

# Programação Funcional

## Tipos algébricos - data

2021

# Declarações de sinónimos I

Podemos definir um nome novo para um tipo existente usando uma declaração `type`.

Exemplo (do prelúdio-padrão):

```
type String = [Char]
```

Diz-se que esta declaração define um **sinónimo**.

## Declarações de sinónimos II

As declarações de sinónimos são usadas para melhorar legibilidade de programas.

Exemplo: no jogo da Vida definimos sinónimos:

```
type Pos = (Int,Int)           -- coluna,linha  
type Cells = [Pos]            -- colónia
```

Assim podemos escrever

```
isAlive :: Cells -> Pos -> Bool
```

em vez de

```
isAlive :: [(Int,Int)] -> (Int,Int) -> Bool
```

## Declarações de sinónimos III

As declarações `type` também podem ter parâmetros.

Exemplo: listas de associações entre chaves e valores.

```
type Assoc ch v = [(ch,v)]      -- lista de associações
```

```
idades :: Assoc String Int
```

```
idades = [("Pedro", 41), ("João", 27), ("Maria", 19)]
```

```
emails :: Assoc String String
```

```
emails = [("Pedro", "pbv@dcc.fc.up.pt"),  
          ("João", "joao@gmail.com")]
```

# Declarações de sinónimos IV

Os sinónimos podem ser usados noutras definições:

```
type Pos = (Int,Int)
type Cells = [Pos]                                -- OK
```

Mas não podem ser usados recursivamente:

```
type Tree = (Int,[Tree])                          -- ERRO
```

# Declarações de novos tipos I

Podemos definir novos tipos de dados usando declarações `data`.

Exemplo (do prelúdio-padrão):

```
data Bool = False | True
```

# Declarações de novos tipos II

- ▶ A declaração `data` enumera as alternativas de valores do novo tipo
- ▶ `True` e `False` são os *construtores* do tipo `Bool`
- ▶ Os construtores devem ser únicos (não podem ser usados em tipos diferentes)
- ▶ Os nome dos tipos e construtores devem começar por uma letra maiúscula

# Declarações de novos tipos III

Podemos usar novos tipos tal qual como tipos pré-definido na linguagem.

Exemplo: com a declaração

```
data Dir = Esquerda | Direita | Cima | Baixo
```

podemos definir

```
direções :: [Dir]
```

```
direções = [Esquerda, Direita, Cima, Baixo]
```

```
oposta :: Dir -> Dir
```

```
oposta Esquerda = Direita
```

```
oposta Direita  = Esquerda
```

```
oposta Cima     = Baixo
```

```
oposta Baixo    = Cima
```



# Construtores com parâmetros I

Os construtores podem também ter parâmetros.

Exemplo:

```
data Figura = Circ Float          -- raio
            | Rect Float Float -- largura, altura
```

```
quadrado :: Float -> Figura
quadrado h = Rect h h
```

```
area :: Figura -> Float
area (Circ r)    = pi*r^2
area (Rect l a) = l*a
```

# Construtores com parâmetros II

- ▶ Os construtores podem ter diferentes números de parâmetros
- ▶ Os parâmetros podem ser de tipos diferentes
- ▶ Podemos usar os construtores de duas formas:  
*aplicando argumento* para construir um valor

```
Circ :: Float -> Figura
```

```
Rect :: Float -> Float -> Figura
```

*em padrões* no lado esquerdo de equações

```
area (Circ r)    = pi*r^2
```

```
area (Rect w h) = w*h
```

# Igualdade e conversão em texto I

Por omissão novos tipos não têm instâncias de classes como Show, Eq OU Ord.

```
> show (Circ 2)
```

```
ERROR: No instance for (Show Figura)...
```

```
> Rect 1 (1+1) == Rect 1 2
```

```
ERROR: No instance for (Eq Figura)...
```

# Igualdade e conversão em texto II

Podemos pedir definições automáticas destas instâncias usando `deriving` na declaração.

```
data Figura = Circ Float
            | Rect Float Float
            deriving (Eq, Show)
```

Exemplos:

```
> show (Circ 2)
"Circ 2.0"
```

```
> Rect 1 (1+1) == Rect 1 2
True
```

# Novos tipos com parâmetros

As declarações de novos tipos também podem ter parâmetros.

Exemplo:

```
data Maybe a = Nothing | Just a   -- do prelúdio-padrão
```

Podemos usar `Maybe` para definir uma divisão inteira que não dá erros:

```
safediv :: Int -> Int -> Maybe Int  
safediv _ 0 = Nothing  
safediv n m = Just (n `div` m)   -- m diferente de 0
```

# Modelar informação I

Exemplo: items num inventário de uma loja.

```
data Produto
  = Produto Nome Stock PreçoCusto PreçoVenda

type Nome      = String
type Stock     = Int    -- quantidade inteira
type PreçoCusto = Float  -- preços em euros
type PreçoVenda = Float

inventário :: [Produto]
inventário =
  [Produto "Haskell for dummies" 4 30.0 40.0,
   Product "The C programming language" 5 35.0 50.0,
   Product "The Art of Computer Programming" 0 55.0 80.0
  ]
```

# Processar informação I

```
-- Calcular o valor de stock (euros)
valorEmStock :: [Produto] -> Float
valorEmStock items
    = sum [fromIntegral stock*custo
           | Produto _ stock custo _ <- items]

-- Aplicar um desconto a todos os items em stock
desconto :: Float -> [Produto] -> [Produto]
desconto taxa items
    = [Produto nome stock custo (venda*(1-taxa))
       | Produto nome stock custo venda <- items, stock>0]
```

# Tipos recursivos

As declarações `data` podem ser *recursivas*.

Exemplo: os números naturais.

```
data Nat = Zero | Suc Nat
```

O tipo `Nat` tem dois construtores:

- ▶ `Zero :: Nat`
- ▶ `Suc :: Nat -> Nat`



# Valores do tipo Nat

Alguns valores de Nat:

Zero

# Valores do tipo Nat

Alguns valores de Nat:

Zero

Suc Zero

# Valores do tipo Nat

Alguns valores de Nat:

Zero

Suc Zero

Suc (Suc Zero)

# Valores do tipo Nat

Alguns valores de Nat:

Zero

Suc Zero

Suc (Suc Zero)

Suc (Suc (Suc Zero))

⋮

Em geral: os valores de Nat são obtidos aplicado  $n$  vezes Succ a Zero.

Suc (Suc (... (Suc Zero)...))                      --  $n$  aplicações

Podemos pensar nestes valores como representado os naturais  $n \geq 0$ .

# Funções sobre naturais I

Podemos definir funções recursivas que convertem entre inteiros e este novo tipo.

```
natFromInt :: Int -> Nat
natFromInt 0      = Zero
natFromInt n | n>0 = Suc (natFromInt (n-1))
```

```
intFromNat :: Nat -> Int
intFromNat Zero    = 0
intFromNat (Suc n) = 1 + intFromNat n
```

# Funções sobre naturais II

Podemos usar as funções de conversão para somar naturais.

```
add :: Nat -> Nat -> Nat
```

```
add n m = natFromInt (intFromNat n + intFromNat m)
```

# Funções sobre naturais III

Em alternativa, podemos definir diretamente a adição usando recursão sobre naturais.

```
add :: Nat -> Nat -> Nat
add Zero m      = m
add (Suc n) m = Suc (add n m)
```

Estas duas equações traduzem as seguintes igualdades algébricas:

$$\begin{aligned}0 + m &= m \\ (1 + n) + m &= 1 + (n + m)\end{aligned}$$

# Exemplo

Vamos calcular soma de 2 com 1:

```
add (Suc (Suc Zero)) (Suc Zero)
```



# Exemplo

Vamos calcular soma de 2 com 1:

```
add (Suc (Suc Zero)) (Suc Zero)
=
Suc (add (Suc Zero) (Suc Zero))
```

# Exemplo

Vamos calcular soma de 2 com 1:

```
add (Suc (Suc Zero)) (Suc Zero)
=
Suc (add (Suc Zero) (Suc Zero))
=
Suc (Suc (add Zero (Suc Zero)))
```

# Exemplo

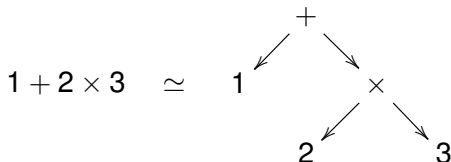
Vamos calcular soma de 2 com 1:

```
add (Suc (Suc Zero)) (Suc Zero)
=
Suc (add (Suc Zero) (Suc Zero))
=
Suc (Suc (add Zero (Suc Zero)))
=
Suc (Suc (Suc Zero))
```

# Árvores sintáticas I

Podemos representar expressões por uma *árvore sintática* em que os operadores são os *nós* e as constantes são as *folhas*.

Exemplo:



# Árvores sintáticas II

As árvores podem ser representadas em Haskell por um tipo recursivo.

```
data Expr = Val Int           -- constante
          | Soma Expr Expr    -- soma de 2 sub-expressões
          | Mult Expr Expr     -- multiplicação
```

A árvore no *slide* anterior é:

```
Soma (Val 1) (Mult (Val 2) (Val 3))
```

# Árvores sintáticas III

Podemos definir funções sobre árvores de expressões usando encaixe de padrões e recursão.

```
-- contar o tamanho da expressão
```

```
tamanho :: Expr -> Int
```

```
tamanho (Val n) = 1
```

```
tamanho (Soma e1 e2) = tamanho e1 + tamanho e2
```

```
tamanho (Mult e1 e2) = tamanho e1 + tamanho e2
```

```
-- calcular o valor representado pela expressão
```

```
valor :: Expr -> Int
```

```
valor (Val n) = n
```

```
valor (Soma e1 e2) = valor e1 + valor e2
```

```
valor (Mult e1 e2) = valor e1 * valor e2
```

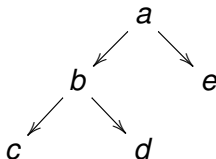
# Árvores binárias I

Um **árvore binária** é um grafo dirigido, conexo e acíclico em que cada vértice é de um de dois tipos:

**nó**: grau de saída 2 e grau de entrada 1 ou 0;

**folha**: grau de saída 0 e grau de entrada 1.

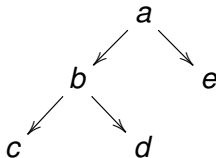
*a e b são nós; c, d e e são folhas.*



# Árvores binárias II

Numa árvore binária existe sempre um único nó, que se designa *raiz*, com grau de entrada 0.

Exemplo: a raiz é o nó *a*





# Representação recursiva I

Partindo da raiz podemos decompor uma árvore binária de forma recursiva.

Uma árvore é:

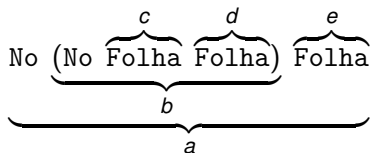
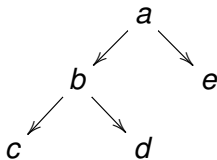
- ▶ um *nó* com duas sub-árvores; ou
- ▶ uma *folha*.

Traduzindo num tipo recursivo em Haskell:

```
data Arv = No Arv Arv -- sub-árvores esquerda e direita
        | Folha
```

# Representação recursiva II

Exemplo anterior:



# Anotações I

Podemos associar informação à árvore colocando anotações nos nós, nas folhas ou em ambos.

Alguns exemplos:

```
-- anotar nós com inteiros
data Arv = No Int Arv Arv
         | Folha
```

```
-- anotar folhas com inteiros
data Arv = No Arv Arv
         | Folha Int
```

```
-- anotar nós com inteiros e folhas com booleanos
data Arv = No Int Arv Arv
         | Folha Bool
```

# Anotações II

Em vez de usar tipos concretos, podemos parametrizar o tipo de árvore com os tipos de anotações.

Exemplos:

```
-- nós anotados com 'a'
data Arv a = No a (Arv a) (Arv a)
           | Folha
```

```
-- folhas anotadas com 'a'
data Arv a = No (Arv a) (Arv a)
           | Folha a
```

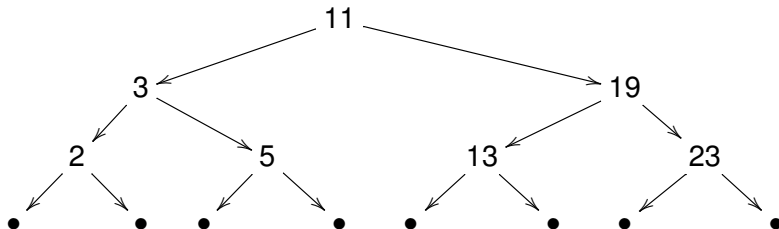
```
-- nós anotados com 'a' e folhas com 'b'
data Arv a b = No a (Arv a b) (Arv a b)
              | Folha b
```

# Árvores de pesquisa I

Uma árvore binária diz-se **ordenada** (ou **de pesquisa**) se o valor em todos os nós for:

- ▶ maior do que valores na sub-árvore esquerda;
- ▶ menor do que os valores na sub-árvore direita.

Exemplo:



# Árvores de pesquisa II

Vamos representar árvores de pesquisa por um tipo recursivo parametrizado pelo tipo dos valores guardados nos nós.

```
data Arv a = No a (Arv a) (Arv a) -- nó
          | Vazia                  -- folha
```

As folhas representam árvores vazias, pelo que não têm anotações.

# Listar todos os valores I

Para listar todos os valores de uma árvore por *ordem infix*:

- ▶ listamos a sub-árvore esquerda;
- ▶ listamos o valor do nó;
- ▶ listamos a sub-árvore direita.

O caso base da recursão é a árvore vazia.

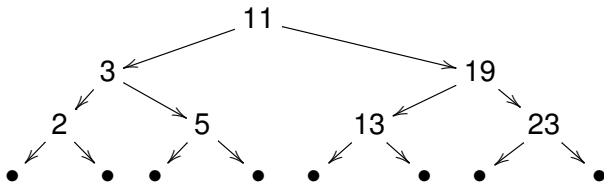
```
listar :: Arv a -> [a]
```

```
listar Vazia = []
```

```
listar (No x esq dir) = listar esq ++ [x] ++ listar dir
```

# Listar todos os valores II

Exemplo:



listar

(No 11

(No 3 (No 2 Vazia Vazia) (No 5 Vazia Vazia))

(No 19 (No 13 Vazia Vazia) (No 23 Vazia Vazia)))

==

[2, 3, 5, 11, 13, 19, 23]



## Listar todos os valores III

- ▶ Se a árvore estiver ordenada, então *listar* produz valores por ordem ascendente
- ▶ Podemos usar este facto para escrever uma função que testa se a árvore está ordenada

```
ordenada :: Ord a => Arv a -> Bool
```

```
ordenada arv = ascendente (listar arv)
```

```
  where
```

```
    ascendente []          = True
```

```
    ascendente [_]        = True
```

```
    ascendente (x:y:xs) = x<y && ascendente (y:xs)
```

# Procurar um valor I

Para procurar um valor numa árvore ordenada:

- ▶ comparamos com o valor do nó;
- ▶ recursivamente procuramos na sub-árvore esquerda ou direita.

```
procurar :: Ord a => a -> Arv a -> Bool
procurar x Vazia = False                -- não ocorre
procurar x (No y esq dir)
  | x==y = True                          -- encontrou
  | x<y  = procurar x esq                -- procura à esquerda
  | x>y  = procura x dir                 -- procura à direita
```

A restrição de classe “Ord a =>” indica que necessitamos de operações de comparação entre valores.

# Inserir um valor I

Também podemos inserir um valor numa árvore recursivamente, usando o valor em cada nó para optar por uma sub-árvore.

```
inserir :: Ord a => a -> Arv a -> Arv a
inserir x Vazia = No x Vazia Vazia
inserir x (No y esq dir)
    | x==y = No y esq dir           -- já ocorre; não insere
    | x<y  = No y (inserir x esq) dir -- insere à esquerda
    | x>y  = No y esq (inserir x dir) -- insere à direita
```

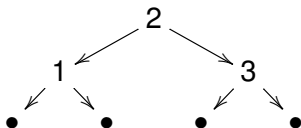
# Construir a partir duma lista I

Podemos inserir valores numa lista um-a-um recursivamente a partir da árvore vazia.

```
construir :: Ord a => [a] -> Tree a  
construir [] = Vazia  
construir (x:xs) = insert x (construir xs)
```

Exemplo:

```
construir [3,1,2]  
== No 2 (No 1 Vazia Vazia) (No 3 Vazia Vazia)
```



# Construir a partir duma lista II

Alternativa: podemos usar *foldr* em vez da recursão.

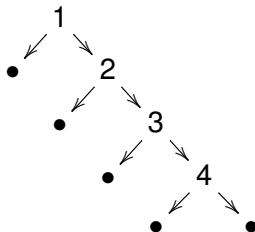
```
construir xs = foldr inserir Vazia xs
```

# Construir a partir duma lista III

- ▶ A inserção garante que a árvore fica ordenada
- ▶ Mas podemos obter uma árvore desequilibrada

construir [4,3,2,1] ==

No 1 Vazia (No 2 Vazia (No 3 Vazia (No 4 Vazia Vazia)))



# Construir árvores equilibradas I

Partindo de uma lista ordenada, podemos construir uma árvore equilibrada usando partições sucessivas.

```
-- Construir uma árvore equilibrada
-- pré-condição: a lista de valores deve estar
-- por ordem crescente
construir :: [a] -> Arv a
construir [] = Vazia
construir xs = No x (construir xs') (construir xs'')
    where n = length xs `div` 2          -- ponto médio
          xs' = take n xs                -- partir a lista
          x:xs'' = drop n xs
```

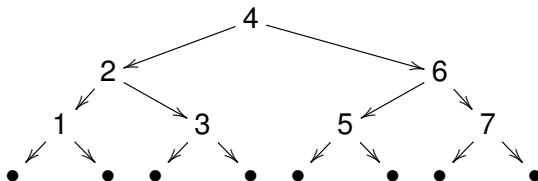
# Construir árvores equilibradas II

Exemplo:

```
construir [1,2,3,4,5,6,7]
```

==

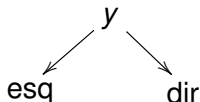
```
No 4 (No 2 (No 1 Vazia Vazia) (No 3 Vazia Vazia))  
      (No 6 (No 5 Vazia Vazia) (No 7 Vazia Vazia))
```





# Remover um valor I

Para remover um valor  $x$  numa árvore não-vazia



começamos por procurar o nó correcto:

se  $x < y$ : procuramos em *esq*;

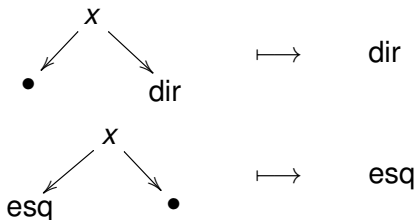
se  $x > y$ : procuramos em *dir*;

se  $x = y$ : encontramos o nó.

Se chegarmos à árvore vazia: o valor  $x$  não ocorre.

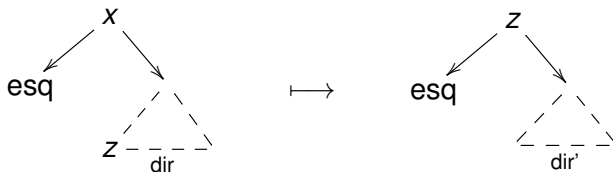
## Remover um valor II

Podemos facilmente remover um nó duma árvore com um só descendente não-vazio.



## Remover um valor III

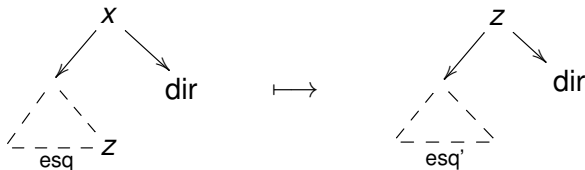
Se o nó tem dois descendentes não-vazios, então podemos substituir o seu valor pelo do *menor valor* na sub-árvore direita.



Note que temos ainda que remover *z* da sub-árvore direita.

## Remover um valor IV

Em alternativa, poderíamos usar o *maior valor* na sub-árvore esquerda.



## Remover um valor V

Usamos uma função auxiliar para obter o **o valor mais à esquerda** numa árvore de pesquisa não vazia (isto é, o *menor valor*).

```
maisEsq :: Arv a -> a
maisEsq (No x Vazia _) = x
maisEsq (No _ esq _)   = maisEsq esq
```

## Remover um valor VI

Podemos agora definir a remoção considerando os diferentes casos.

```
remover :: Ord a => a -> Arv a -> Arv a
remover x Vazia = Vazia -- não ocorre
remover x (No y Vazia dir) -- um descendente
    | x==y = dir
remover x (No y esq Vazia) -- um descendente
    | x==y = esq
remover x (No y esq dir) -- dois descendentes
    | x<y = No y (remover x esq) dir
    | x>y = No y esq (remover x dir)
    | x==y = let z = maisEsq dir
              in No z esq (remover z dir)
```

# Exercício

Escrever a definição alternativa

```
remover' :: Ord a => a -> Arv a -> Arv a
```

que usa o valor mais à direita da sub-árvore esquerda no caso dos dois descendentes não-vazios

Sugestão: escrever uma função auxiliar

```
maisDir :: Arv a -> a
```

que obtém o valor mais à direita de uma árvore não-vazia, i.e., o maior valor.