Árvores irregulares *(rose trees)*

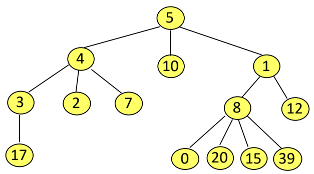
Nas **árvores irregulares** cada nodo pode ter um número variável de descendentes. O seguinte tipo de dados é uma implementação de árvores irregulares, não vazias.

```
data RTree a = R a [RTree a]
   deriving (Show)
```

O único construtor da árvore é `R :: a -> [RTree a] -> RTree a`

R recebe o elemento que fica na raiz da árvore e a lista das sub-árvores com que vai construir a árvore.

```
R 5 [ R 4 [ R 3 [R 17 []], R 2 [], R 7 []],
      R 10 [],
      R 1 [ R 8 [ R 0 [], R 20 [], R 15 [],  R 39 [] ],
            R 12 [] ]
    ]
```



Árvores irregulares *(rose trees)*

Como é de esperar, as funções definidas sobre rose trees seguem um padrão de recursividade compatível com sua definição indutiva.

Exemplo: Contar os nodos de uma árvore.

```
contaRT :: RTree a -> Int
contaRT (R x l) = 1 + sum (map contaRT l)
```

Exemplo: Calcular a altura de uma árvore.

```
alturaRT :: RTree a -> Int
alturaRT (R x []) = 1
alturaRT (R x l) = 1 + maximum (map alturaRT l)
```


Árvores irregulares *(rose trees)*

Exemplo: Testar se um elemento pertence a uma árvore.

```
pertenceRT :: Eq a => a -> RTree a -> Bool
pertenceRT x (R y l) = x==y || or (map (pertenceRT x) l)
```

(pertenceRT x) :: RTree a -> Bool

Exemplo: Fazer uma travessia *preorder* uma árvore.

```
preorderRT :: RTree a -> [a]
preorderRT (R x l) = x : concat (map preorderRT l)
```

Exercício: Defina uma função que converte uma árvore binária numa *rose tree.*

Outras árvores

Leaf trees: Árvores binárias em que a informação está apenas nas folhas da árvore. Os nós intermédios não têm informação.

```
data LTree a = Tip a
             | Fork (LTree a) (LTree a)
```

Full trees: Árvores binárias que têm informação nos nós intermédios e nas folhas. A informação guardada nos nós e nas folhas pode ser de tipo diferente.

```
data FTree a b = Leaf b
               | No a (FTree a b) (FTree a b)
```

Overloading

- Em Haskell é possível usar o mesmo identificador para funções computacionalmente distintas. A isto chama-se sobrecarga (*overloading*) de funções.
- Ao nível do sistema de tipos a sobrecarga de funções é tratada introduzindo o conceito de classe e **tipos qualificados**.

Exemplo:

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

```
> 3 + 2
5
> 10.5 + 1.7
12.2
> 'a' + 'b'
error: …
```

- `a = Int` que pertence à classe `Num`
- `a = Float` que pertence à classe `Num`
- `Char` não pertence à classe `Num`

Classes & instâncias

- As **classes** são uma forma de classificar tipos quanto às funcionalidades que lhe estão associadas.
  - Uma classe estabelece um conjunto de assinaturas de funções (i.e., seu nome e tipo).
  - Os tipos que são declarados como instâncias dessa classe têm que ter essas funções definidas.

Exemplo: A declaração (simplificada) da classe `Num`

```
class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
```

exige que todo o tipo `a` da classe `Num` tenha que ter as funções `(+)` e `(*)` definidas

Para declarar `Int` e `Float` como pertencendo à classe `Num` têm que se fazer as seguintes declarações de instância:

```
instance Num Int where
    (+) = primPlusInt
    (*) = primMulInt
```

```
instance Num Float where
    (+) = primPlusFloat
    (*) = primMulFloat
```


Classes & instâncias

```
instance Num Int where
    (+) = primPlusInt
    (*) = primMulInt
```

```
instance Num Float where
    (+) = primPlusFloat
    (*) = primMulFloat
```

Neste caso as funções `primPlusInt`, `primMulInt`, `primPlusFloat` e `primMulFloat` são funções primitivas da linguagem.

```
> 3 + 2
5
> 10.5 + 1.7
12.2
> 7 * 3
21
> 3.4 * 2.0
6.8
```

- ``3 `primPlusInt` 2``
- ``10.5 `primPlusFloat` 1.7``
- ``7 `primMultInt` 3``
- ``3.4 `primMultFloat` 2.0``

Tipos principais

O **tipo principal** de uma expressão é o tipo mais geral que lhe é possível associar, de forma a que todas as possíveis instâncias desse tipo constituam ainda tipos válidos para a expressão.

- Toda a expressão válida tem um tipo principal único.
- O Haskell infere sempre o tipo principal de uma expressão.

Exemplo: Podemos definir uma classe `FigFechada`

```
class FigFechada a where
    area :: a -> Float
    perimetro :: a -> Float
```

e definir a função `areaTotal` que calcula o total das áreas das figuras que estão numa lista

```
areaTotal l = sum (map area l)
```

```
> :type areaTotal
areaTotal :: (FigFechada a) => [a] -> Float
```

A classe `Eq`

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    -- Minimal complete definition: (==) or (/=)
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Esta classe estabelece as funções `(==)` e `(/=)` e, para além disso, fornece também definições por omissão para estas funções *(default methods)*.

- Caso a definição de uma função seja omitida numa declaração de instância, o sistema assume a definição feita na classe.
- Se existir uma nova definição da função na declaração de instância, será essa definição a ser usada.

A classe `Eq`

Exemplo: Considere a seguinte definição do tipo dos números naturais

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

Um valor do tipo `Nat` ou é `Zero`, ou é `Suc n`, em que `n` é do tipo `Nat`

Os valores do tipo `Nat` são portanto, `Zero, Suc Zero, Suc (Suc Zero), …`

O tipo `Nat` pode ser declarado como instância da classe `Eq` assim:

```
instance Eq Nat where
    (Suc n) == (Suc m) = n == m
    Zero == Zero       = True
    _ == _             = False
```

A função `(/=)` fica definida por omissão.

Esta declaração de instância, por testar a **igualdade literal (estrutural)** entre dois valores do tipo `Nat`, poderia ser derivada automaticamente fazendo

```
data Nat = Zero | Suc Nat
   deriving (Eq)
```


A classe `Eq`

Nem sempre a igualdade estrutural (que testa se dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos também iguais) é o que precisamos.

Exemplo: Considere o seguinte tipo para representar horas em dois formatos distintos.

```
data Time = AM Int Int | PM Int Int | Total Int Int
```

Queremos, por exemplo, que `(PM 3 30)` e (Total 15 30) sejam iguais, pois representam a mesma hora do dia.

Exercício: Defina uma função que converte para minutos um valor `Time` e, com base nela, declare `Time` como instância da classe `Eq`.

Instâncias com restrições

Exemplo: Considere o tipo das árvores binárias.

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

Só poderemos declarar `(BTree a)` como instância da classe `Eq` se o tipo `a` for também uma instância da classe `Eq`. Este tipo de restrição pode ser colocado na declaração de instância, fazendo:

```
instance (Eq a) => Eq (BTree a) where
    Empty == Empty = True
    (Node x1 e1 d1) == (Node x2 e2 d2) = (x1==x2) && (e1==e2)&& (d1==d2)
    _ == _  = False
```

Igualdade sobre valores do tipo `(BTree a)`

Igualdade sobre valores do tipo `a`

Esta declaração de instância poderia ser derivada automaticamente fazendo:

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