

# Métodos de programação III

LMCC & LESI, Universidade do Minho

Ano lectivo 2004/2005

Ficha Teórico-Prática N°11

Este texto está escrito em **literate Haskell**. Isto é, pode ser interpretado como um documento L<sup>A</sup>T<sub>E</sub>X ou como um puro programa na linguagem Haskell. Responda às perguntas sobre Haskell neste próprio ficheiro para assim produzir o programa e a sua documentação.

## 1 Gramáticas Independentes de Contexto em Haskell

1.1 Defina combinadores de parsing para as seguintes linguagens:

1. A linguagem usual para definir strings nas linguagens de programação. Uma string é uma sequência de caracteres iniciada e terminada pelo carácter ".

```
import Parser -- Ficha10.lhs
string :: Parser Char [Char]
string = sf <$>symbol '"' <*>zeroOuMais (satisfy (≠ '"')) <*>symbol '"'
  where sf a b c = a : b ++ [c]
```

2. Usualmente os espaços são consumidos e ignorados durante o parsing de um texto. Re-escreva o combinador `token` de modo a consumir os espaços que possam ocorrer antes do token. Utilize para tal, a função de parsing feita na aula anterior que define a ocorrência de zero ou mais espaços: `zeroOuMaisEspacos`.

```
token' :: [Char] → Parser Char [Char]
token' t = sf <$>zeroOuMaisEspacos <*>token t
  where sf a b = b
```

1.2 Considere de novo a linguagem HTML e a sua notação para descrever tabelas. A gramática independente do contexto para definir a estrutura concreta desta linguagem foi definida na ficha-teórica nº9. A gramática abstracta e o tipo de dados isomórfico foi

definido na ficha-teórica nº10. Re-escreva a gramática concreta da linguagem com combinadores de Parsing. O resultado do parser é a árvore construída usando os construtores do tipo de dados induzidos pela gramática abstracta. De seguida apresenta-se a gramática concreta para a linguagem das tabelas HTML:

$$\begin{aligned}
 G &= (T, N, S, P), \text{ com} \\
 T &= \{ \text{"<table>"}, \text{"</table>"}, \text{"<tr>"}, \text{"</tr>"}, \text{"<td>"}, \text{"</td>"}, \text{texto} \} \\
 N &= \{ \text{Tabela}, \text{Linhas}, \text{Linha}, \text{Colunas}, \text{Coluna}, \text{Elemento} \} \\
 S &= \text{Tabela} \\
 P &= \{ \text{Tabela: Tabela} \rightarrow \text{"<table>"} \text{Linhas} \text{"</table>"} \\
 & \quad , \text{NoLinhas: Linhas} \rightarrow \\
 & \quad , \text{CLinhas:} \quad \quad \quad | \quad \text{Linha Linhas} \\
 & \quad , \text{Linha: Linha} \rightarrow \text{"<tr>"} \text{Colunas} \text{"</tr>"} \\
 & \quad , \text{NoCols: Colunas} \rightarrow \\
 & \quad , \text{CCols:} \quad \quad \quad | \quad \text{Coluna Colunas} \\
 & \quad , \text{Coluna: Coluna} \rightarrow \text{"<td>"} \text{Elemento} \text{"</td>"} \\
 & \quad , \text{Texto: Elemento} \rightarrow \text{texto} \\
 & \quad , \text{TabAnin:} \quad \quad \quad | \quad \text{Tabela} \\
 & \quad \}
 \end{aligned}$$

E o tipo de dados calculado a partir da gramática abstracta está já escrito na solução do exercício.

```

data Tabela    = Tabela Linhas
  deriving Show
data Linhas    = NoLinhas
  | CLinhas Linha Linhas
  deriving Show
data Linha     = Linha Colunas
  deriving Show
data Colunas   = NoCols
  | CCols Coluna Colunas
  deriving Show
data Coluna    = Coluna Elemento
  deriving Show
data Elemento = Texto String
  | TabAnin Tabela
  deriving Show
  
```

Aplicando as regras de conversão de gramáticas independentes do contexto concretas para combinadores de parsing obtém-se o seguinte parser:

```

tabela :: Parser Char Tabela
tabela = tabela <$> token' "<table>" <*> linhas <*> token' "</table>"
  where tabela a b c = Tabela b
  
```

```

linhas    = succeed nolinhas
          <|> clinhas <$> linha <*> linhas
  where nolinhas    = NoLinhas
        clinhas a b = CLinhas a b
linha      = linha <$> token' "<tr>" <*> colunas <*> token' "</tr>"
  where linha a b c = Linha b
colunas    = succeed nocols
          <|> ccols <$> coluna <*> colunas
  where nocols     = NoCols
        ccols a b = CCols a b
coluna     = coluna <$> token' "<td>" <*> elemento <*> token' "</td>"
  where coluna a b c = Coluna b
elemento   = texto <$> string
          <|> tabanin <$> tabela
  where texto a     = Texto a
        tabanin a  = TabAnin a

```

A função que lê uma tabela HTML de um ficheiro (com nome `inputTabela`) e escreve o resultado na saída apresenta-se a seguir.

```

runparser_table :: [Char] → Maybe Tabela
runparser_table inp =
  case tabela inp of
    []      → Nothing
    (x : xs) → case x of
      (ast, _) → (Just ast)
      _       → Nothing
showResTable :: Maybe Tabela → String
showResTable (Nothing)    = ""
showResTable (Just tabela) = show tabela
mainTabela :: IO ()
mainTabela = do s ← readFile "inputTabela"
              putStr (showResTable (runparser_table s))

```

**1.3** Considere a gramática independente do contexto que define uma representação concreta de expressões aritméticas.

$$\begin{array}{l}
G = (T, N, S, P), \text{ com} \\
T = \{+, -, *, /, (, ), \text{num}\} \\
N = \{Expr\} \\
S = Expr
\end{array}
\qquad
\begin{array}{l}
P = \{ \\
\text{Soma} : Expr \rightarrow Expr \text{ ' + ' } Expr \\
, \text{Mul} : \quad | Expr \text{ ' * ' } Expr \\
, \text{Div} : \quad | Expr \text{ ' / ' } Expr \\
, \text{Sub} : \quad | Expr \text{ ' - ' } Expr \\
, \text{Prio} : \quad | \text{' ( ' } Expr \text{ ' ) ' } \\
, \text{Num} : \quad | \text{num} \\
\}
\end{array}$$

*Será possível converter directamente esta gramática em combinadores de parsing? Porquê? Defina uma gramática concreta equivalente, para a qual podemos construir combinadores de parsing. Determine a gramática abstracta, escreva os tipos de dados abstractos do Haskell e defina o catamorfismo sobre esse tipo de dados. Implemente os combinadores de parsing de modo a construírem a estrutura de dados que representa a expressão aritmética processada.*

Não é possível converter esta gramática concreta em combinadores de parsing pois esta gramática é recursiva à esquerda. Assim, temos de construir primeiro uma gramática equivalente (ie, que define a mesma linguagem) mas que não apresenta recursividade à esquerda. Isto é feito neste caso tendo em conta a prioridade dos operadores: o operador \* tem mais prioridade que o operador +.

$$\begin{array}{l}
G' = (T', N', S', P'), \text{ com} \\
T' = \{+, -, *, /, (, ), \text{num}\} \\
N' = \{expr\} \\
S' = expr
\end{array}
\qquad
\begin{array}{l}
P' = \{ \\
p_0 : expr \rightarrow \text{termo ' + ' } expr \\
, p_1 : \quad | \text{termo ' - ' } expr \\
, p_2 : \quad | \text{termo} \\
, p_3 : \text{termo} \rightarrow \text{factor ' * ' } \text{termo} \\
, p_4 : \quad | \text{factor ' / ' } \text{termo} \\
, p_5 : \quad | \text{factor} \\
, p_6 : \text{factor} \rightarrow \text{num} \\
, p_7 : \quad | \text{' ( } expr \text{ ' ) ' } \\
\}
\end{array}$$

Vamos agora aplicar as regras de conversão de gramáticas independentes de contexto em programas Haskell. Começamos por representar a gramática abstracta em Haskell e o catamorfismo que o tipo induz:

```

data Expr = Soma Expr Expr
          | Mul   Expr Expr
          | Div   Expr Expr
          | Sub   Expr Expr
          | Prio  Expr
          | Num   Numero
deriving Show

type Numero = Int

cataExpr (soma, mul, div, sub, prio, num) = cata
  where cata (Soma l r) = soma (cata l) (cata r)
        cata (Mul l r)  = mul  (cata l) (cata r)
        cata (Div l r)  = div  (cata l) (cata r)

```

```

cata (Sub l r) = sub (cata l) (cata r)
cata (Prio e) = prio (cata e)
cata (Num n) = num n

```

E agora construímos o parser baseado em combinadores. As funções semânticas são os construtores dos tipos induzidos pela gramática abstracta:

```

expr :: Parser Char Expr
expr = p_0<$>termo<*>symbol '+'<*>expr
      <|>p_1<$>termo<*>symbol '-'<*>expr
      <|>p_2<$>termo
  where p_0 a b c = Soma a c
        p_1 a b c = Sub a c
        p_2 a = a

termo :: Parser Char Expr
termo = p_3<$>factor<*>symbol '* '<*>termo
      <|>p_4<$>factor<*>symbol '/'<*>termo
      <|>p_5<$>factor
  where p_3 a b c = Mul a c
        p_4 a b c = Div a c
        p_5 a = a

factor :: Parser Char Expr
factor = p_6<$>num
      <|>p_7<$>symbol '('<*>expr<*>symbol ')'
  where p_6 a = Num a
        p_7 a b c = b

num :: Parser Char Int
num = f<$>umOuMais digito
  where f a = read a

runparser_exp :: [Char] -> Maybe Expr
runparser_exp inp =
  case expr inp of
    [] -> Nothing
    (x : xs) -> case x of
      (ast, _) -> (Just ast)
      _ -> Nothing
  -- interpretador :: [Char] -> a
  interpretador input = cataExp ((+), (*), (+), (-), id, id) e
  where (Just e) = runparser_exp input

```

#### 1.4 Construa as seguintes funções:

1. Defina uma função `showLateX` para o tipo de dados `Tabela` de modo a "mostrar" as tabelas abstractas na notação (concreta) do `LaTeX`. Use a função `show` para mostrar o resultado do parser e assim obter um conversor de tabelas `HTML` para tabelas em `LaTeX`.
2. Considere os parsers construídos nos dois exercícios anteriores. Generalize esse parsers de modo a ser possível instanciar esses parsers com diferentes semânticas. Isto é, parameterize esses parsers com as funções que são aplicadas durante o reconhecimento sintático.

O parser genérico para as gramáticas de expressões é:

```
gen_parser_exp (soma, sub, ptermo, mul, div, pfactor, pnum, prio) = expr
  where
    expr = soma <$>termo<*>symbol '+'<*>expr
          <|>sub <$>termo<*>symbol '-'<*>expr
          <|>ptermo<$>termo
    termo = mul <$>factor<*>symbol '* '<*>termo
           <|>div <$>factor<*>symbol '/'<*>termo
           <|>pfactor<$>factor
    factor = pnum<$>num
            <|>prio <$>symbol '('<*>expr<*>symbol ')'
```

Este parser pode ser parameterizado com os operadores aritméticos pré-definidos em Haskell de modo a calcular o resultado da expressão.

```
interpretaExp = gen_parser_exp (\a b c → a + c
                                , \a b c → a - c
                                , id
                                , \a b c → a * c
                                , \a b c → a * c
                                , id
                                , id
                                , \a b c → b
                                )
```

E executando o interpretador em Hugs temos:

```
Main> interpretaExp "3+4*5"
[(23,""),(7,"*5"),(3,"+4*5")]
```

O parser pode também ser parameterizado com as funções construtoras do tipo `Expr` de modo a construir a árvores abstracta:

```
concrete2Ast = gen_parser_exp ( $\lambda a b c \rightarrow$  Soma a c  
    ,  $\lambda a b c \rightarrow$  Sub a c  
    , id  
    ,  $\lambda a b c \rightarrow$  Mul a c  
    ,  $\lambda a b c \rightarrow$  Div a c  
    , id  
    , Num  
    ,  $\lambda a b c \rightarrow$  Prio b  
    )
```

E executando o interpretador em Hugs temos:

```
Main> concrete2Ast "3+4*5"  
[(Soma (Num 3) (Mul (Num 4) (Num 5)),""),(Soma (Num 3) (Num 4),"*5"),(Num 3,"+4*5")]
```

A função catamórfica pode ser agora acoplada a esta função para calcular o valor da expressão.