```
module FichaPratica2 where

import Char

--1)
iLower :: Char -> Bool
iLower x = if (ord x >= 97 && ord x<=122) then True else
False

iDigit :: Char->Bool
iDigit x = if (ord x >= ord '0' && ord x <= ord '9') then
True else False

iAlpha :: Char->Bool
iAlpha x = (isUpper x) || (isLower x)

tUpper :: Char->Char
tUpper x = if (ord x >= ord 'a' && ord x <= ord 'z')
      then chr (ord x - 32)
      else x

--2)
max2 :: Int->Int->Int
max2 x y = if (x>=y) then x else y

--3)
max3 :: Int->Int->Int->Int
max3 x y z = if (x>=y && x>=z)
        then x
        else if (y>=x && y>=z)
        then y
        else z
--max3 x y z = max2 (max2 x y) z


--4)
etriangulo :: (Float,Float,Float)->Bool
etriangulo (a,b,c) = a+b>=c && a+c>=b && b+c>=a && a>0
&& b>0 && c>0


--5)
opp :: (Int, (Int,Int)) -> Int
opp (1,(y,z)) = y+z
```

```
opp (2,(y,z)) = y-z
opp (_,(y,z)) = 0


--6)
nraiz :: Float->Float->Float->Int
nraiz a b c = if delta<0
         then 0
         else if delta==0
         then 1
         else 2
   where delta = b^2 - 4*a*c


--7)
raizes :: Double->Double->Double->[Double]
raizes a b c = if delta <0
          then []
          else if delta==0
          then [-b/(2*a)]
          else [(-b + sqrt delta)/(2*a), (-b - sqrt
delta)/(2*a)]
   where delta = b^2 - 4*a*c


--8)
tnraiz :: (Float,Float,Float)->Int
tnraiz (a,b,c) = if delta<0
         then 0
         else if delta==0
         then 1
         else 2
   where delta = b^2 - 4*a*c

traizes :: (Double,Double,Double)->[Double]
traizes (a,b,c) = if delta <0
          then []
          else if delta==0
          then [-b/(2*a)]
          else [(-b + sqrt delta)/(2*a), (-b - sqrt
delta)/(2*a)]
   where delta = b^2 - 4*a*c

module Ficha3 where
```

```
type Time = (Int,Int)

--1a)
horavalida :: Time->Bool
horavalida (h,m) = h>=0 && h<24 && m>=0 && m<=59

--1b)
horacompar :: (Time,Time)->Bool
horacompar ((h1,m1),(h2,m2)) = (h1>h2 || (h1 == h2 &&
m1>m2)) && (horavalida (h1,m1) && horavalida(h2,m2))

--1c)
horamin :: Time->Int
horamin (h,m) = h*60+m

--1d)
minhora :: Int->Time
minhora m = (div m 60, mod m 60)

--1e)
difhora :: (Time,Time)->Int
difhora ((h1,m1),(h2,m2)) =
abs(horamin(h1,m1)-horamin(h2,m2))

--2)
netapas :: [(Time,Time)]->Int
netapas [] = 0
netapas (x:xs) = 1 + netapas xs

--3)
hPartida :: [(Time,Time)]->Time
hPartida l = fst (head l)

hchegada :: [(Time,Time)]->Time
hchegada l = snd (last l)

--4)
etapavalida :: (Time,Time)->Bool
etapavalida ((h1,m1),(h2,m2)) = horacompar
((h2,m2),(h1,m1))

--5)
viagemvalida :: [(Time,Time)]->Bool
```

```
viagemvalida ((p1,c1):(p2,c2):t) = etapavalida (p1,c1)
&& horacompar (p2,c1) && viagemvalida ((p2,c2):t)
viagemvalida [e] = etapavalida e

--6)
tempoViagem::[(Time,Time)]->Int
tempoViagem [] = 0
tempoViagem ((p,c):xs) = if viagemvalida ((p,c):xs)
          then (difhora (c,p)) + (tempoViagem xs)
          else 0

--7)
tempoEspera::[(Time,Time)]->Int
tempoEspera [] = 0
tempoEspera ((p,c):xs) = if viagemvalida ((p,c):xs)
          then (difhora (snd (last xs),p)) - tempoViagem
((p,c):xs)
          else 0

--8)
tempoTotal::[(Time,Time)]->Int
tempoTotal [] = 0
tempoTotal ((p,c):xs) = if viagemvalida ((p,c):xs)
          then difhora (snd (last xs),p)
          else 0

module Stocks where

type Stock = [(Produto,Preco,Quantidade)]
type Produto = String
type Preco = Float
type Quantidade = Float

--1 a)
emStock::Produto->Stock->Quantidade
emStock x [] = 0
emStock x ((p,s,q):hs) |x==p = q
             |otherwise = emStock x hs
--Stocks> emStock "bbb" [("aba",10,3.5),("bbb",75,9)]
--9.0

--1 b)
consulta::Produto->Stock->(Preco,Quantidade)
consulta x [] = (0,0)
```

```
consulta x ((a,b,c):ls) = if x==a
           then (b,c)
           else consulta x ls

--1 c)
tabPrecos::Stock->[(Produto,Preco)]
tabPrecos [] = []
tabPrecos ((a,b,c):ls) = (a,b):tabPrecos ls

--1 d)
valorTotal::Stock->Float
valorTotal [] = 0
valorTotal ((a,b,c):ls) = b*c+valorTotal ls

--1 e)
inflacao::Float->Stock->Stock
inflacao x [] = []
inflacao x ((a,b,c):ls) = let k = x*b+b
               in (a,k,c):inflacao x ls

--1 f)
omaisBarato::Stock->(Produto,Preco)
omaisBarato [] = ("Nenhum",0)
omaisBarato [(a,b,c)] = (a,b)
omaisBarato ((a,b,c):t) = let (r,p) = omaisBarato t
           in if (p>b)
              then (a,b)
              else (r,p)

--1 g)
maisCaros::Preco->Stock->[Produto]
maisCaros x [] = []
maisCaros x ((a,b,c):ls) = if b>x
            then a:maisCaros x ls
            else maisCaros x ls

--2
type ListaCompras = [(Produto,Quantidade)]

--2 a)
verifLista::ListaCompras->Stock->Bool
verifLista [] s = True
verifLista ((p,q):xs) s = if (emStock p s) >=q then
verifLista xs s
```

```
                    else False
--2 b)
falhas::ListaCompras->Stock->ListaCompras
falhas [] s = []
falhas ((p,q):ls) s = if (emStock p s) >=q then falhas
ls s
                 else (p,q-emStock p s):falhas ls s

--2c)
custoTotal::ListaCompras->Stock->Float
custoTotal [] s = 0
custoTotal ((x,y):t) s = let (p,q) = consulta x s
          in (min y q)*p + custoTotal t s

--2d)
partePreco::Preco->ListaCompras->Stock->(ListaCompras
,ListaCompras)
partePreco p ((x,y):t) s = let (l1,l2) = partePreco p t
s
               (a,_) = consulta x s
            in if a<p then ((x,y):l1,l2)
                else (l1,(x,y):l2)
partePreco p [] s = ([],[])

module Ficha5 where

--1a)
div1::Int->Int->Int
div1 x y |x>=y =  1 +div1 (x-y) y
         | otherwise = 0

mod1::Int->Int->Int
mod1 x y |x>=y = mod1 (x-y) y
         | otherwise = x

--1b)
divMod1::Int->Int->(Int,Int)
divMod1 x y |x>=y = let (a,b) = (divMod1 (x-y) y)
          in (a+1,b)
       |otherwise =(0,x)

--2)
splitAt1::Int->[a]->([a],[a])
splitAt1 x []=([],[])
```

```
splitAt1 x l | x<=0 = ([],l)
splitAt1 x (l:ls)= let(a,b) = splitAt1 (x-1) ls
         in (l:a,b)

--3a)
lines1::String->[String]
lines1 [] = []
lines1 x = let (a,b) = linha x
      in a:lines1 b

linha::String->(String,String)
linha [] = ([],[])
linha (x:xs) |x=='\n' = ([],xs)
        |otherwise = let (a,b) = linha xs
           in (x:a,b)

unlines1::[String]->String
unlines1 [] = []
unlines1 (x:xs) = x++enter:unlines xs
   where enter = '\n'


--3b)
words1::String->[String]
words1 []= []
words1 x = let (a,b) = palavras x
      in (a:words1 b)


palavras::String->(String,String)
palavras [] = ([],[])
palavras (x:xs) |x==' ' = ([],ajuda xs)
      |x=='\n' = ([],xs)
      |otherwise = let (a,b) = palavras xs
              in (x:a,b)

ajuda::String->String
ajuda [] = []
ajuda (x:xs) |x==' ' = ajuda xs
        |otherwise = (x:xs)

unwords1::[String]->String
unwords1 [] = []
unwords1 (x:xs) = x++' ':unwords xs
```

```
--4
separa::String->[String]
separa [] = []
separa x = let (a,b) = ponto x
      in (a: separa b)

ponto::String->(String,String)
ponto [] = ([],[])
ponto (x:xs) |x=='.' = (['.'],xs)
        |otherwise = let (a,b) = ponto xs
           in (x:a,b)


module Ficha6 where

type Stock = [(Produto,Preco,Quantidade)]
type Produto = String
type Preco = Float
type Quantidade = Float
type ListaCompras = [(Produto,Quantidade)]

--1)
takeWhile1::(a->Bool)->[a]->[a]
takeWhile1 f (x:xs) | f x = x:takeWhile1 f xs
          |otherwise = []
takeWhile1 _ [] = []

dropWhile1::(a->Bool)->[a]->[a]
dropWhile1 f (x:xs) | f x = dropWhile1 f xs
          |otherwise = x:xs
dropWhile1 _ [] = []

--2)
break1::(a->Bool)->[a]->([a],[a])
break1 f [] = ([],[])
break1 f (x:xs) |f x = let (a,b) = break1 f xs
         in (x:a,b)
      |otherwise = ([],x:xs)

--3)
emStock::Produto->Stock->Quantidade --so esta aqui
porque e precisa para a funçao "falhas"
```

```
emStock x [] = 0
emStock x ((p,s,q):hs) |x==p = q
             |otherwise = emStock x hs

tabPrecos::Stock->[(Produto,Preco)]
tabPrecos s = map tabela s
   where tabela::(String,Float,Float)->(String,Float)
         tabela (x,y,z) = (x,y)

inflacao::Float->Stock->Stock
inflacao x s = map (\ (p,pr,q)->(p,pr*x+pr,q)) s

falhas::Stock->ListaCompras->ListaCompras
--falhas s l = filter (\(a,b)->(emStock a s) >= b) l
--esta era a maneira de fazer com filter
falhas s l = foldr aux [] l
   where
aux::(Produto,Quantidade)->ListaCompras->ListaCompras
         aux (p,q) r = if (emStock p s) >=q
             then r
             else (p,q):r
--maneira de fazer com foldr


-- 4)
indica :: String -> [String] -> [String]
indica [] telefs = telefs
indica ind [] = []
indica ind (y:ys) | concorda ind y = y : indica ind ys
                  | otherwise = indica ind ys
   where concorda :: String -> String -> Bool
         concorda [] _ = True
         concorda (x:xs) (y:ys) = (x==y) && (concorda xs
ys)
         concorda (x:xs) [] = False

odule Ficha7 where

type Bit = Bool

bitToInt::Bit->Int
bitToInt False =0
bitToInt True=1
```

```
intToBit::Int->Bit
intToBit 0 = False
intToBit 1 = True


bListToInt::[Bit]->Int
bListToInt l = aux 0 l

aux::Int->[Bit]->Int
aux e [] = 0
aux e (x:xs) = (bitToInt x) * 2^e + aux (e+1) xs

intTobList :: Int -> [Bit]
intTobList 0 = []
intTobList x =((intToBit(mod x 2)):(intTobList(div x
2)))

tabuada :: Bit -> Bit -> Bit -> (Bit, Bit) -- (res, carry)
tabuada False False False = (False, False)
tabuada False False True = (True, False)
tabuada False True False = (True, False)
tabuada True False False = (True, False)
tabuada False True True = (False, True )
tabuada True False True = (False, True )
tabuada True True False = (False, True )
tabuada True True True = (True, True )

soma::[Bit]->[Bit]->[Bit]
soma l1 l2 = somaAux False l1 l2

somaAux::Bit->[Bit]->[Bit]->[Bit]
somaAux False l [] = l
somaAux False [] l = l
somaAux True [] [] = [True]
somaAux True [] (x:xs) = let (a,b) = tabuada True False
x
          in a: (somaAux b [] xs)
somaAux True (x:xs) [] = let (a,b) = tabuada True x False
          in a:(somaAux b [] xs)
somaAux c (x:xs) (y:ys) = let (a,b) = tabuada c x y
           in a:(somaAux b xs ys)

multiplica::[Bit]->[Bit]->[Bit]
multiplica [] []= []
```

```
multiplica l [] = []
multiplica [] l = []
multiplica l1 (x:xs) = let k = multiplicaAux x l1
            j = False:(multiplica l1 xs)
             in soma k j

multiplicaAux::Bit->[Bit]->[Bit]
multiplicaAux l [] = []
multiplicaAux b (x:xs) = intToBit ((bitToInt
b)*(bitToInt x)):multiplicaAux b xs


mult :: Int -> Int -> Int
mult x y = let bx = intTobList x
           by = intTobList y
           br = multiplica bx by
      in bListToInt br

module Ficha8 where

type Polinomio = [Coeficiente]
type Coeficiente = Int
type Bit = Bool

--1)
somaPol::Polinomio->Polinomio->Polinomio
somaPol [] [] = []
somaPol l [] = l
somaPol [] l = l
somaPol (x:xs) (y:ys) = (x+y):somaPol xs ys

multPol::Polinomio->Polinomio->Polinomio
multPol _ [] = []
multPol [] l = []
multPol (x:xs) y = somaPol (map (x*) y) (0:multPol xs y)

--2)
normaliza::Int->Polinomio->Polinomio
normaliza b [] = []
normaliza b (x:xs) = let (q,r) = divMod x b
           in r:normaliza b (junta q xs)

junta::Int->Polinomio->Polinomio
junta 0 [] = []
```

```
junta x []  = [x]
junta b (c:cs) = (b+c):cs

--3)
somaBase::Int->Polinomio->Polinomio->Polinomio
somaBase b p1 p2 = normaliza b (somaPol p1 p2)

multBase::Int->Polinomio->Polinomio->Polinomio
multBase b p1 p2 = normaliza b (multPol p1 p2)

--4)
multBits::[Bit]->[Bit]->[Bit]
multBits a b = map intToBit (multBase 2 (map bitToInt a)
(map bitToInt b))

somaBits::[Bit]->[Bit]->[Bit]
somaBits a b = map intToBit (somaBase 2 (map bitToInt a)
(map bitToInt b))

bitToInt::Bit->Int
bitToInt False =0
bitToInt True=1

intToBit::Int->Bit
intToBit 0 = False
intToBit 1 = True


module Ficha9 where

data ExpInt = Const Int
       | Simetrico ExpInt
       | Mais ExpInt ExpInt
       | Menos ExpInt ExpInt
       | Mult ExpInt ExpInt

type ExpN = [Parcela]
type Parcela = [Int]

--1 a)
calcula::ExpInt->Int
calcula (Const x) = x
calcula (Simetrico x)= - (calcula x)
calcula (Mais x y) = (calcula x) + (calcula y)
```

```
calcula (Menos x y) = (calcula x) - (calcula y)
calcula (Mult x y) = (calcula x)*(calcula y)


--1 b)
expString::ExpInt->String
expString (Const x) = show x
expString (Simetrico x) = "-" ++ "(" ++ (expString x) ++
")"
expString (Mais x y) = "("++(expString x) ++"+" ++
(expString y) ++")"
expString (Menos x y) = "("++(expString x) ++"-" ++
(expString y) ++")"
expString (Mult x y) = "("++(expString x) ++"*" ++
(expString y) ++")"

--1 c)
posfix::ExpInt->String
posfix (Const x) = show x
posfix (Simetrico x) = (posfix x) ++ " " ++ "-"
posfix (Mais x y) = (posfix x) ++" " ++ (posfix y) ++"
"++ "+"
posfix (Menos x y) = (posfix x) ++" " ++ (posfix y) ++"
"++ "-"
posfix (Mult x y) = (posfix x) ++" " ++ (posfix y) ++"
"++ "*"

--2 a)
calcN::ExpN->Int
calcN = sum.(map product)

--2 b)
normaliza::ExpInt->ExpN
normaliza (Const x) = [[x]]
normaliza (Simetrico x) = map (-1:) (normaliza x)
normaliza (Mais x y) = (normaliza x)++(normaliza y)
normaliza (Menos x y) = normaliza (Mais x (Simetrico y))
normaliza (Mult x y) = let l1 = normaliza x
            l2 = normaliza y
             in multiplica l1 l2

multiplica::ExpN->ExpN->ExpN
multiplica [] l = []
multiplica (p:ps) l = (map (p++) l)++multiplica ps l
```

```
--2 c)
expNString::ExpN->String
expNString [x] = aplicaVezes x
expNString (p:ps) = aplicaVezes p ++ "+" ++ expNString
ps

aplicaVezes::Parcela->String
aplicaVezes [x] = show x
aplivaVezes (x:xs) = show x ++ "*" ++ (aplicaVezes xs)
```