

Aula Prática 8

Paradigmas da Programação I / Programação Funcional

ESI/MCC 1º ano (2005/2006)

Nesta aula pretende-se que os alunos: consolidem os conceitos de *classe*, *instância de classe* e de *tipo qualificado*; explorem algumas das classes pré-definidas mais usadas do Haskell (por exemplo: `Eq`, `Ord`, `Num`, `Show`, `Enum`, ...) e definam novas classes.

Classes, Instâncias de Classes e Tipos Qualificados

As classes em Haskell permitem classificar tipos. Mais precisamente, uma classe estabelece um conjunto de assinaturas de funções (os *métodos* da classe) cujos tipos que são instância dessa classe devem ter definidas.

É possível que uma classe já forneça algumas funções *definidas por omissão*. Caso uma função não seja definida explicitamente numa declaração de instância, o sistema assume a definição por omissão estabelecida na classe. Se existir uma nova definição do método na declaração de instância, será essa definição a ser usada.

Para saber informação sobre uma determinada classe pode usar o comando do interpretador ghci `:i nome_da_classe`. Em anexo apresenta-se um resumo de algumas das classes pré-definidas mais utilizadas do Haskell.

Tarefa 1 Verifique qual é o tipo inferido pelo ghci (*o tipo principal*), para cada uma das seguintes funções (pré-definidas): `elem`, `sum` e `minimum`. Justifique o tipo da função, com base no que deverá ser a sua definição.

Tarefa 2 Considere as seguintes declarações de tipo usadas para representar as horas de um dia nos formatos usuais.

```
data Part = AM | PM
    deriving (Eq, Show)

data TIME = Local Int Int Part
    | Total Int Int
```

1. Defina algumas constantes do tipo `TIME`.
2. Defina a função `totalMinutos :: TIME -> Int` que conta o total de minutos de uma dada hora.

3. Defina `TIME` como instância da classe `Eq` de forma a que a igualdade entre horas seja independente do formato em que hora está guardada.
4. Defina `TIME` como instância da classe `Ord`.
5. Defina `TIME` como instância da classe `Show`, de modo a que a apresentação dos termos (`Local 10 35 AM`), (`Local 4 20 PM`) e (`Total 17 30`) seja respectivamente: `10:35 am`, `4:20 pm` e `17h30m`.
6. Defina a função `seleciona :: TIME -> [(TIME,String)] -> [(TIME,String)]` que recebe uma hora e uma lista de horários de cinema, e seleciona os filmes que começam depois de uma dada hora.
7. Declare `TIME` como instância da classe `Enum`, de forma a que `succ` avance o relógio 1 minuto e `pred` recue o relógio 1 minuto. Assuma que o sucessor de `11:59 pm` é `00:00 am`. Depois, faça o interpretador calcular o valor das seguintes expressões: `[(Total 10 30)..(Total 10 35)]` e `[(Total 10 30),(Local 10 35 AM)..(Total 15 20)]`.

Tarefa 3 Considere as declarações da classe `FigFechada` e da função `fun` a seguir apresentadas

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

fun figs = filter (\fig -> (area fig) > 100) figs
```

1. Indique, justificado, qual é o tipo inferido pelo interpretador Haskell para a função `fun`.
2. No plano cartesiano um retângulo com os lados paralelos aos eixos pode ser univocamente determinado pelas coordenadas do vértice inferior esquerdo e pelos comprimentos dos lados, ou por uma diagonal dada por dois pontos. Assim, para representar esta figura geométrica, definiu-se em Haskell o seguinte tipo de dados:

```
type Ponto = (Float,Float)
type Lado = Float
data Rectangulo = PP Ponto Ponto
                | PLL Ponto Lado Lado
```

Declare `Rectangulo` como instância da classe `FigFechada`.

3. Defina a função `somaAreas :: [Rectangulo] -> Float` que calcula o somatório de uma lista de retângulos. (De preferência, utilize funções de ordem superior.)

Tarefa ** Revisite a tarefa 2 da ficha da aula prática 7. Redefina o tipo `Aluno` e declare-o como instância da classe `Ord`, de forma a que possa reutilizar as funções de inserção e pesquisa sobre árvores binárias de procura polimórficas, já definidas nas aulas teóricas.

Tarefa ** Revisite a tarefa 1 da ficha da aula prática 7, e defina `Expr` como instância da classe `Num`. Depois, experimente usar a função `sum` para somar listas de `Expr`.

Algumas das classes pré-definidas do Haskell

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)
```

A classe Ord

```
data Ordering = LT | EQ | GT
               deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

```
class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

  -- Minimal complete definition: (<=) or compare
  -- using compare can be more efficient for complex types
  compare x y | x==y      = EQ
               | x<=y     = LT
               | otherwise = GT

  x <= y          = compare x y /= GT
  x < y           = compare x y == LT
  x >= y          = compare x y /= LT
  x > y           = compare x y == GT

  max x y | x <= y      = y
           | otherwise  = x
  min x y | x <= y      = x
           | otherwise  = y
```

A classe Num

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
  -- Minimal complete definition: All, except negate or (-)
  x - y          = x + negate y
  negate x       = 0 - x
```

A classe Show

```
class Show a where
  show      :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList  :: [a] -> ShowS

  -- Minimal complete definition: show or showsPrec
  show x      = showsPrec 0 x ""
  showsPrec _ x s = show x ++ s
  showList [] = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showl xs
                    where showl [] = showChar ']'
                          showl (x:xs) = showChar ',' . shows x . showl xs
```

A classe Enum

```
class Enum a where
  succ, pred      :: a -> a
  toEnum          :: Int -> a
  fromEnum        :: a -> Int
  enumFrom        :: a -> [a]           -- [n..]
  enumFromThen    :: a -> a -> [a]     -- [n,m..]
  enumFromTo      :: a -> a -> [a]     -- [n..m]
  enumFromThenTo  :: a -> a -> a -> [a] -- [n,n'..m]

  -- Minimal complete definition: toEnum, fromEnum
  succ      = toEnum . (1+) . fromEnum
  pred      = toEnum . subtract 1 . fromEnum
  enumFrom x = map toEnum [ fromEnum x .. ]
  enumFromThen x y = map toEnum [ fromEnum x, fromEnum y .. ]
  enumFromTo x y = map toEnum [ fromEnum x .. fromEnum y ]
  enumFromThenTo x y z = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```