2º Trabalho Prático

Métodos de Programação I
LESI/LMCC
Universidade do Minho

Ano Lectivo de 2002/2003

1 Preâmbulo

Este trabalho deve ser realizado por grupos com um máximo de três alunos. O relatório do trabalho deve ser entregue até ao dia 26 de Novembro de 2002 na Recepção do *Departamento de Informática* (ext. 4430). Simultaneamente, deverá ser realizada a entrega electrónica, de acordo com as instruções que serão muito brevemente divulgadas na página da disciplina (WWW).

2 O problema

Neste trabalho pretende-se explorar a capacidade expressiva do conceito de *mónada* em Haskell e do mecanismo de *classes* da linguagem. O tema escolhido para esse efeito é o da construção de uma pequena calculadora de *expressões aritméticas* tipo `bc` (faça `man bc` numa "shell" em Linux para saber do que se trata). Essa calculadora deverá, em modo de comando,

- validar uma expressão aritmética, que pode conter variáveis como, por exemplo `2 * (x + 1)`
- convertê-la para código máquina
- interpretar de imediato esse código, produzindo como resultado o valor da expressão dada.

O código máquina escolhido é o MSP, um "assembler" muito simples cujo manual de utilização está disponível.

Tal como é habitual nos trabalhos desta disciplina, fornece-se de seguida um "kit" para arranque do projecto. Este enunciado concluir-se-á com sugestões para valorização do trabalho a realizar.

3 Material fornecido como "kit" para o projecto

Entre na página da disciplina e descarregue, descomprimindo-o (*eg.* via `unzip`), o ficheiro *`mpi0203mp.zip`* que contém o respectivo material pedagógico. Para além de outros ficheiros relevantes para a disciplina, deverá obter:

1. `mpi0203t2.lhs` - trata-se do ficheiro que está a ler neste momento, escrito em "*literate* HASKELL". Isto significa que:

   - se o carregar no HUGS, este interpretador carregará o código HASKELL nele contido e interpretá-lo-á. Sugestão: invoque o HUGS e experimente `:l mpi0203t2.lhs`. Uma vez iniciada a sessão, escreva

     ```
     Main> exp2msp "2 * ( 3 + 4 )"
     ```

     Deverá obter a resposta

```
PUSH 2
PUSH 3
PUSH 4
ADD
MUL
```

- se o processar via LaTeX (ou PDFLaTeX) obterá este mesmo documento em *PDF*. Sugestão: experimente

  ```
  latex mpi0203t2.lhs
  dvips -o mpi0203t2.ps mpi0203t2
  ps2pdf mpi0203t2.ps
  ```

  ou, mais simplesmente,

  ```
  pdflatex mpi0203t2.lhs
  ```

  Se não está habituado a LaTeX: em LINUX, faz parte da distribuição standard e é só experimentar; em Windows, sugere-se a instalação de *MiKTeX* (`http://www.miktex.org/`).

2. `mpi0203.sty` - trata-se de um ficheiro importado por `mpi0203t2.lhs`, contendo funções de processamento de texto expressas em sintaxe LaTeX (como poderá ver, é normal programar *funcionalmente* em LaTeX).
3. `mpi0203t2.pdf` - trata-se do resultado do processamento de `mpi0203t2.lhs`, fornecido para sua conveniência, caso não tenha LaTeX acessível.
4. `MonadicPComb.hs` - módulo auxiliar de reconhecimento sintático.
5. `MSPLPMAN.pdf` - manual de MSP.

O trabalho consiste em editar o ficheiro `mpi0203t2.lhs`, acrescentando-lhe não só texto seu (por exemplo, preâmbulo, conclusões, detalhes de execução, pistas para trabalho futuro, etc.) mas todo o código HASKELL que vai desenvolver, introduzindo-o pela ordem que lhe parecer mais natural.

4 O que se pretende

Comece por analisar o código Haskell dado em anexo (pág.3), que é fornecido como primeira abordagem aos requisitos do trabalho.

Repare que o tradutor `exp2msp` está dividido em duas partes. Em primeiro lugar o reconhecedor de expressões ("parser") que, baseado em mónadas, gera a árvore sintática da cada expressão ou um erro sintático (secção A.3). Em segundo lugar, a conversão para MSP (secção A.4). Esta é feita de forma subtil declarando o tipo `Exp` de tais árvores sintáticas como instância da classe `Show`. Assim, basta por exemplo escrever

```
Main> exp2msp "2 * (x + 1)"
```

que a correspondente sequência de instruções MSP é de imediato dada pelo `show` do resultado:

```
PUSH 2
LOAD "x"
PUSH 1
ADD
MUL
```

Note-se que estas instruções não são interpretadas. Ora o que se pretende é passá-las a um interpretador de MSP por forma a obter como resultado o *valor* da expressão dada. Assim, há que:

- Mudar a geração de código de forma a produzir não uma *string* (como está a ser feito agora) mas uma lista de instruções a passar a um interpretador de MSP.

- Construir esse interpretador como uma máquina monádica baseado na mónada de estado. Neste contexto, a interpretação de uma lista de instruções MSP pode ser realizada directamente com `sequence`.
- Enriquecer o parser (e respectivo tipo das árvores) para tratar atribuições e sequenciação, eg. `x = 3 ; 2 * (x + 1)`.

Será conveniente conhecer bem MSP, pelo que se recomenda a leitura do respectivo manual de utilização, fornecido no "kit" do projecto. Para mais informação, poderá ser ainda explorado o WINMSP, um IDE para MSP desenvolvido para Windows disponível a partir de `http://www.ipb.pt/~rufino/past.html`.

5 Sugestões para Valorização

- Estender o processador de expressões por forma a emular mais fielmente a shell `bc` *(arbitrary precision calculator language)* da GNU.
- Enriquecer o estado do interpretador de MSP com informação extra para efeitos de "debug" (buscar inspiração em WINMSP)
- Tornar o parser menos permissivo (por exemplo, `"x y"` é traduzido em `LOAD "x"`, quando deveria dar erro sintático).

A Anexo

O código Haskell fornecido neste anexo é baseado no módulo

```
import MonadicPComb
```

que deverá ser cuidadosamente estudado.

A.1 Syntaxe abstracta para expressões aritméticas

```
type ExpArit = Exp String Int BArit UArit
data BArit = Mais | Menos | Div | Mult
data UArit = Simetrico | Inverso
```

onde

```
data Exp vars values bop uop
     = C values
     | V vars
     | Bop (Exp vars values bop uop) bop (Exp vars values bop uop)
     | Uop uop (Exp vars values bop uop)
```

A.2 Sintaxe concreta para expressões aritméticas

```
{- Gramatica das expressoes aritmeticas

   ExpArit = Termo  | Termo + ExpArit | Termo  - ExpArit
   Termo  = Factor | Factor * Termo | Factor / Termo
   Factor = ( ExpArit ) | - Factor | Constante | Variavel
-}
```

A.3 Processador (monádico) da sintaxe concreta para expressões aritméticas

Consiste no comando

```
exp2msp = parse parseExpArit
```

o qual se baseia na função de "parsing"

```
parse :: Parse a -> String -> Error a
```

definida em `MonadicPComb.hs`, a qual recebe como primeiro argumento o "parser" `parseExpArit` que se segue:

Parser de Expressões

```
parseExpArit = do  f1 <- parseTermo ;
                    try [ (do  parseConst "+" ;
                               f2 <- parseExpArit ;
                               return (Bop f1 Mais f2)
                           ),(do  parseConst "-" ;
                                  f2 <- parseExpArit ;
                                  return (Bop f1 Menos f2)
                           ),(return f1)]
```

Parser de Termos

```
parseTermo = do  p1 <- parseFactor ;
                  try [ (do  parseConst "*" ;
                             p2 <- parseTermo ;
                             return (Bop p1 Mult p2)
                         ),(do  parseConst "/" ;
                                p2 <- parseTermo ;
                                return (Bop p1 Div p2)
                         ),(return p1)]
```

Parser de Factores

```
parseFactor = try [ (parsePCurvos parseExpArit
                    ),(do  parseConst "-" ;
                           p <- parseFactor ;
                           return (Uop Simetrico p)
                    ),(do  n <- parseInt ;
                           return (C n)
                    ),(do  v <- parseString ;
                           return (V v)
                      )]
```

A.4 Geração de código MSP (via Show)

```
instance (Show a, Show b, Show c, Show d) => Show (Exp a b c d) where
  showsPrec n (C v) =
         showString "PUSH " . showsPrec n v . showLine ""
  showsPrec n (V v) =
         showString "LOAD " . showsPrec n v . showLine ""
  showsPrec n (Bop e1 b e2) = showsPrec n e1 .
                              showsPrec n e2 .
                              showsPrec n b
  showsPrec n (Uop u e) = showsPrec n e . showsPrec n u

instance Show UArit where
   showsPrec _ Simetrico = showLine "SSIGN"
   showsPrec _ Inverso = showLine "INV"

instance Show BArit where
   showsPrec _ Mais = showLine "ADD"
   showsPrec _ Menos = showLine "SUBT"
   showsPrec _ Div = showLine "DIV"
   showsPrec _ Mult = showLine "MUL"
```