Novos Tipos de Dados

Para além dos *tipos básicos*, dos *tipos compostos* e dos *tipos sinónimos*, o Haskell dá ainda a possibilidade de definir **novos tipos de dados**, através de declarações da forma:

Nome do *construtor do tipo*, iniciado por letra maiúscula.

Parâmetros (*variáveis de tipo*).

```
data Nome p1 ... pn = Construtor t1 ... tm | ...
                      ...
                    | Construtor t1 ... tk
```

Nome dos *construtores de dados* do tipo (ou simplesmente, construtores). Iniciado por letra maiúscula.

Tipo dos argumentos de cada construtor *(as variáveis de tipo dos parâmetros podem ocorrer neste tipos).*

Estas declarações definem ***tipos algébricos***, eventualmente, *polimórficos*.

Cada *construtor de dados* funciona como uma função (eventualmente, constante) que recebe argumentos (do tipo indicado para o construtor) e *constroi* um valor do novo tipo de dados.

Tipos Algébricos

Exemplos:

```
data Cor = Azul | Amarelo | Verde | Vermelho
```

O tipo `Cor` está a ser definido à custa de 4 construtores constantes: `Azul`, `Amarelo`, `Verde` e `Vermelho`, que serão os únicos valores deste tipo.

```
Azul :: Cor          Amarelo :: Cor
Verde :: Cor         Vermelho :: Cor
```

A este género de tipo algébrico dá-se o nome de **tipo enumerado**.

O tipo `Bool` já pré-definido é também um exemplo de um tipo enumerado.

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

Podemos agora definir funções envolvendo estes tipos algébricos:

```
fria :: Cor -> Bool
fria Azul = True
fria Verde = True
fria _ = False
```

```
quente :: Cor -> Bool
quente Amarelo = True
quente Vermelho = True
quente _ = False
```


Tipos Algébricos

Exemplo:

```
data CCart = Coord Float Float
```

Os valores do tipo `CCart` são expressões da forma `(Coord x y)`, em que `x` e `y` são valores do tipo `Float`.

`Coord` pode ser vista como uma função cujo tipo é

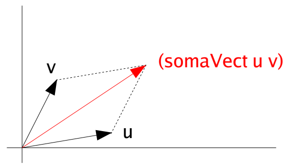
```
Coord :: Float -> Float -> CCart
```

mas os *construtores são funções especiais*, pois não têm nenhuma definição associada.

Expressões como `(Coord 1 3.1)` ou `(Coord 3 0.7)`, não podem ser reduzidas, e são exemplos de valores atómicos do tipo `CCart`.

Exemplo:

Função que soma de dois vectores:




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


Tipos Algébricos

Exemplo:

```
data Hora = AM Int Int
          | PM Int Int
```

Os valores do tipo `Hora` são expressões da forma `(AM x y)` ou `(PM x y)`, em que `x` e `y` são valores do tipo `Int`.

Os construtores do tipo `Hora` são:

```
AM :: Int -> Int -> Hora
PM :: Int -> Int -> Hora
```

e podem ser vistos como uma "etiqueta" que indica de que forma os argumentos a que são aplicados devem ser entendidos.

Os *data types* implementam o **co-produto** (ou a **união disjunta**) de tipos.

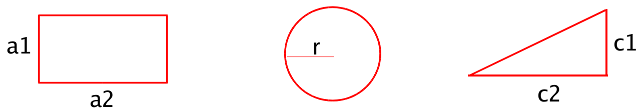
NOTA: Erradamente, pode parecer que termos como `(AM 5 10)`, `(PM 5 10)` ou `(5,10)` contêm a mesma informação, mas não! Os construtores `AM` e `PM` têm aqui um papel essencial na interpretação que fazemos destes termos.

Exemplo: As funções sobre tipos algébricos geralmente definiem-se por *pattern matching.*

```
totalMinutos :: Hora -> Int
totalMinutos (AM h m) = h*60 + m
totalMinutos (PM h m) = (h+12)*60 + m
```

Tipos Algébricos

Exemplo: Um tipo de dados para representar as seguintes figuras geométricas.




```
data Figura = Rectangulo Float Float
            | Circulo Float
            | Triangulo Float Float
```

Cálculo da área de uma figura:

```
area :: Figura -> Float
area (Rectangulo a1 a2) = a1 * a2
area (Circulo r) = pi * r^2
area (Triangulo c1 c2) = c2 * c1 / 2
```

Uma lista com figuras geométricas:

```
lfig = [(Rectangulo 5 3.2), (Circulo 5.7), (Triangulo 4 3)]
```

Note que é o facto de termos definido o tipo de dados Figura que nos permite construir esta lista, uma vez que só são aceites *listas homegéneas*.

Tipos Algébricos

As definições de tipos também podem ser *recursivas*.

Exemplo: O tipo dos números naturais pode ser definido por

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

O tipo `Nat` é definido à custa dos construtores

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

isto é,

`Zero` é um valor do tipo `Nat`, e
se `n` é um valor do tipo `Nat`, `(Suc n)` é também um valor do tipo `Nat`.

A este género de tipo algébrico dá-se o nome de **tipo recursivo**.

Exemplos:

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

São números naturais.

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

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


Tipos Algébricos

O tipo pré-definido `[a]` das listas é um outro exemplo de um *tipo recursivo*.

Exemplo: Poderiamos definir o tipo das listas, através da seguinte definição:

```
data Lista a = Nil
             | Cons a (Lista a)
```

O tipo `(Lista a)` é aqui definido à custa dos contrutores

```
Nil :: Lista a
Cons :: a -> Lista a -> Lista a
```

A lista `[3, 7, 1]` seria representada pela expressão

```
Cons 3 (Cons 7 (Cons 1 Nil))
```

`(Lista a)` é um exemplo de um **tipo polimórfico**.

`Lista` está parameterizada com uma variável de tipo **a**, que poderá ser substituida por um tipo qualquer. *(É neste sentido que se diz que `Lista` é um construtor de tipos.)*

Exemplo:

```
comprimento :: Lista a -> Int
comprimento Nil = 0
comprimento (Cons _ xs) = 1 + comprimento xs
```


Expressões `case`

O Haskell tem ainda uma forma construir expressões que permite fazer **análise de casos** sobre a estrutura dos valores de um tipo. Essas expressões têm a forma:

de um dado tipo T

de valores do tipo T

case *expressão* **of** *padrão* **->** *expressão*
...
padrão **->** *expressão*

todas de um mesmo tipo

Exemplos:

```
fria :: Cor -> Bool
fria c = case c of Azul  -> True
                   Verde -> True
                   _     -> False
```

```
comprimento :: Lista a -> Int
comprimento l = case l of
                  Nil         -> 0
                  (Cons _ xs) -> 1 + comprimento xs
```

Expressões case

Exemplos:

```
par :: Nat -> Bool
par n = case n of
          Zero           -> True
          (Suc (Suc x)) -> par x
          _              -> False
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p l = case l of
                  []     -> []
                  (x:xs) -> if (p x) then x : (takeWhile p xs)
                                     else []
```

Exercícios:

- Defina duas versões da função `impar` (com e sem expressões case).
- Defina uma outra versão da função `takeWhile` utilizando várias equações.

Nota: As expressões `if-then-else` são equivalentes à análise de casos no tipo `Bool`.

```
if e then e1
     else e2
```

é equivalente a

```
case e of True  -> e1
          False -> e2
```


A construção de tipos algébricos dá à linguagem Haskell um enorme poder expressivo, pois permite a implemetação de:

- tipos enumerdos;
- co-produtos (união disjunta de tipos);
- tipos recursivos;
- uma certa forma de *encapsulamento de dados*.

Além disso, os tipos algébricos:
(falaremos destes aspectos mais tarde)

- podem ter uma apresentação escrita própria
- podem ser declarados como instâncias de classes

Nota: Se quiser experimentar os exemplos apresentados atrás, será melhor acrescentar às declarações dos tipos algébricos a indicação: **`deriving Show`** , para que os valores dos novos tipos possam ser escritos (no formato usual).

Exemplo:

```
data Nat = Zero | Suc Nat
   deriving Show
```


O construtor de tipos Maybe

Um tipo algébrico importante, já pré-definido no `Prelude` é o tipo polimórfico

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

que permite representar a *parcialidade*, podendo ser usado para lidar com situações de excepções e erros.

Exemplos:

```
divisao :: Integer -> Integer -> Maybe Integer
divisao x y | y /= 0 = Just (x `div` y)
            | otherwise = Nothing
```

```
cabeca :: [a] -> Maybe a
cabeca (x:xs) = Just x
cabeca [] = Nothing
```

em casos de excepções o resultado é `Nothing`

Funções que trabalham sobre um tipo `t` terão que ser adaptadas para trabalhar com o tipo `Maybe t`.

Exemplo:

```
soma (Just x) (Just y) = Just (x+y)
soma _ _ = Nothing
```


Exemplo: A seguinte função que procura o nome associado a um dado número de BI, numa tabela implementada como uma lista de pares.

```
type BI = Integer
type Nome = String
```

```
procura :: BI -> [(BI,Nome)] -> Nome
procura n ((x,y):xs) | n == x     = y
                     | otherwise = procura n xs
procura _ [] = error "Não existe!"
```

A função `procura` termina em <u>erro</u> caso o BI não exista na tabela.
Ou seja, `procura` é uma função parcial.

Podemos *totalizar* a função de procura usando o tipo `(Maybe Nome)`.

```
proc :: BI -> [(BI,Nome)] -> Maybe Nome
proc n ((x,y):xs) | n == x     = Just y
                  | otherwise = proc n xs
proc _ [] = Nothing
```

Desta forma, se o BI não existir na tabela a função `proc` devolve `Nothing` e nunca termina em erro. Ou seja, `proc` é uma função total.

A função `proc` só consegue concluir que um dado BI não ocorre na tabela, ao fim de pesquisar toda a lista.

Esta conclusão poderia ser tirada mais cedo, se que a lista estivesse *ordenada* por BI.

Exemplo: Tendo a garantia de que a lista está ordenada por ordem crescente de BI, podemos definir a função de procura da seguinte forma:

```
proc :: BI -> [(BI,Nome)] -> Maybe Nome
proc n ((x,y):xys) | n < x  = proc n xys
                   | n == x = Just y
                   | n > x  = Nothing
proc _ [] = Nothing
```

Nos casos de insucesso, esta versão de `proc` é bastante *mais eficiente* do que a versão do slide anterior.

Exercício: Compare o funcionamento das duas versões da função `proc` para o seguinte exemplo:

```
proc 3  [ (bi,"xxxxx") | bi <- [1,5..1000] ]
```


Árvores Binárias

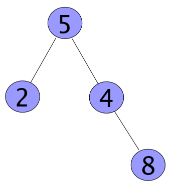
Uma estrutura de dados muito útil para organizar informação são as **árvores binárias**.

O tipo polimórfico das árvores binárias pode definido pelo seguite tipo recursivo:

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
```

Ou seja, uma árvore binária: ou é vazia; ou é um nodo com um valor e duas sub-árvores.

Exemplo: A árvore de valores inteiros:



é representada pela expressão

```
(Nodo 5 (Nodo 2 Vazia Vazia)
        (Nodo 4 Vazia (Nodo 8 Vazia Vazia))
)
```

de tipo `(ArvBin Int)`.

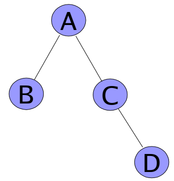
As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com *padrões de recursividade semelhantes aos dos tipos de dados*.

Exemplo:

```
somaL :: [Int] -> Int
somaL [] = 0
somaL (x:xs) = x + (somaL xs)
```

```
somaA  :: ArvBin Int -> Int
somaA Vazia = 0
somaA (Nodo x esq dir) = x + (somaA esq) + (somaA dir)
```

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

O ***caminho*** *(path)* de um nodo é a sequência de nodos da raiz até esse nodo.
A ***altura*** é o comprimento do caminho mais longo.

Funções sobre árvores binárias

Exemplos:

```
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)
```

Exercício: Defina as funções

`contaNodos :: ArvBin a -> Integer`

`zipAB :: ArvBin a -> ArvBin b -> ArvBin (a,b)`