`foldl` *(left fold)*

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

`z` é o acumulador. `f` é usado para combinar o acumulador com a cabeça da lista. `(f z x)` é o novo valor do acumulador.

Exemplo: Vejamos a relação entre função somatório implementada com um acumulador

```
sumAc [] ac = ac
sumAc (x:xs) ac = sumAc xs (ac+x)
```

```
sumAc [1,2,3] 0 = sumAc [2,3] (0+1)
                = sumAc [3] ((0+1)+2)
                = sumAc [] (((0+1)+2)+3)
                = ((0+1)+2)+3
                = 6
```

e o somatório implementado com um `foldl`

```
foldl (+) 0 [1,2,3] = foldl (+) (0+1) [2,3]
                    = foldl (+) ((0+1)+2) [3]
                    = foldl (+) (((0+1)+2)+3) []
                    = ((0+1)+2)+3
                    = 6
```


`foldl`

Muitas funções (mais do que à primeira vista poderia parecer) podem ser definidas usando o `foldl`.

Exemplo:

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

Pode ser definida assim: `inverte l = foldl (\ac x -> x:ac)) [] l`

`ac` representa o valor acumulado e `x` a cabeça da lista. `[]` é o valor inicial do acumulador.

Ou assim: `inverte l = foldl (flip (:)) [] l`

Exemplo: A função `stringToInt :: String -> Int` definida anteriormente, com um parâmetro de acumulação, pode ser definida de forma equivalente assim:

```
stringToInt l = foldl (\ac x -> 10*ac + digitToInt x) 0 l
```


`foldr vs foldl`

Note que as expressões **`(foldr f z xs)`** e **`(foldl f z xs)`** só darão o mesmo resultado se a função **`f`** for comutativa e associativa, caso contrário dão resultados distintos.

Exemplo:

```
foldr (-) 8 [4,7,3,5] = 4 – (7 – (3 – (5 - 8)))
                      = 3
```

```
foldl (-) 8 [4,7,3,5] = (((8 - 4) – 7) – 3) – 5
                      = -11
```


Tipos algébricos

A construção de tipos algébricos dá à linguagem Haskell um enorme poder expressivo, pois permite a implementação de tipos enumerados, co-produtos (união disjunta de tipos), e tipos indutivos (recursivos).

O tipo das listas é um exemplo de um tipo indutivo (recursivo) :

```
data [a] = []
         | (:) a [a]
```

Uma lista,
- ou é vazia,
- ou tem um elemento e uma sub-estrutura que é também uma lista.

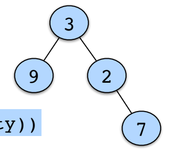
```
[1,2,3] = 1 : [2,3] = 1 : 2 : [3] = 1 : 2 : 3 : []
```

A noção de árvore binária expande este conceito.

Uma árvore binária,
- ou é vazia,
- ou tem um elemento e duas sub-estruturas que são também árvores.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```

```
Node 3 (Node 9 Empty Empty) (Node 2 Empty (Node 7 Empty Empty))
```



Árvores binárias

As árvores binárias são estruturas de dados muito úteis para organizar a informação.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
   deriving (Show)
```

Os construtores da árvores são:

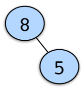
```
Empty :: BTree a

Node :: a -> (BTree a) -> (BTree a) -> (BTree a)
```

`Empty` representa a árvore vazia.

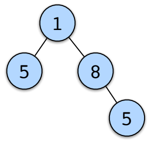
`Node` recebe um elemento e duas árvores, e constrói a árvore com esse elemento na raiz, uma árvore do lado esquerdo e outra do lado direito.

```
arv1 = Node 5 Empty Empty
```


```
arv2 = Node 8 Empty arv1
```

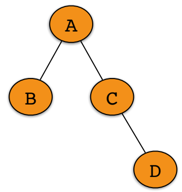



```
arv3 = Node 1 arv1 arv2
```



Árvores binárias

Terminologia



- O nodo A é a **raiz** da árvore.
- Os nodos B e C são **filhos** (ou **descendentes**) de A.
- O nodo C é **pai** de D.
- B e D são **folhas** da árvore.
- O **caminho** *(path)* de um nodo é a sequência de nodos da raiz até esse nodo. Por exemplo, A,C,D é o caminho para o modo D.
- A **altura** da árvore é o comprimento do caminho mais longo. Esta árvore tem altura 3.

Árvores binárias

As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com padrões de recursividade semelhantes aos dos tipos de dados.

Exemplo: Calcular o número de nodos que tem uma árvore.

```
conta :: BTree a -> Int
conta Empty = 0
conta (Node x e d) = 1 + conta e + conta d
```

Exemplo: Somar todos de nodos de uma árvore de números .

```
sumBT :: Num a =>  BTree a -> a
sumBT Empty = 0
sumBT (Node x e d) = x + sumBT e + sumBT d
```

```
> sumBT (Node 2 Empty (Node 7 Empty Empty))
          = 2 + (sumBT Empty) + sumBT (Node 7 Empty Empty)
          = 2 + 0 + (7 + sumBT Empty + sumBT Empty)
          = 2 + 0 + (7 + 0 + 0)
          = 9
```


Árvores binárias

Exemplo: Calcular a altura de uma árvore.

```
altura :: BTree a -> Int
altura Empty = 0
altura (Node _ e d) = 1 + max (altura e) (altura d)
```

Exemplos: As funções map e zip para árvores binárias.

```
mapBT :: (a -> b) -> BTree a -> BTree b
mapBT f Empty = Empty
mapBT f (Node x e d) = Node (f x) (mapBT f e) (mapBT f d)
```

```
zipBT :: BTree a -> BTree b -> BTree (a,b)
zipBT (Node x1 e1 d1) (Node x2 e2 d2) =
                         Node (x1,x2) (zipBT e1 e2) (zipBT d1 d2)
zipBT _ _ = Empty
```

Travessias de árvores binárias

Uma árvore pode ser percorrida de várias formas. As principais estratégias são:

Travessia preorder: visitar a raiz, depois a árvore esquerda e a seguir a árvore direita.

```
preorder :: BTree a -> [a]
preorder Empty = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

Travessia inorder: visitar árvore esquerda, depois a raiz e a seguir a árvore direita.

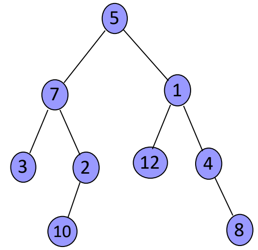
```
inorder :: BTree a -> [a]
inorder Empty = []
inorder (Node x e d) = (inorder e) ++ [x] ++ (inorder d)
```

Travessia postorder: visitar árvore esquerda, depois árvore direita, e a seguir a raiz..

```
postorder :: BTree a -> [a]
postorder Empty = []
postorder (Node x e d) = (postorder e) ++ (postorder d) ++ [x]
```


Travessias de árvores binárias

```
arv = (Node 5 (Node 7 (Node 3 Empty Empty)
                      (Node 2 (Node 10 Empty Empty) Empty)
              )
              (Node 1 (Node 12 Empty Empty)
                      (Node 4 Empty (Node 8 Empty Empty))
              )
      )
```



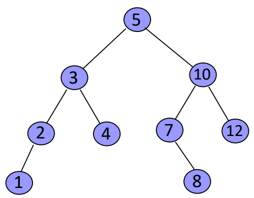

```
preorder arv  = [5,7,3,2,10,1,12,4,8]

inorder arv   = [3,7,10,2,5,12,1,4,8]

postorder arv = [3,10,2,7,12,8,4,1,5]
```


Árvores binárias de procura

Uma árvore binária em que o valor de cada nodo é maior do que os nodos à sua esquerda, e menor do que os nodos à sua direita diz-se uma árvore binária de procura (ou de pesquisa)

Uma **árvore binária de procura** é uma árvore binária que verifica as seguinte condição:
- a raiz da árvore é maior do que todos os elementos que estão na sub-árvore esquerda;
- a raiz da árvore é menor do que todos os elementos que estão na sub-árvore direita;
- ambas as sub-árvores são árvores binárias de procura.

Exemplo: Esta é uma árvore binária de procura de procura



Árvores binárias de procura

Exemplo: Testar se um elemento pertence a uma árvore binária de procura.

```
elemBT :: Ord a => a -> BTree a -> Bool
elemBT x Empty = False
elemBT x (Node y e d)
          | x < y  = elemBT x e
          | x > y  = elemBT x d
          | x == y = True
```

Exemplo: Inserir um elemento numa árvores binária de procura.

```
insertBT :: Ord a => a -> BTree a -> BTree a
insertBT x Empty = Node x Empty Empty
insertBT x (Node y e d)
          | x < y  = Node y (insertBT x e) d
          | x > y  = Node y e (insertBT x d)
          | x == y = Node y e d
```