

Linguagens de Programação

Programação Funcional e Haskell
Declarando Tipos
Thiago Alves

Declarações de Tipos

Em Haskell, um novo nome para um tipo existente pode ser definido usando uma declaração de tipo.

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

String é um sinônimo para o tipo
[Char].

Declarações de tipo são úteis para tornar outros tipos mais fáceis de ler. Por exemplo, com a declaração de tipo:

```
type Pos = (Int,Int)
type Board [Pos]
```

Podemos definir:

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Declarações de tipos podem ter parâmetros.
Por exemplo:

```
type Pair a = (a,a)
```

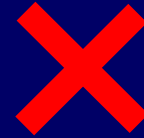
Podemos definir:

```
mult :: Pair Int → Int  
mult (m,n) = m*n
```

```
copy :: a → Pair a  
copy x = (x,x)
```

Declarações de tipos não podem ser recursivas:

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



Declarações de Dados

Um tipo completamente novo pode ser definido pela especificação dos seus valores usando uma declaração de dados.

```
data ValorVerdade = Falso | Verdadeiro
```

ValorVerdade é um novo tipo com dois novos valores Falso e Verdadeiro.

- Os dois valores Falso e Verdadeiro são chamados construtores para o tipo ValorVerdade.
- Nomes de tipos e construtores devem começar com letra maiúscula.
- O símbolo | é lido como “ou”.

Valores de novos tipos podem ser usados da mesma forma que os tipos primitivos.

```
data Move = Left | Right | Up | Down
```

Podemos definir:

```
directions :: [Move]  
directions = [Left, Up, Right]
```

```
flip :: Move → Move  
flip Left = Right  
flip Right = Left  
flip Up = Down  
flip Down = Up
```


Podemos definir a função move que recebe um movimento e uma posição e retorna outra posição de acordo com o movimento:

```
move :: Move → Pos → Pos
```

Podemos definir a função move que recebe um movimento e uma posição e retorna outra posição de acordo com o movimento:

```
move :: Move → Pos → Pos  
move Left (x,y) = (x-1,y)  
move Right (x,y) = (x+1,y)  
move Up (x,y) = (x,y+1)  
move Down (x,y) = (x,y-1)
```

Podemos definir a função moves que recebe uma lista de movimentos e uma posição e retorna a posição final de acordo com a lista de movimentos:

```
moves :: [Move] → Pos → Pos
```

Podemos definir a função `moves` que recebe uma lista de movimentos e uma posição e retorna a posição final de acordo com a lista de movimentos:

```
moves :: [Move] → Pos → Pos  
moves [] p = p  
moves (m:ms) p = moves ms (move m p)
```

Os construtores em uma declaração de dados pode ter parâmetros.

```
data Shape = Circle Float | Rect Float Float
```

Podemos definir:

```
square :: Float → Shape  
square n = Rect n n
```

```
area :: Shape → Float  
area (Circle r) = pi * r^2  
area (Rect x y) = x * y
```

- Shape possui valores da forma Circle r em que r é um float, e Rect x y em que x e y são floats.
- Circle e Rect podem ser vistos como funções que constroem valores do tipo Shape:

```
Circle :: Float → Shape
```

```
Rect   :: Float → Float → Shape
```

A própria declaração de dados pode ter parâmetros.

```
data Maybe a = Nothing | Just a
```

Podemos definir:

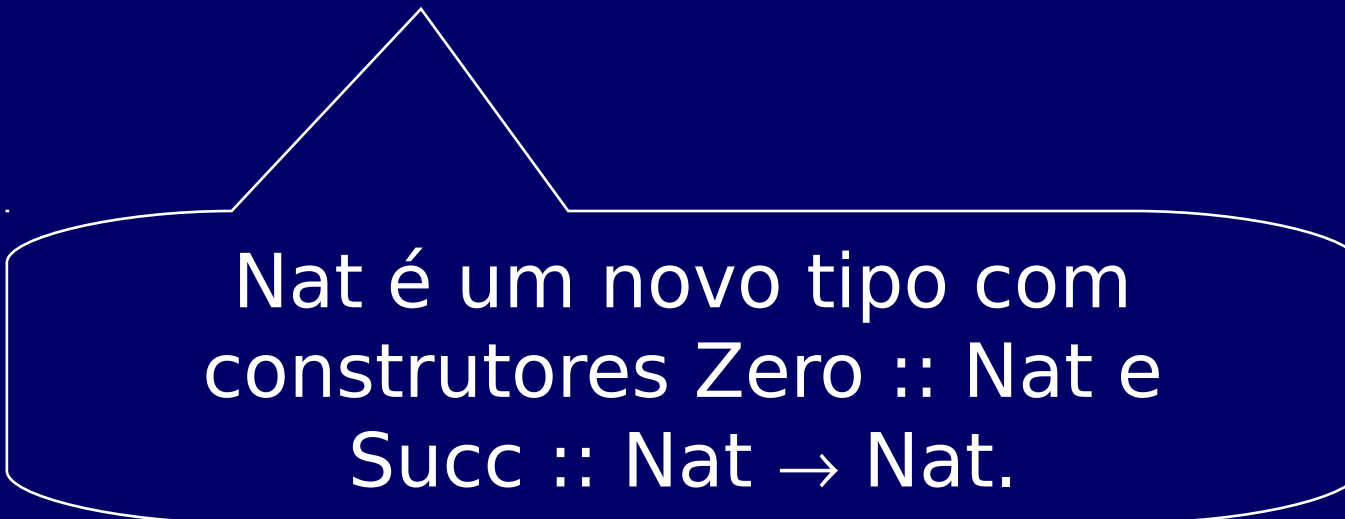
```
safediv :: Int → Int → Maybe Int  
safediv _ 0 = Nothing  
safediv m n = Just (m `div` n)
```

```
safehead :: [a] → Maybe a  
safehead [] = Nothing  
safehead xs = Just (head xs)
```

Tipos Recursivos

Em Haskell, novos tipos podem ser declarados em termos deles mesmos.

```
data Nat = Zero | Succ Nat
```



Nat é um novo tipo com construtores `Zero :: Nat` e `Succ :: Nat → Nat`.

- Um valor do tipo `Nat` ou é `Zero`, ou é da forma `Succ n` em que $n :: \text{Nat}$. Ou seja,, `Nat` possui a seguinte sequencia infinita de valores:

`Zero`

`Succ Zero`

`Succ (Succ Zero)`

⋮

- Podemos pensar nos valores do tipo Nat como números naturais, em que Zero representa 0, e Succ representa a função sucessor 1+.

- Por exemplo,

Succ (Succ (Succ Zero))

representa o número natural

$$1 + (1 + (1 + 0)) = 3$$

Usando recursão, é fácil definir funções que convertem entre valores do tipo Nat e Int:

```
nat2int :: Nat → Int
```

```
nat2int Zero = 0
```

```
nat2int (Succ n) = 1 + nat2int n
```

```
int2nat :: Int → Nat
```

```
int2nat 0 = Zero
```

```
int2nat n = Succ (int2nat (n-1))
```

Podemos definir nossa própria versão de listas:

```
Data List a = Nil | Cons a (List a)
```

E definir nossa própria versão do length:

```
len :: List a → Int
```

Podemos definir nossa própria versão de listas:

```
Data List a = Nil | Cons a (List a)
```

E definir nossa própria versão do length:

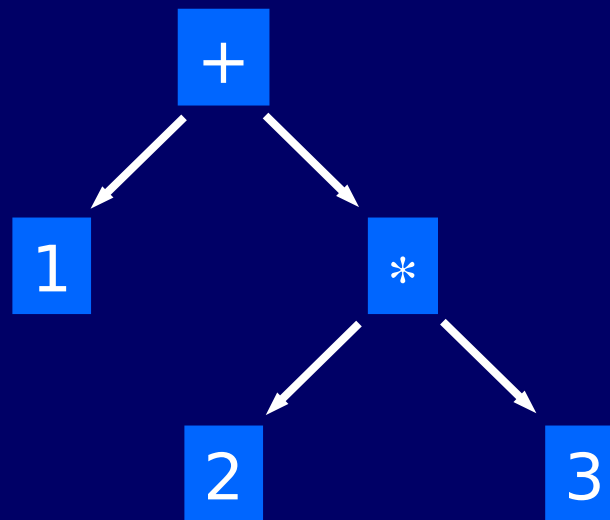
```
len :: List a → Int
```

```
len Nil = 0
```

```
len Cons x xs = 1 + length xs
```

Expressões Aritméticas

Considere uma simples forma de expressões construídas a partir de inteiros usando adição e multiplicação.



Usando tipos recursivos, podemos declarar expressões aritméticas como:

```
data Expr = Val Int | Add Expr Expr | Mul Expr Expr
```

A expressões no slide anterior é representada como:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Usando recursão, é fácil definir funções que processam expressões:

```
size :: Expr → Int
```

```
size (Val n) = 1
```

```
size (Add x y) = size x + size y
```

```
size (Mul x y) = size x + size y
```


Podemos definir uma função que efetua a expressão aritmética:

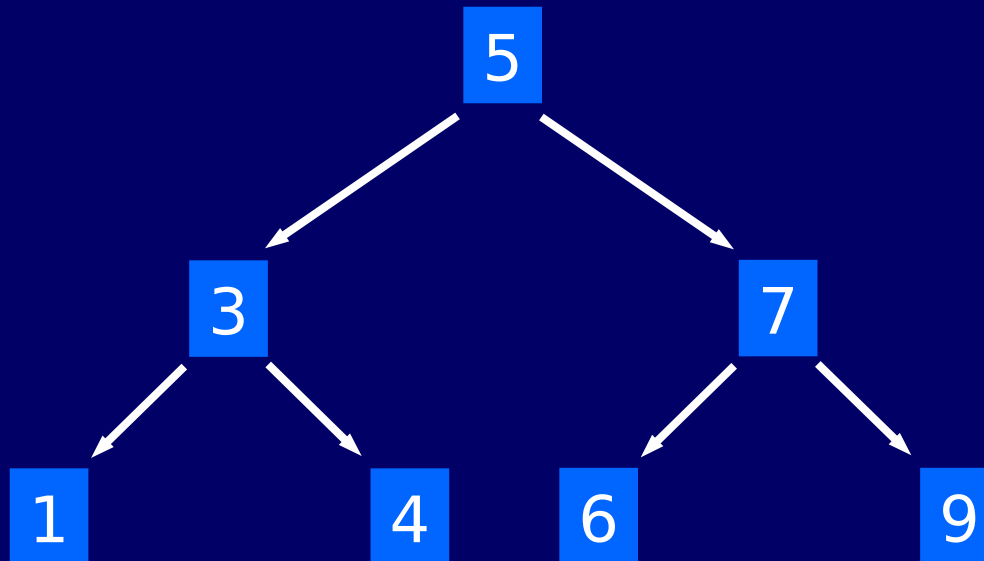
```
eval :: Expr → Int
```

Podemos definir uma função que efetua a expressão aritmética:

```
eval :: Expr → Int  
eval (Val n)   = n  
eval (Add x y) = eval x + eval y  
eval (Mul x y) = eval x * eval y
```

Árvores Binárias

São árvores em que cada nó possui dois filhos.



Usando tipos recursivos, podemos representar árvores binárias por:

Usando tipos recursivos, podemos representar árvores binárias por:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

A árvore do slide anterior é representada da seguinte forma:

```
t :: Tree Int  
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5  
        (Node (Leaf 6) 7 (Leaf 9))
```

Podemos definir uma função que decide se um dado valor ocorre na árvore binária:

```
occurs :: Eq a => a -> Tree a -> Bool
```

Podemos definir uma função que decide se um dado valor ocorre na árvore binária:

```
occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y) = x == y
occurs x (Node left y right) = x == y
                                || occurs x left
                                || occurs x right
```

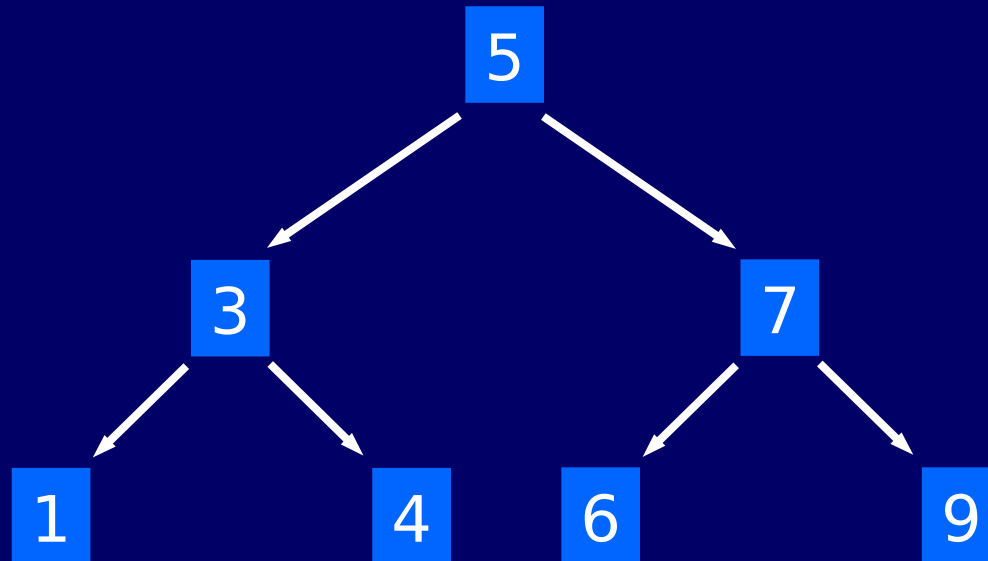
Podemos definir a função flatten que retorna a lista de todos os valores na árvore:

```
flatten :: Tree a → [a]
```


Podemos definir a função flatten que retorna a lista de todos os valores na árvore:

```
flatten :: Tree a → [a]
flatten (Leaf x)    = [x]
flatten (Node left x right) = flatten left
                             ++ [x]
                             ++ flatten right
```

Uma árvore binária de busca é um árvore em que para todo nó, a subárvore esquerda tem todos os nós com valor menor e a subárvore direita tem todos os nós com valor maior



Podemos definir a função `occursbst` que para achar um valor, escolhe a subárvore adequada

```
occursbst :: Ord a => a -> Tree a -> Bool
```

Podemos definir a função `occursbst` que para achar um valor, escolhe a subárvore adequada

```
occursbst :: Ord a => a -> Tree a -> Bool
occursbst x (Leaf y) = x == y
occursbst x (Node left y right)
    | x == y = True
    | x < y  = occursbst x left
    | x > y  = occursbst x right
```

Essa nova definição é mais eficiente pois percorre apenas um caminho na árvore.