, in cui tutti i numeri sono stati cambiati di segno.

Esercizio 40

È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Albero = Nodo Int [Albero]
```

Si chiede di codificare le seguenti funzioni (protocollo incluso):

- `concatena`: riceve una lista di liste di interi e restituisce la concatenazione delle liste di interi;
- `appiattisci`: riceve un albero di interi e restituisce la lista di interi contenuta nell'albero;
- `complementa`: riceve un albero di interi e restituisce un albero di interi isomorfo all'albero in ingresso, in cui tutti i numeri sono stati cambiati di segno.

```
concatena :: [[Int]] -> [Int]
concatena [] = []
concatena (x:xs) = x ++ (concatena xs)

appiattisci :: Albero -> [Int]
appiattisci (Nodo n figli) = [n]++(concatena (map appiattisci figli))

complementa :: Albero -> Albero
complementa (Nodo n figli) = (Nodo (-n) (map complementa figli))
```

Esercizio 41

È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Albero = Nodo Float [Albero]
```

Si chiede di codificare la funzione `media` (protocollo incluso), la quale riceve in ingresso un albero e restituisce la media dei numeri nell'albero.

Esercizio 41

È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Albero = Nodo Float [Albero]
```

Si chiede di codificare la funzione `media` (protocollo incluso), la quale riceve in ingresso un albero e restituisce la media dei numeri nell'albero.

```
somma :: Albero -> Float
somma (Nodo n figli) = n + sum [somma f | f <- figli ]

num :: Albero -> Float
num (Nodo n figli) = 1 + sum [num f | f <- figli ]

media :: Albero -> Float
media a = s / n
          where s = somma a
                n = num a
```

Esercizio 42

Specificare nel linguaggio *Haskell* la classe di tipi Expr, nella quale è definita la funzione binaria `somma`. Quindi, istanziare la classe `Expr` mediante i seguenti tipi:

- `Int`: in cui `somma` è la classica somma aritmetica tra interi;
- `[Int]`: in cui `somma` genera una lista di interi di lunghezza minima fra i due operandi, in cui ogni intero nella posizione *ì*-esima del risultato corrisponde alla somma aritmetica dei due interi nella posizione *ì*-esima nei rispettivi operandi, come nel seguente esempio:

  ```
  somma [1,2,3] [4,5,6,7] = [5,7,9]
  ```

- `[[Int]]`: in cui `somma` genera una lista di liste di interi di lunghezza minima fra le due liste operando, in cui ogni lista nella posizione *ì*-esima del risultato corrisponde alla concatenazione delle due liste nella posizione *ì*-esima nei rispettivi operandi, come nel seguente esempio:

  ```
  somma [[1,2],[3,4,5],[6,7]] [[8,9],[10]] = [[1,2,8,9],[3,4,5,10]].
  ```

Esercizio 42

Specificare nel linguaggio *Haskell* la classe di tipi Expr, nella quale è definita la funzione binaria `somma`. Quindi, istanziare la classe `Expr` mediante i seguenti tipi:

- `Int`: in cui `somma` è la classica somma aritmetica tra interi;
- `[Int]`: in cui `somma` genera una lista di interi di lunghezza minima fra i due operandi, in cui ogni intero nella posizione *i*-esima del risultato corrisponde alla somma aritmetica dei due interi nella posizione *i*-esima nei rispettivi operandi, come nel seguente esempio:

  `somma [1,2,3] [4,5,6,7] = [5,7,9]`
- `[[Int]]`: in cui `somma` genera una lista di liste di interi di lunghezza minima fra le due liste operando, in cui ogni lista nella posizione *i*-esima del risultato corrisponde alla concatenazione delle due liste nella posizione *i*-esima nei rispettivi operandi, come nel seguente esempio:

  `somma [[1,2],[3,4,5],[6,7]] [[8,9],[10]] = [[1,2,8,9],[3,4,5,10]].`

```
class Expr a where
  somma :: a-> a-> a

instance Expr Int where
  somma n m = n + m

instance Expr [Int] where
  somma numeri1 numeri2 = [n1+n2 | (n1,n2) <- zip numeri1 numeri2]

instance Expr [[Int]] where
  somma lista1 lista2 = [num1++num2 | (num1,num2) <- zip lista1 lista2]
```

Esercizio 43

È data la seguente dichiarazione Haskell, relativa ad espressioni di interi:

```
data Expr = Number Int
          | Const String
          | Plus Expr Expr
          | Fun (Int->Int) Expr

type Bindings = [(String, Int)]
```

in cui `Bindings` associa ad ogni costante simbolica un numero. Si chiede di definire mediante la notazione di pattern matching la funzione `computa` (protocollo incluso), avente in ingresso una espressione ed una lista di bindings, la quale computa il risultato della espressione.

Esercizio 43

È data la seguente dichiarazione Haskell, relativa ad espressioni di interi:

```
data Expr = Number Int
          | Const String
          | Plus Expr Expr
          | Fun (Int->Int) Expr

type Bindings = [(String, Int)]
```

in cui `Bindings` associa ad ogni costante simbolica un numero. Si chiede di definire mediante la notazione di pattern matching la funzione `computa` (protocollo incluso), avente in ingresso una espressione ed una lista di bindings, la quale computa il risultato della espressione.

```
computa :: Expr -> Bindings -> Int
computa (Number n) _ = n
computa (Const s) bs = head [ n | (name, n) <- bs, name == s ]
computa (Plus e1 e2) bs = (computa e1 bs) + (computa e2 bs)
computa (Fun f e) bs = f (computa e bs)
```

Esercizio 44

É data la seguente dichiarazione nel linguaggio *Haskell,* relativa alla specifica di espressioni regolari:

```
data Regex = Symbol Char
           | Opt Regex
           | Star Regex
           | Range String
           | Cat Regex Regex
           | Alt Regex Regex
```

in cui `Symbol`, `Opt`, `Star`, `Range`, `Cat` ed `Alt` si riferiscono, rispettivamente, ad un carattere dell'alfabeto, l'opzionalità, la ripetizione zero o più volte, un range di caratteri, la concatenazione e l'alternativa. Si chiede di definire la funzione `mostra` (protocollo incluso) che, ricevendo in ingresso una espressione regolare, genera la stringa di testo che la rappresenta, come nel seguente esempio:

```
x :: Regex
x = (Alt (Star (Cat (Star (Alt (Symbol 'a') (Symbol 'b')))(Opt (Range "cde")))) (Symbol 'z'))
```

```
> mostra x
  (((a|b)*[cde]?)*|z)
```

É richiesto che le espressioni generate dagli operatori binari `Cat` ed `Alt` siano racchiuse tra parentesi tonde.

Esercizio 44

É data la seguente dichiarazione nel linguaggio *Haskell,* relativa alla specifica di espressioni regolari:

```
data Regex = Symbol Char
           | Opt Regex
           | Star Regex
           | Range String
           | Cat Regex Regex
           | Alt Regex Regex
```

in cui `Symbol`, `Opt`, `Star`, `Range`, `Cat` ed `Alt` si riferiscono, rispettivamente, ad un carattere dell'alfabeto, l'opzionalità, la ripetizione zero o più volte, un range di caratteri, la concatenazione e l'alternativa. Si chiede di definire la funzione `mostra` (protocollo incluso) che, ricevendo in ingresso una espressione regolare, genera la stringa di testo che la rappresenta.

```
mostra :: Regex -> String
mostra (Symbol c) = [c]
mostra (Opt x) = (mostra x)++"?"
mostra (Star x) = (mostra x)++"*"
mostra (Range s) = "["++s++"]"
mostra (Cat x y) = "("++(mostra x)++(mostra y)++")"
mostra (Alt x y) = "("++(mostra x)++"|"++(mostra y)++")"
```

Esercizio 45

Specificare in *Haskell* la classe di tipi `Expr`, nella quale è definito l'operatore binario `(#)`, che rappresenta l'elevamento a potenza (quest'ultima rappresentata sempre da un intero $p \geq 0$). Quindi, istanziare la classe `Expr` mediante i seguenti tipi:

- `Int`: in cui `(#)` è la classica potenza a base intera;
- `String`: in cui `(#)` rappresenta la ripetizione p volte della stringa.

Esercizio 45

Specificare in *Haskell* la classe di tipi `Expr`, nella quale è definito l'operatore binario `(#)`, che rappresenta l'elevamento a potenza (quest'ultima rappresentata sempre da un intero $p \geq 0$). Quindi, istanziare la classe `Expr` mediante i seguenti tipi:

- `Int`: in cui `(#)` è la classica potenza a base intera;
- `String`: in cui `(#)` rappresenta la ripetizione p volte della stringa.

```
class Expr a where
  (#) :: a -> Int -> a

instance Expr Int where
  n # 0 = 1
  n # p = n * (n # (p-1))

instance Expr String where
  s # 0 = []
  s # p = s ++ (s # (p-1))
```

Esercizio 46

Dopo aver illustrato il significato della forma funzionale `foldr` in *Haskell*, sulla base della seguente definizione:

```
genera :: [Int] -> Int
genera = foldr (\x y -> x + 2 * y) 1
```

indicare l'espressione aritmetica computata dalla seguente applicazione ed il corrispondente risultato:

```
genera [1,2,3]
```

Esercizio 46

Dopo aver illustrato il significato della forma funzionale `foldr` in *Haskell*, sulla base della seguente definizione:

```
genera :: [Int] -> Int
genera = foldr (\x y -> x + 2 * y) 1
```

indicare l'espressione aritmetica computata dalla seguente applicazione ed il corrispondente risultato:

```
genera [1,2,3]
```

```
1 + 2 * (2 + 2 * (3 + (2 * 1))) = 25
```

Esercizio 47

Specificare nel linguaggio *Haskell* la funzione `separa`, avente in ingresso una lista (anche vuota) di triple, la quale computa la corrispondente tripla di liste, come nei seguenti esempi:

```
separa [(1, 'a', True)] = ([1], ['a'], [True])

separa [(1, 'a', True), (2, 'b', False),(3, 'c', True)] =
       ([1, 2, 3], ['a', 'b', 'c'], [True, False, True])
```

Esercizio 47

Specificare nel linguaggio *Haskell* la funzione `separa`, avente in ingresso una lista (anche vuota) di triple, la quale computa la corrispondente tripla di liste, come nei seguenti esempi:

```
separa [(1, 'a', True)] = ([1], ['a'], [True])

separa [(1, 'a', True), (2, 'b', False),(3, 'c', True)] =
       ([1, 2, 3], ['a', 'b', 'c'], [True, False, True])
```

```
separa :: [(a,b,c)] -> ([a],[b],[c])
separa lista = ([x | (x,_,_) <- lista], [y | (_,y,_) <- lista], [z | (_,_,z) <- lista])
```

oppure:

```
separa [] = ([],[],[])
separa ((x,y,z):coda) = (x:listax, y:listay, z:listaz)
  where (listax, listay, listaz) = separa coda
```

Esercizio 48

Dopo aver illustrato il significato della forma funzionale `foldr` in *Haskell*, sulla base della seguente definizione,

```
computa :: [[a]] -> [a]
computa = foldr (\x y -> init (x++y)) []
```

indicare sia l'espressione computata dalla seguente applicazione che il corrispondente risultato:

```
computa ["alfa", "beta", "gamma"]
```

Esercizio 48

Dopo aver illustrato il significato della forma funzionale `foldr` in *Haskell*, sulla base della seguente definizione,

```
computa :: [[a]] -> [a]
computa = foldr (\x y -> init (x++y)) []
```

indicare sia l'espressione computata dalla seguente applicazione che il corrispondente risultato:

```
computa ["alfa", "beta", "gamma"]
```

```
computa ["alfa", "beta", "gamma"] =
init ("alfa" ++ init ("beta" ++ (init ("gamma" ++ [])))) = "alfabetaga"
```