PROGRAMAÇÃO FUNCIONAL

String == [Char]

init → retira o último

tail → retira o primeiro

FUNÇÃO REVERSE COM ACUMULADOR (+ EFICIENTE)

```
reverse :: [a] → [a]
reverse l = reverseAc [] l
    where reverseAc ac [] = ac
          reverseAc ac (x:xs) =
              reverseAc (x:ac) xs
```

FUNÇÃO MAP

↳ aplicar uma função a cada elemento de uma lista, gerando deste modo uma nova lista

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

FUNÇÃO FILTER

↳ dada uma lista, gera uma nova lista com os elementos da lista que satisfazem um determinado predicado (mantém os elementos da lista para os quais o predicado é verdadeiro).

```
filter :: (a → Bool) → [a] → [a]
filter p [] = []
filter p (x:xs)
    | (p x) == True = x : filter p xs
    | otherwise = filter p xs
```

FUNÇÕES ANÓNIMAS

EXEMPLO:

```
trocaPares xs = map troca xs
    where troca (x,y) = (y,x)
```

```
trocaPares xs = map (\(x,y) = (y,x)) xs
```

FOLDR (associa à direita)

↳ aplica um operador binário ao primeiro elemento da lista e ao resultado de aplicar a função ao resto da lista

EXEMPLO:

product xs = foldr (*) (1) xs

↳ product [4,3,5] = 4*(3*(5*(1)) => 60

FOLDL (associa à esquerda)

↳ semelhante ao foldr, mas associa à esquerda.

EXEMPLO:

sum xs = foldl (+) (0) xs

↳ sum [1,2,3] = (((0)+1)+2)+3 => 6

OUTRAS FUNÇÕES DE ORDEM SUPERIOR:

- FUNÇÃO CURRY

curry :: ((a,b) → c) → a → b → c

curry f x y = f (x,y)

• FUNÇÃO UNCURRY

```
uncurry :: (a → b → c) → (a, b) → c
uncurry f (x,y) = f x y
```

exemplo

```
quocientes pares = map (uncurry div) pares
```

↳ quocientes [(3,4), (23,5), (7,3)] ⇒ [0, 4, 2]

• FUNÇÃO FLIP

```
flip :: (a → b → c) → b → a → c
flip f x y = f y x
```

exemplo

```
reverse xs = foldl (flip (:)) [] xs
```

FUNÇÃO QUE SOMA 2 VETORES

```
data CCart : Coord Float Float
      "deriving Show"

somaVect :: CCart → CCart → CCart
somaVect (Coord x1 y1) (Coord x2 y2) =
      Coord (x1 + x2) (y1 + y2)
```

FUNÇÃO QUE TRANSFORMA HORAS EM MINUTOS

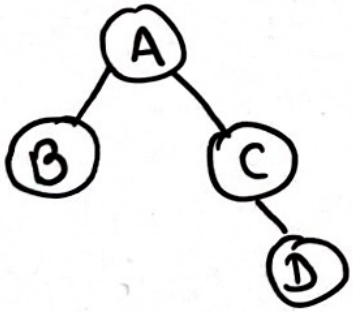
```
data Hora = AM INT INT
          | PM INT INT
  " deriving Show "

totalminutos :: Hora → Int
totalminutos (AM h m) = (h*60) + m

totalminutos (PM h m) =
    (12*60) + (h*60) + m
```

ÁRVORES BINÁRIAS

↳ TERMINOLOGIA



- O nodo A é a raiz da árvore
- Os nodos B e C são filhos / descendentes de A
- O nodo C é pai de D.
- O caminho / path de um nodo é a sequência de nodos da raiz até esse nodo
- A altura é o comprimento do caminho + longo.

↳ FUNÇÕES

```
altura :: ArvBin a → Integer
altura Vazia = 0
altura (Nodo _ e d) =
    1 + max (altura e) (altura d)

mapAB :: (a → b) → ArvBin a → ArvBin b
mapAB f Vazia = Vazia
mapAB f (Nodo x e d) =
    Nodo (f x) (mapAB f e) (mapAB f d)

unzipAB :: ArvBin (a,b) → (ArvBin a, ArvBin b)
unzipAB Vazia = (Vazia, Vazia)
unzipAB (Nodo (x,y) e d) =
    let (e1, e2) = unzipAB e
        (d1, d2) = unzipAB d
    in (Nodo x e1 d1, Nodo y e2 d2)
```

↳ CONVERTER UMA ÁRVORE BINÁRIA NUMA LISTA

Seja:

R → visitar Raiz

→

E → atravessar a sub - árvore da esquerda

D → atravessar a sub - árvore da direita

PREORDER : RED

```
preorder :: ArvBin a → [a]
preorder Vazia = []
preorder (NNode x e d) =
  [x] ++ (preorder e) ++ (preorder d)
```

INORDER : ERD

```
inorder :: ArvBin a → [a]
inorder Vazia = []
inorder (NNode x e d) =
  (inorder e) ++ [x] ++ (inorder d)
```

POSTORDER : EDR

```
postorder :: ArvBin
postorder Vazia = []
postorder (NNode x e d) =
  (postorder e) ++ (postorder d) ++ [x]
```

ÁRVORES BINÁRIAS DE PROCURA

Nesta árvore:

- a raiz é maior do que todos os elementos da sub-árvore esquerda
- a raiz é menor do que todos os elementos da sub-árvore direita
- ambas as sub-árvores são árvores binárias de procura

↳ FUNÇÃO QUE ACRESCENTA UM ELEMENTO À ÁRVORE BINÁRIA DE PROCURA

```
insABproc :: a → ArvBin a → ArvBin a
insABproc x Vazia = (Nodo x Vazia Vazia)
insABproc x (Nodo y e d)
    | x < y = Nodo y (insABproc x e) d
    | x == y = Nodo y e d
    | otherwise = Nodo y e (insABproc x d)
```

ÁRVORES IRREGULARES

```
data Tree a = Node a [Tree a]
```

CLASSES

TIPOS

- Num : Int, Integer, Float, Double, ...

```
(+) :: Num a => a -> a -> a
```

lê-se "para todo o tipo a que pertence à classe Num, (+) tem tipo a → a → a"

NOTA: os tipos que pertencem a uma classe também serão chamados de instâncias de classe.

Eq → tipos para os quais existe uma operação de igualdade

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Ord → herda todos os métodos de Eq e, além disso, estabelece um conjunto de operações de comparação e as funções máximo e mínimo.

→

class (Eq a) => Ord a where

(<), (<=), (>=), (>) :: a → a → Bool

max, min :: a → a → a

⚠ todo o tipo que é instância de Ord tem necessariamente que ser instância de Eq.

(data Ordering = LT | GT | EQ)

Show → estabelece métodos para converter um valor de um tipo qualquer (que lhe pertença) numa string.

O interpretador Haskell usa o método show para apresentar o resultado dos seus cálculos.

Num → desenhada para controlar as operações que devem estar definidas sobre os ≠ tipos de nºs.

Os tipos Int, Integer, Float e Double são instâncias desta classe.

⚠ succ → função que retorna o elemento seguinte

succ 2 = 3

succ a = b

Enum → estabelece um conjunto de operações que permitem sequências aritméticas.

Read → estabelece funções que são usadas na conversão de uma string num valor do tipo de dados (instância de Read).

O MÓNADE IO

• A NOTAÇÃO "DO"

O Haskell fornece uma construção sintática (do) para escrever de forma simplificada cadeias de operações monádicas.

• FUNÇÕES DE IO DO PRELUDE

Para ler "do standard input" (por defeito, o teclado):

→ getChar : IO Char (lê um caracter)

→ getLine : IO String (lê uma string até se premir enter).

Para escrever no "standard output" (por defeito, o ecrã):

→ putChar :: Char → IO() (escreve um caracter)

→ putStr :: String → IO() (escreve uma string)

→ putStrLn :: String → IO() (escreve uma string e muda de linha)

→ print :: Show a → a → IO()

(⇐) a (putStrLn . show))

FUNÇÕES IMPORTANTES

↳ and (= ∧) [True, True] = True

↳ or (= ∨) [True, False] = True