Trabalho Prático nº 2

Métodos de Programação I

2005/2006

1 Introdução

Este é o enunciado do segundo trabalho prático de Métodos de Programação I, e tem por objectivo desenvolver uma pequena aplicação para efectuar simplificações e provas sobre expressões *point-free*. O desenvolvimento do trabalho deverá ser dividido em duas componentes:

- Uma biblioteca genérica para manipulação de árvores sintáticas, com ênfase na utilização de padrões de recursividade, e na implementação de operações de reescrita. Este compomente tem um peso de 12 valores na nota e é descrito nas secções 2 e 3.
- Utilização dessa biblioteca para implementar uma estratégia de simplificação que permita demonstrar alguns factos triviais sobre expressões *point-free*. Este componente é brevemente descrito na secção 4.

Para apresentação do trabalho, pretende-se que o relatório e o código constituam um único documento no estilo *literate programming*. Nesse estilo, a documentação e explicação dos diversos fragmentos de código que compõe uma aplicação constitui um texto coerente, podendo assim ser lido tal como se lê um qualquer ensaio literário (recomenda-se uma pesquisa do termo no *google* para aceder a informação adicional e exemplos). A linguagem de programação *Haskell* disponibiliza suporte a esse estilo de programação (ver exemplo em anexo).

2 Expressões genéricas

Considere o seguinte tipo de dados Haskell, que permite codificar expressões genéricas que envolvam variáveis.

```
data Exp v c = Term c [Exp v c] | Var v
```

Nesta declaração `v` representa o tipo dos identificadores das variáveis e `c` o tipo dos construtores da linguagem que pretendemos modelar. Como o número de argumentos pode variar de acordo com o construtor é usada uma lista. Por exemplo, a seguinte instanciação deste tipo, permite representar expressões *point-free* apenas com produtos e onde as variáveis são identificadas por um caracter (a utilização destas será vista mais tarde).

```
type Pointfree = Exp Char PF

data PF = Comp | Id | Split | Prod | Fst | Snd

swap :: Pointfree
swap = Term Split [Term Snd [],Term Fst []]

assocr :: Pointfree
assocr = Term Split [Term Comp [Term Fst [], Term Fst []],
                     Term Prod [Term Snd [], Term Id []]]
```

Para o tipo de dados `Exp` pretende-se que se implementem as seguintes funcionalidades:

- Funções que permitam programar com este tipo de dados no estilo *point-free*: `in`, `out`, `cata`, `ana` e `hylo`.
- Instância para a classe `Show`.
- Verificação da boa formação de uma expressão, ou seja, que o mesmo construtor é sempre aplicado ao mesmo número de argumentos.
- Definição de um tipo adequado para representar a noção de subexpressão, e enumeração de todas as subexpressões de uma determinada expressão.
- Substituição de uma determinada subexpressão por outra.

Naturalmente, será valorizada a utilização dos padrões de recursividade definidos no ponto 1 para definir qualquer outra função sobre expressões.

3 Reescrita

Sobre um determinado tipoo de expressões é possível definir equações que definem uma relação de igualdade sobre as mesmas. Estas podem ser definidas simplesmente como um par de expressões. Por exemplo, no caso de expressões *point-free* podemos definir a lei de fusão para produtos da seguinte forma.

```
fusao_prod :: (Pointfree, Pointfree)
fusao_prod = (Term Comp [Term Split [Var 'f', Var 'g'], Var 'h'],
              Term Split [Term Comp [Var 'f', Var 'h'],
                          Term Comp [Var 'g', Var 'h']])
```

Uma lei pode ser usada para simplificar uma expressão através de um processo de reescrita. Para tal é necessário definir a orientação em que vamos aplicar a equação, por forma a obter uma regra de reescrita. A única restrição que se deve verificar é que todas variáveis que aparecem na expressão resultante devem aparecer na expressão original. Vamos assumir que as equações estão sempre correctamente orientadas da esquerda para a direita.

Para aplicar uma regra de reescrita é necessário encontrar (pelo menos) uma subexpressão que faça *matching* com o lado esquerdo, ou seja, é necessário encontrar uma substituição para as variáveis que torne o lado esquerdo da regra igual a essa subexpressão. Se tal se verificar é possível substituir a subexpressão pelo lado direito da regra após lhe aplicar a substituição encontrada. A seguinte função implementa um algoritmo de *matching* entre uma expressão com variáveis e uma expressão sem variáveis.

```
match :: (Eq v, Eq c) => Exp v c -> Exp v c -> Maybe (Subst v c)
match l r | null (vars r) = execStateT (aux (l,r)) []
    where aux (Var v, e) = do s <- get
                              z <- lift (update s v e)
                              put z
          aux (Term _ _, Var _) = fail "As expressões não fazem matching!"
          aux (Term c l, Term d m) =
              do unless (c==d) $ fail "As expressões não fazem matching!"
                 sequence_ $ map aux (zip l m)
match _ _ = fail "A segunda expressão não pode ter variáveis!"
```

Esta definição usa o *monad* estado e para que funcione correctamente deverá importar a biblioteca respectiva. Também deverá implementar as funções em falta, nomeadamente a função `vars` que determina as variáveis existentes numa expressão e a função `update` com tipo

```
update :: (Eq v, Eq c) => Subst v c -> v -> Exp v c -> Maybe (Subst v c)
```

que dada uma substituição, uma variável, e uma expressão para atribuir a essa variável, acrescenta à substituição actual essa variável e respectiva expressão. Note que tal só poderá ocorrer com sucesso caso não tenha sido previamente atribuída à mesma variável uma expressão diferente. Obviamente deverá definir um tipo adequado para substituições e uma função que aplica uma substituição a uma expressão com variáveis.

Com toda esta "maquinaria" poderá finalmente definir uma função que, dada uma regra de reescrita, reescreve uma expressão devolvendo todas as possíveis expressões resultantes.

```
reescreve :: (Eq v, Eq c) => (Exp v c, Exp v c) -> Exp v c -> [Exp v c]
```

4 Calculador *Point-free*

Neste componente deverá utilizar a biblioteca anteriormente definida para implementar um simplificador de expressões *point-free*. Para tal, deverá definir estratégias de simplificação, ou seja, funções que reescrevem uma expressão de acordo com um determinado algoritmo. Naturalmente, a estratégia mais simples é a que apenas tenta aplicar uma lei para reescrever uma única vez uma expressão. Muitas outras poderão ser definidas, como por exemplo, uma que dada uma lista de regras tenta reescrever a expressão com a primeira que se aplique.

Também pode ser útil definir combinadores de estratégias, ou seja, funções que dada uma ou mais estratégias as combinam de acordo com um determinado algoritmo para formar uma nova estratégia de simplificação. Um desses combinadores poderia, por exemplo, tentar aplicar repetidamente uma estratégia até que a sua aplicação não seja mais possível (note que, para tal é conveniente que as estratégias tenham a possibilidade de falhar).

Seria interessante se as estratégias guardassem os passos intermédios e o nome las leis que foram usadas durante a simplificação, para que no final possa ser mostrado ao utilizador os detalhes do cálculo efectuado no estilo normalmente usado nas aulas.

A *Literate Programming* em *Haskell*

A linguagem *Haskell* inclui algum suporte a *literate programming* ao estabelecer que o compilador/interpretador, perante um ficheiro *literate Haskell* (com extensão `.lhs`), ignora tudo que não se encontre no ambiente `code` do LaTeX (i.e. entre `\begin{code}` e `\end{code}`). A título de exemplo, considere-se um ficheiro `ex.lhs` com o seguinte conteúdo:

```
\documentclass{article}
\usepackage[portuges]{babel}
\usepackage[latin1]{inputenc}
\usepackage{a4wide}
\usepackage{fancyvrb}

\DefineVerbatimEnvironment{code}{Verbatim}{fontsize=\small}

\begin{document}

\title{Exemplo de \emph{Literate Programming}}
\author{Métodos de Programação I}
\date{2005/2006}

\maketitle
```

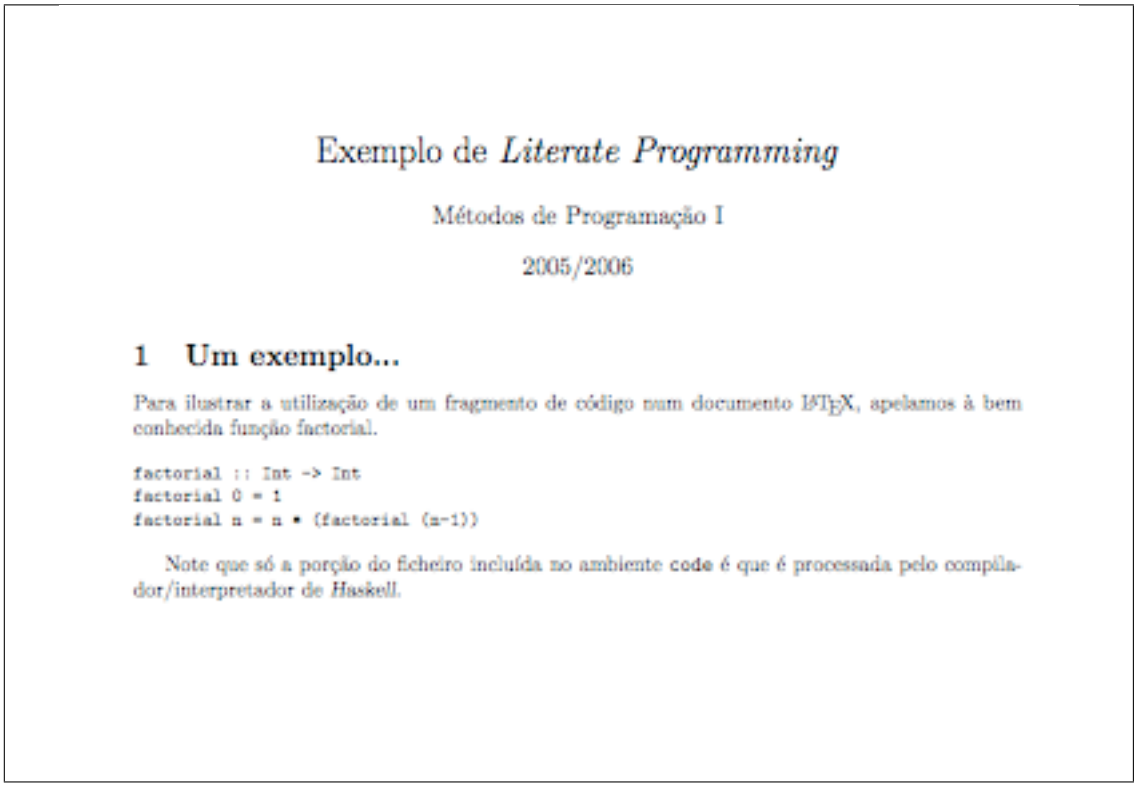
```
\section{Um exemplo...}

Para ilustrar a utilização de um fragmento de código num documento
\LaTeX, apelamos à bem conhecida função factorial.

\begin{code}
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
\end{code}

Note que só a porção do ficheiro incluída no ambiente \texttt{code} é
que é processada pelo compilador/interpretador de \textsl{Haskell}.
\end{document}
```

Ao invocar "`ghci ex.lhs`" iniciaríamos uma sessão no interpretador ghci com a função `factorial` definida. Já o processamento do mesmo ficheiro pelo LaTeX (através do comando "`latex ex.lhs`") permitiria obter um documento como o que se segue:



Exemplo de *Literate Programming*

Métodos de Programação I

2005/2006

1 Um exemplo...

Para ilustrar a utilização de um fragmento de código num documento LaTeX, apelamos à bem conhecida função factorial.

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

Note que só a porção do ficheiro incluída no ambiente `code` é que é processada pelo compilador/interpretador de *Haskell*.