"Records"

Sendo `p` um valor do tipo `PontoC`, **`p {xx=0}`** é um novo valor com o campo `xx=0` e os restantes campos com o valor que tinham em `p`.

Exemplos:

```
p1 {cor = Amarelo}  ⇒  Pt {xx=3.2, yy=5.5, cor=Amarelo)
p3 {xx=0, yy=0}     ⇒  Pt {xx=0, yy=0, cor=Verde}
```

```
simetrico :: PontoC -> PontoC
simetrico p = p {xx=(yy p), yy=(xx p)}
```

É possível ter campos etiquetados em tipos com mais de um construtor.
Um campo não pode aparecer em mais do que um tipo, mas dentro de um tipo pode aparecer associado a mais de um construtor, desde que tenha o mesmo tipo.

Exemplo:

```
data EX = C1 { s :: Int, r :: Float }
        | C2 { s :: Int, w :: String }
```


Polimorfismo paramétrico

Com já vimos, o sistema de tipos do Haskell incorpora tipos polimórficos, isto é, tipos com variáveis (*quantificadas universalmente*, de forma implícita).

Exemplos:

Para qualquer tipo `a`, `[a]` é o tipo das listas com elementos do tipo `a`.

Para qualquer tipo `a`, `(ArvBin a)` é o tipo das árvores binárias com nodos do tipo `a`.

As variáveis de tipo podem ser vistas como *parâmetros (dos constructores de tipos)* que podem ser substituídos por tipos concretos. Esta forma de polimorfismo tem o nome de ***polimorfismo paramétrico***.

Exemplo:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + (length xs)
```

```
length [5.6,7.1,2.0,3.8]    ⇒  4
length ['a','b','c']        ⇒  3
length [(3,True),(7,False)] ⇒  2
```

```
Prelude> :t length
length :: forall a. [a] -> Int
```

O tipo `[a]->Int` não é mais do que uma abreviatura de $\forall a.[a]\text{->}Int$:

*"**para todo o tipo** a, [a]->Int é o tipo das funções com domínio em [a] e contradomínio Int".*

Polimorfismo ad hoc (sobrecarga)

O Haskell incorpora ainda uma outra forma de polimorfismo que é a ***sobrecarga** de funções*. Um mesmo identificador de função pode ser usado para designar funções computacionalmente distintas. A esta caracterísa também se chama ***polimorfismo ad hoc***.

Exemplos:

O operador `(+)` tem sido usado para somar, tanto valores inteiros como valores decimais.

O operador `(==)` pode ser usado para comparar inteiros, caracteres, listas de inteiros, strings, booleanos, ...

Afinal, qual é o tipo de `(+)` ? E de `(==)` ?

A sugestão
```
(+)  :: a -> a -> a
(==) :: a -> a -> Bool
```
não serve, pois são tipos demasiado genéricos !

Faria com que fossem aceites espressões como, por exemplo:

`('a' + 'b')` , `(True + False)` , `("esta" + "errado")` ou `(div == mod)` ,

e estas expressões resultariam em **erro**, pois estas operações não estão preparadas para trabalhar com valores destes tipos.

Em Haskell esta situação é resolvidas através de **tipos qualificados** *(qualified types)*, fazendo uso da noção de **classe**.

Tipos qualificados

Conceptualmente, um tipo qualificado pode ser visto como um tipo polimórfico só que, em vez da quantificação universal da forma *"para todo o tipo a, ..."* vai-se poder dizer *"para todo o tipo a que pertence à classe C, ..."* . Uma classe pode ser vista como um conjunto de tipos.

Exemplo:

Sendo `Num` uma classe *(a classe dos números)* que tem como elementos os tipos: `Int`, `Integer`, `Float`, `Double`, ..., pode-se dar a `(+)` o tipo preciso de:

$\forall a \in Num.\ a\text{->}a\text{->}a$

o que em Haskell se vai escrever:
```
(+) :: Num a => a -> a -> a
```

e lê-se: *"para todo o tipo a que pertence à classe Num, (+) tem tipo a->a->a".*

Uma classe surge assim como uma forma de classificar tipos (quanto às funcionalidades que lhe estão associadas). Neste sentido as classes podem ser vistas como os *tipos dos tipos*.

Os tipos que pertencem a uma classe também serão chamados de ***instâncias*** da classe.

A capacidade de *qualificar* tipos polimórficos é uma característica inovadora do Haskell.

Classes & Instâncias

Uma ***classe*** estabelece um conjunto de assinaturas de funções (os *métodos da classe*). Os tipos que são declarados como ***instâncias*** dessa classe têm que ter definidas essas funções.

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

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

impõe que todo o tipo `a` da classe `Num` tenha que ter as operações `(+)` e `(*)` definidas.

Para declarar `Int` e `Float` como elementos da classe `Num`, tem que se fazer as seguintes *declarações de instância*

```
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.

Se `x::Int` **e** `y::Int` **então** `x + y ⇒ x` `` `primPlusInt` `` `y`
Se `x::Float` **e** `y::Float` **então** `x + y ⇒ x` `` `primPlusFloat` `` `y`

Tipo principal

O **tipo principal** de uma expressão ou de uma função 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 ou função.

Qualquer expressão ou função válida tem um tipo principal *único*. O Haskell *infere* sempre o tipo principal das expressões ou funções, mas é sempre possível associar tipos mais específicos (que são instância do tipo principal).

Exemplo: O tipo principal inferido pelo haskell para o operador (+) é

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

Mas,
```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```
são também tipos válidos dado que tanto `Int` como `Float` são instâncias da classe `Num`, e portando podem substituir a variável `a`.

Note que `Num a` não é um tipo, mas antes uma restrição sobre um tipo. Diz-se que `(Num a)` é o ***contexto*** para o tipo apresentado.

Exemplo:
```
sum [] = 0
sum (x:xs) = x + sum xs
```

O tipo principal da função sum é
```
sum :: Num a => [a] -> a
```

- `sum::[a]->a` seria um tipo demasiado geral. Porquê ?
- Qual será o tipo principal da função `product` ?

Definições por defeito

Relembre a definição da função pré-definida `elem`:

```
elem x [] = False
elem x (y:ys) = (x==y) || elem x ys
```

Qual será o seu tipo ?

É necessário que (==) esteja definido para o tipo dos elementos da lista.

Existe pré-definida a classe **`Eq`**, dos tipos para os quais existe uma operação de igualdade.

```
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 defeito** para estes métodos *(default methods)*.

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

Exemplos de instâncias de Eq

O tipo `Cor` é uma instância da classe `Eq` com `(==)` definido como se segue:

O método `(/=)` está definido por defeito.

```
instance Eq Cor where
    Azul == Azul          = True
    Verde == Verde        = True
    Amarelo == Amarelo    = True
    Vermelho == Vermelho  = True
    _ == _                = False
```

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

(==) de `Nat`

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

O tipo `PontoC` com instância de `Eq`:

(==) de `Float`

(==) de `Cor`

```
instance Eq PontoC where
    (Pt x1 y1 c1) == (Pt x2 y2 c2) = (x1==x2) && (y1==y2)
                                     && (c1==c2)
```

Nota: `(==)` é uma função recursiva em `Nat`, mas não em `PontoC`.

Instâncias com restrições

Relembre a definição das árvores binárias.

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

Como poderemos fazer o teste de igualdade para árvores binárias ?

Duas árvores são iguais se tiverem a mesma estrutura (a mesma forma) e se os valores que estão nos nodos também forem iguais.

Portanto, para fazer o teste de igualdade em `(ArvBin a)`, necessariamente, tem que se saber como testar a igualdade entre os valores que estão nos nodos, i.e., em `a`.

Só poderemos declarar `(ArvBin a)` como instância da classe `Eq` se `a` for também uma instância da classe `Eq`.

Este tipo de *restrição* pode ser colocado na declaração de instância, fazendo:

```
instance (Eq a) => Eq (ArvBin a) where
    Vazia == Vazia  = True
    (Nodo x1 e1 d1) == (Nodo x2 e2 d2) = (x1==x2) && (e1==e2)
                                         && (d1==d2)
    _ == _  = False
```

(==) de `a`

(==) de `(ArvBin a)`

Instâncias derivadas de Eq

O testes de igualdade definidos até aqui implementam a **igualdade estrutural** *(dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos também iguais).*

Quando assim é pode-se evitar a declaração de instância se na declaração do tipo for acrescentada a instrução **`deriving Eq`**.

Exemplos: Com esta declarações, o Haskell deriva automáticamente declarações de instância de `Eq` (iguais às que foram feitas) para estes tipos.

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

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

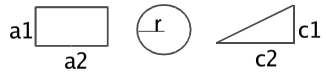
```
data PontoC = Pt {xx :: Float, yy :: Float, cor :: Cor}
    deriving Eq
```

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


Mas, nem sempre a igualdade estrutural é a desejada.

Exemplo: Relembre o tipo de dados `Figura`:

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



Neste caso queremos que duas figuras sejam consideradas iguais ainda que a ordem pela qual os valores são passados possa ser diferente.

```
instance Eq Figura where

    (Rectangulo x1 y1) == (Rectangulo x2 y2) =
                ((x1==x2) && (y1==y2)) || ((x1==y2)) && (x2==y1))

    (Circulo r1) == (Circulo r2) = r1==r2

    (Triangulo x1 y1) == (Triangulo x2 y2) =
                ((x1==x2) && (y1==y2)) || ((x1==y2)) && (x2==y1))
```


Exercícios:

- Considere a seguinte definição de tipo, para representar horas nos dois formatos usuais.

```
data Time = Am Int Int
          | Pm Int Int
          | Total Int Int
```

  Declare `Time` como instância da classe `Eq` de forma a que `(==)` teste se dois valores representam a mesma hora do dia, independentemente do seu formato.

- Qual o tipo principal da seguinte função:

```
lookup x ((y,z):yzs) | x /= y   = lookup x yzs
                     | othewise = Just z
lookup _ [] = Nothing
```

- Considere a seguinte declaração: `type Assoc a b = [(a,b)]`

  Será que podemos declarar `(Assoc a b)` como instância da classe `Eq` ?