Paradigmas da Programação I (LESI)
Programação Funcional (LMCC)

Ficha 4

Ano lectivo 2004/05

Tipos sinónimos e novos tipos de dados

A declaração `type` do *Haskell* permite associar um identificador a um tipo de dados já existente — estabelece um *sinónimo de tipos*. Como exemplo, poderíamos declarar

```
type ListaInt = [Int]

len :: ListaInt -> Int
len [] = 0
len (x:xs) = 1 + (len xs)
```

Aqui, `ListaInt` é só um nome alternativo para o tipo `[Int]`. Verifica-se então que a declaração `type` não oferece a habilidade de definir novos tipos de dados mas simplesmente dar novos nomes a tipos já existentes[1].

Para além de permitir atribuir sinónimos de tipos, o *Haskell* permite ainda declarar novos tipos de dados. Para tal utilizamos a declaração `data` para explicitar como podem ser "construídos" os valores do novo tipo. Um exemplo muito simples:

```
data DiaSemana = Dom | Seg | Ter | Qua | Qui | Sex | Sab

diaUtil :: DiaSemana -> Bool
diaUtil Dom = False
diaUtil Sab = False
diaUtil _   = True
```

O novo tipo `DiaSemana` contém como valores `Dom`, `Seg`, ..., `Sab`. A predicado `diaUtil` é apresentado como exemplo de uma função que manipula o novo tipo de dados.

No tipo apresentado, enumeraram-se todos os possíveis valores desse tipo. Esta forma de definirmos novos tipos só é viável se o conjunto de valores for muito restrito. Normalmente, em vez de enumerarmos todos os valores, informarmos o *Haskell* de "como pode construir" os valores do novo tipo (podendo para o efeito utilizar outros tipos disponíveis). Vejamos como poderíamos definir um tipo para representar sequências de inteiros (análogo às listas de inteiros)[2]:

```
data ListInt2 = LVazia
              | LCons Int ListaInt
              deriving (Eq,Show)
```

[1] Um exemplo da utilização dos sinónimos de tipos foi já referido neste curso: o tipo `String` é só um nome alternativo para o tipo `[Char]`. De facto, no `Prelude` do *Haskell* está algures declarado "`type String = [Char]`"

[2] Neste exemplo adiciona-se à declaração de tipo a instrução `deriving (Eq,Show)` que dá instruções ao *Haskell* para derivar automaticamente as funções de comparação e de visualização para este tipo. Trata-se de um exemplo de utilização de *classes* que será estudado adiante.

```
-- para representar a sequência "2,7,1"
exemploSeq = LCons 2 (LCons 7 (LCons 1 LVazia))

len2 :: ListaInt2 -> Int
len2 LVazia = 0
len2 (LCons x xs) = 1 + (len2 xs)
```

Neste caso, os valores do tipo `ListaInt2` serão `LVazia` e `(LCons x xs)` quando `x` e `xs` forem valores do tipo `Int` e `ListaInt2` respectivamente. Por isso, dizemos que `LVazia` e `LCons` são os *construtores* do tipo `ListaInt`. Como exemplo apresenta-se um valor do tipo definido e a função do cálculo do comprimento (compare com a apresentada atrás...).

Tipos Parametrizados

Numa declaração de tipo é possível incluir parâmetros por variáveis de tipo. Como exemplo apresentam-se os seguintes tipos já estudados nas aulas teóricas:

```
data Maybe a = Nothing | Just a   -- ja' definido no Prelude

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

O primeiro tipo permite representar a *parcialidade* (funções que possam não estar definidas para todo o domínio) e o segundo é o tipo das *árvores binárias* (os nós contém informação de um dado tipo e duas sub-árvores).

Tarefa 1

1. Desenhe a árvore binária de inteiros representada pelo termo `arv`:

   ```
   arv = (Nodo 3 (Nodo 6 (Nodo 2 Vazia (Nodo 7 Vazia Vazia)) Vazia)
                 (Nodo 5 (Nodo 8 (Nodo 1 Vazia Vazia) (Nodo 4 Vazia Vazia))
                         (Nodo 23 Vazia Vazia) ) )
   ```

2. Defina uma função que conta o número de nodos de uma árvore binária.

3. Defina uma função que soma todos os valores que estão numa árvore binária de inteiros.

Tarefa 2

Considere as seguintes declarações de tipo:

```
type Nome = String
type Numero = Integer

data Curso = LMCC | LESI  deriving (Eq,Show)

data NotaPratica = Congelada Float
                 | AvContinua  Float
             deriving (Eq,Show)

data NotaTeorica = Faltou
                 | Teste1 Float
                 | Teste2 Float
             deriving (Eq,Show)
```

```
type Aluno = (Curso, Numero, Nome, NotaPratica, NotaTeorica)

type PP1PF = [Aluno]
```

1. Defina um valor do tipo `PP1PF` contendo a informação dos seguintes alunos:

Num.	Nome	Curso	Prática	Teórica
11111	José	LESI	12 (congelada)	16 (Teste 2)
22222	Maria	LMCC	16 (avaliação contínua)	14 (Teste 1)
33333	João	LESI	10 (avaliação contínua)	Faltou

2. Defina a função `procuraAluno :: Numero-> PP1PF -> Maybe Aluno` que, dado um número de um aluno e a informação de um conjunto de alunos, retorne a informação referente ao aluno com o número dado (se existir, retornando `Nothing` noutro caso).

3. Defina o tipo `NotaFinal` (um aluno pode "reprovar" ou "passar" com uma dada classificação).

4. Defina a função `notaFinal` que recebe informação sobre a nota prática e teórica de um aluno e calcula a sua nota final (relembre que o peso da parte prática é de 40% e não esqueça as notas mínimas).

5. Defina o tipo `Listagem` que contenha informação sobre o número, o nome e a nota final de um conjunto de alunos.

6. Defina a função `listaNotas :: PP1FP -> Listagem` produz a listagem com as notas finais de um conjunto de alunos.

Classes

Considere-se a seguinte função que testa se um elemento está contido numa lista:

```
elem e [] = False
elem e (x:xs) = if x==e then True else (elem e xs)
```

Esta função não depende do tipo de dados dos elementos da lista, mas requer que esse tipo de dados tenha definida a função que testa a igualdade entre valores desse tipo (`==`). Este facto é reflectido no tipo inferido pelo *Haskell* (neste caso seria `(Eq a) => a -> [a] -> Bool`) em que se condiciona a variável de tipo a pertencer à classe `Eq` — diz-se que `(Eq a)` é o *contexto* para o tipo apresentado).

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 *instâncias* dessa classe devem ter definido. No caso da classe `Eq`, os métodos são

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

Para tornar um tipo concreto instância de uma classe devemos definir os métodos respectivos. Ilustrando com um tipo muito simples:

```
data M = M1 | M2

instance Eq M where
  M1 == M1 = True
  M2 == M2 = True
  _  == _  = False
```

Dado que não é fornecida qualquer definição para o método (`/=`), o *Haskell* assume a definição *por omissão* fornecida na declaração da classe.

O teste de igualdade aqui definido corresponde à "igualdade sintáctica" (dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos iguais). Neste caso, a declaração de instância poderia ser substituída pela instrução "`deriving Eq`" aquando da declaração do tipo `M` (o *Haskell* encarregar-se-ia de derivar automaticamente a declaração da função de igualdade e respectiva declaração de instância). Mas é importante referir que muitas vezes a igualdade não é sintáctica: podemos dispor de dois valores distintos de um dado tipo que "representam" a mesma entidade, e como tal devem ser identificados quando comparados (veremos adiante um exemplo).

Existem diversas classes pré-definidas no *Haskell*. Classes que agrupam variedades de tipos numéricos (`Integral, Fractional, Floating, Num`); que refletem a habilidade para relacionar valores dos tipos (`Eq`, `Ord`); que disponibilizam funcionalidades uteis na manipulação dos valores dos tipos (`Show`, `Read`); que representam abstrações relevantes (`Monad`, `Functor`); etc.

Em rigor, as classes formam uma hiearquia já que é possível a uma classe declarar-se como *sub-classe* de outra. Um exemplo ocorre na classe `Ord` (cujos métodos são os operadores relacionais, como `(<), (<=),` etc.). Esta classe está declarada como uma sub-classe de `Eq` (referida atrás). Diz-se assim que `Ord` extende a classe `Eq` já que um tipo, para ser declarado como instância de `Ord`, tem de ser também decladado como instância de `Eq`. Esta dependencia é estabelecida por intermédio de um contexto apropriado na declaração da classe. No caso da classe `Ord` será:

```
classe (Eq a) => Ord a where
  (<=) :: a -> a -> a
  ...(etc)...
```

A classe `Show`

Uma classe importante na utilização do *Haskell* é a classe `Show`. Esta classe dispõe de métodos responsáveis por visualizar a representação dos valores. Quando o interpretador GHCI necessita visualizar um valor de um dado tipo (e.g. o resultado da avaliação de uma expressão) faz uso do método apropriado definido nesta classe.

O método mais importante desta classe (o método `show`) simplesmente produz uma representação textual do valor dado. Para o tipo `M` referido atrás, uma declaração de instância possível para a classe `Show` seria[3]:

```
instance Show M where
  show M1 = ">>M1<<"
  show M2 = ">>M2<<"
```

Também as declarações de instâncias podem ser condicionadas por contextos que imponham a pertença a determinadas classe. Por exemplo, uma declaração de instância `Show` para o tipo `Maybe a` poderia assumir a forma:

```
instance (Show a) => Show (Maybe a) where
  show Nothing = "Nothing"
  show (Just x) = "(Just "++(show x)++")"
```

que reflete o facto de ser necessário saber visualizar valores do tipo `a` para visualizar valores do tipo `Maybe a`.

Tarefa 3

1. Defina o tipo `NotaFinal` que definiu na tarefa 2, como instãncia de classe Show, de modo que a informação impressa seja do género: "aprovado com ... valores" ou "reprovado".

[3]Esta declaração difere da representação "standard" que seria produzida pela adição da instrução `deriving Show` na declaração de tipo.

2. Apague a instrução `deriving Eq` da declaração do tipo `(ArvBin a)` e defina `(ArvBin a)` como instância da classe Eq.

Projecto Prático

A última parte deste enunciado descreve um projecto de programação em Haskell que deverá ser realizado com base em toda a matéria abrangida por esta ficha, bem como pelas fichas de trabalho anteriores.

Este projecto representa apenas parte de uma aplicação em Haskell, que será completada no âmbito da ficha de trabalho número 5.

Considere um campeonato de automobilismo com as seguintes características:

- Num campeonato estão inscritas N equipas, cada qual concorrendo com M carros.
- O campeonato consiste num conjunto de C provas, cada qual com um prémio total em dinheiro D.
- Em cada prova apura-se o tempo total que cada carro demora a completar o percurso ou, no caso de isso não acontecer, o tempo que o carro esteve em prova até desistir.
- A cada participante numa prova é atribuída uma pontuação P que depende da sua classificação, e pode ser nula.
- A cada participante é também atribuído um prémio em dinheiro, que será uma percentagem do prémio total D, variando também essa percentagem de acordo com a classificação.
- A atribuição de pontos segue o mesmo critério em todas as provas.
- A atribuição de prémios em dinheiro varia de prova para prova.
- Não é possível haver empates na classificação. Para resolver este problema sugere-se a aplicação de um critério de desempate que seja utilizado numa prova real.

Todos os tipos que implementar deverão ser enquadrados dentro do esquema de classes pré-definidas do Haskell quando isso permitir alcançar melhor legibilidade e maior elegância no código produzido. De particular importância são as classes `Eq` , `Ord` e `Show` .

Pretende-se que implemente, no mínimo, as seguintes funcionalidades:

1. Registo de uma prova (participantes na prova, resultados obtidos por cada participante, critério de atribuição de prémios nessa prova, ...).
2. Cálculo da pontuação a atribuir a cada participante numa prova.
3. Cálculo da pontuação a atribuir a cada equipa numa prova.
4. Cálculo do prémio a atribuir a cada participante numa prova.
5. Listagem dos participantes numa prova, ordenados por classificação.
6. Listagem das equipas participantes numa prova, ordenadas por classificação.
7. Cálculo da pontuação global de cada participante.
8. Cálculo da pontuação global de cada equipa.
9. Cálculo do prémio total em dinheiro de um participante num campeonato.
10. ...