```
type Ponto = (Float, Float)                -- (abcissa, ordenada)
type Rectangulo = (Ponto, Float, Float)    -- (canto sup. esq., larg, alt)
type Triangulo = (Ponto, Ponto, Ponto)
type Poligonal = [Ponto]

distancia :: Ponto -> Ponto -> Float
distancia (a,b) (c,d) = sqrt (((c - a) ^ 2) + ((b - d) ^ 2))

poli :: Poligonal
poli = [(1,1),(1,3),(4,3),(4,1),(1,1)]
```

1. Considere as seguintes definições.

```
type Ponto = (Float,Float)                -- (abcissa,ordenada)
type Rectangulo = (Ponto,Float,Float)     -- (canto sup.esq., larg, alt)
type Triangulo = (Ponto,Ponto,Ponto)
type Poligonal = [Ponto]

distancia :: Ponto -> Ponto -> Float
distancia (a,b) (c,d) = sqrt (((c-a)^2) + ((b-d)^2))
```

(a) Defina uma funçao que calcule o comprimento de uma linha poligonal.

```
compPoligonal :: Poligonal -> Float
compPoligonal (x:y:xys) = (distancia x y) + (compPoligonal (y:xys))
compPoligonal _ = 0
```

(b) Defina uma funçao que converta um elemento do tipo `Triangulo` na correspondente linha poligonal.

```
triToPoli :: Triangulo -> Poligonal
triToPoli (x,y,z) = [x,y,z,x]
```

(c) Repita o alínea anterior para elementos do tipo `Rectangulo`.

```
rectToPoli :: Rectangulo -> Poligonal
rectToPoli ((x,y), ab, ord) = [(x,y), (x + ab, y), (x + ab, y - ord), (x, y - ord), (x, y)]
```

(d) Defina uma funçao `fechada` que testa se uma dada linha poligonal é ou nao fechada.

```
fechada :: Poligonal -> Bool
fechada [x, y] = x == y
fechada (x:y:ts) = fechada (x:ts)
```

(e) Defina uma funcao `triangula` que, dada uma linha poligonal fechada e convexa, calcule uma lista de triangulos cuja soma das areas seja igual à area delimitada pela linha poligonal.

```
triangula :: Poligonal -> [Triangulo]
triangula [x,y,w,z] = [(x,y,w)]
triangula (x:y:w:z:ts) = (x,y,w) : (triangula (x:w:z:ts))
triangula _ = [] -- para [], [x], [x,y], [x,y,z];
```

(f) Suponha que existe uma função `areaTriangulo` que calcula a area de um triangulo.

```
areaTriangulo (x,y,z)
   = let a = distancia x y
         b = distancia y z
         c = distancia z x
         s = (a+b+c) / 2 -- semi-perimetro
     in -- formula de Heron
        sqrt (s*(s-a)*(s-b)*(s-c))
```

Usando essa funcao, defina uma função que calcule a area delimitada por uma linha poligonal fechada e convexa.

```
areaTriangulo :: Triangulo -> Float
areaTriangulo (x,y,z) = sqrt (s * (s - a) * (s - b) * (s - c)) where
                          a = distancia x y
                          b = distancia y z
                          c = distancia z x
                          s = (a + b + c) / 2
-- formula de Heron

areaPoligonal :: Poligonal -> Float
areaPoligonal lista = aPAux (triangula lista) where
  aPAux [] = 0
  aPAux (h:ts) = (areaTriangulo h) + (aPAux ts)
```

(g) Defina uma funcao `mover` que, dada uma linha poligonal e um ponto, dá como resultado uma linha poligonal idêntica à primeira mas tendo como ponto inicial o ponto dado. Por exemplo, ao mover o triangulo `[(1,1),(10,10),(10,1),(1,1)]` para o ponto (1,2) devemos obter o triângulo `[(1,2),(10,11),(10,2),(1,2)]`.

```
mover :: Ponto -> Poligonal -> Poligonal
mover _ [] = []
mover (a,b) ((x,y):ts) = mAux (a - x,b - y) ((x,y):ts) where
    mAux _ [] = []
    mAux (ab,od) ((x,y):ts) = (x + ab,y + od) : (mAux (ab,od) ts)
```

(h) Defina uma funçao `zoom2` que, dada uma linha poligonal, dê como resultado uma linha poligonal semelhante e com o mesmo ponto inicial mas em que o comprimento de cada segmento de recta é multiplicado por 2. Por exemplo, o rectângulo

`[(1,1),(1,3),(4,3),(4,1),(1,1)]`

dever ser transformado em `[(1,1),(1,5),(7,5),(7,1),(1,1)]`

```
zoom2 :: Poligonal -> Poligonal
zoom2 [] = []
zoom2 [x] = [x]
zoom2 (x:y:t) = x:(zoomAux x (y:t))
  where zoomAux :: Ponto -> Poligonal -> Poligonal
        zoomAux _ [] = []
        zoomAux x (y:t) = ((\(a,b) (c,d) -> (a+c,b+d))((\(a,b) c ->
(a*c,b*c))((\(a,b) (c,d) -> (a-c,b-d))y x) 2) x):(zoomAux x t)
```

2. Considere as seguintes definições de tipos para representar uma tabela de registo de temperaturas.

```
type TabTemp = [(Data,Temp,Temp)]   -- (data, temp. mínima, temp. maxima)
type Data = (Int,Int,Int)           -- (ano, mes, dia)
type Temp = Float
```

   (a) Defina a funcão `medias :: TabTemp -> [(Data,Temp)]` que constroi a lista com as temperaturas médias de cada dia.

```
medias :: TabTemp -> [(Data,Temp)]
medias [(dt,tp1,tp2)] = [(dt,((tp1 + tp2) / 2))]
medias ((dt,tp1,tp2):ts) = (dt,((tp1 + tp2) / 2)) : (medias ts)
```

   (b) Defina a funçao `decrescente :: TabTemp -> Bool` que testa se a tabela está ordenada por ordem decrescente de data. (Nota: pode usar o operador > para comparar directamente duas datas.)

```
decrescente :: TabTemp -> Bool
decrescente (x:y:ts) = (x > y) && (decrescente (y:ts))
decrescente _ = True -- para [] e [x];
```

   (b) Defina a funçao `conta :: [Data] -> TabTemp -> Int` que, dada uma lista de datas e a tabela de registo de temperaturas, conta quantas das datas da lista têm registo de na tabela.

```
conta :: [Data] -> TabTemp -> Int
conta [] _ = 0
conta (x:xs) list = if (apAux x list) then 1 + (conta xs list) else 0 + (conta xs list)
where
 apAux _ [] = False
 apAux x ((dt,_,_):ts) = (x == dt) || (apAux x ts)
```

3. Um multi-conjunto é um conjunto que admite elementos repetidos. É diferente de uma lista porque a ordem dos elementos nao é relevante. Uma forma de implementar multi-conjuntos em Haskell é através de uma lista de pares, onde cada par regista um elemento e o respectivo número de ocorrências:

```
type MSet a = [(a,Int)]
```

   Uma lista que representa um multi-conjunto nao deve ter mais do que um par a contabilizar o número de ocorrências de um elemento, e o número de ocorrências deve ser sempre estritamente positivo. O multi-conjunto de caracteres `{'b','a','c','a','b','a'}` poderia, por exemplo, ser representado pela lista `[('b',2),('a',3),('c',1)]`.

   (a) Defina a funçao `union :: Eq a => MSet a -> MSet a -> MSet a` que calcula a uniao de dois multi-conjuntos. Por exemplo,

```
> union [('a',3),('b',2),('c',1)] [('d',5),('b',1)]
[('a',3),('b',3),('c',1),('d',5)]
```

```
union :: (Eq a) => MSet a -> MSet a -> MSet a

union lista [] = lista

union lista ((a,b):ts) = union (iMSetAux (a,b) lista) ts where

 iMSetAux (w,z) [] = [(w,z)]

 iMSetAux (w,z) ((x,y):ts) = if (w == x) then (x,(y + z)) : ts else
(x,y) : (iMSetAux (w,z) ts)
```

(b) Defina a funcão `intersect :: Eq a => MSet a -> MSet a -> MSet a` que calcula a intersecçao de dois multi-conjuntos. Por exemplo,

```
> intersect [('a',3),('b',5),('c',1)] [('d',5),('b',2)]
[('b',2)]
```

```
intersect :: (Eq a) => MSet a -> MSet a -> MSet a
intersect [] _ = []
intersect _ [] = []
intersect (h:ts) list = let result = intAux h list in
 if (snd(result) == -1) then intersect ts list else result : intersect ts
list where
        intAux (x,y) [] = (x,-1)
        intAux (x,y) ((a,b):ts) = if (x == a) then if (y < b) then (x,y)
else (x,b)

          else intAux (x,y) ts
```

(c) Defina a funçao `diff :: Eq a => MSet a -> MSet a -> MSet a` que calcula a diferença de dois multi-conjuntos. Por exemplo,

```
> diff [('a',3),('b',5),('c',1)] [('d',5),('b',2)]
[('a',3),('b',3)],('c',1)]
```

```
diff :: (Eq a) => MSet a -> MSet a -> MSet a
diff [] list = list
diff list [] = list
diff (h:ts) list = let result = difAux h list [] in
 if ((snd(fst result)) == 0) then diff ts (snd result) else (fst
result) : diff ts (snd result) where
        difAux (x,y) [] new = ((x,y),new)
        difAux (x,y) ((a,b):ts) new = if (x == a) then if (y > b)
then ((x,(y - b)), new ++ ts) else ((x,(b - y)), new ++ ts)

                                  else difAux (x,y) ts (new ++
[(a,b)])
```

(d) Defina a funçao `ordena :: MSet a -> MSet a` que ordena um multi-conjunto pelo número crescente de ocorrências. Por exemplo,

```
> ordena [('b',2),('a',3),('c',1)]
[('c',1),('b',2),('a',3)]
```

```
ordena :: MSet a -> MSet a
ordena [] = []
ordena list = ordAux list [] where
 ordAux [] new = new
 ordAux (h:ts) new = ordAux ts
(iOrdAux h new) where
        iOrdAux el [] = [el]
        iOrdAux (x,y) ((a,b):ts) = if (y >
b) then (a,b) : (iOrdAux (x,y) ts) else (x,y)
: (a,b) : ts
```

(e) Defina a funçao `moda :: MSet a -> [a]` que devolve a lista dos elementos com maior número de ocorrências. Por exemplo,

```
> moda [('b',2),('a',3),('c',1),('d',3)]
['a','d']
```

```
moda :: MSet a -> [a]

moda [] = []

moda ((x,y):ts) = mAux ts y [x] where

 mAux [] _ new = new

 mAux ((x,y):ts) hg new = if (hg > y) then mAux
ts hg new

                                  else if (hg == y) then
mAux ts hg (new ++ [x])


                 else mAux ts y [x]
```