Métodos de programação III

LMCC & LESI, Universidade do Minho

Ano lectivo 2004/2005

Ficha Teórico-Prática Nº10

Este texto está escrito em **literate Haskell**. Isto é, pode ser interpretado como um documento LaTeX 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 *Considere o tipo de dados* `Parser`*, que expressa o comportamento de um parser (recursivo descendente), e as funções* `symbol` *e* `satisfy`*.*

```
-- Combinadores de Parsing em Haskell
--
-- Métodos de Programação III
-- Universidade do Minho
-- 2001/2002
--
module Parser where
infixl 3<|>; infixl 4<*>
type Parser simbolo resultado = [simbolo] -> [(resultado,[simbolo])]
symbol :: Eq a => a -> [a] -> [(a,[a])]
symbol _ []                   = []
symbol s (x:xs) | x == s      = [(s,xs)]
                | otherwise   = []
satisfy :: (s -> Bool) -> Parser s s
satisfy p []                  = []
satisfy p (x:xs) | p x        = [(x,xs)]
                 | otherwise  = []
```

Responda às seguintes perguntas:

1. *Defina o tipo da função* `symbol` *em termos do tipo* `Parser`*.*

2. *Defina "parsers"para reconhecer os seguintes símbolos e classes de símbolos:*

   (a) *O parser* `simboloA` *que processa o símbolo* `A`*.*

   (b) *O parser* `espaco` *que processa o símbolo espaço. Expresse este parser em termos da função* `symbol` *e* `satisfy`*.*

   (c) *O Parser* `digito` *que processa um símbolo se este pertencer à classe dos digitos.*

   (d) *O Parser* `letraMaiusc` *que processa um símbolo se este pertencer à classe das letras maiúsculas.*

   $digito = satisfy\ isDigit$
   $letra = \ satisfy\ isAlpha$

3. *Generalize a função* `symbol` *de modo a processar sequências de símbolos, usualmente chamadas tokens de uma linguagem, e não um símbolo apenas. Uma execução desta função apresenta-se a seguir:*

```
Main> token "while" "while (x>0) { ... }"
[("while"," (x>0) { ... }")]
```

$$\begin{aligned} &token :: Eq\ a \Rightarrow [a] \rightarrow Parser\ a\ [a] \\ &token\ t\ inp \mid take\ n\ inp \equiv t = [(t, drop\ n\ inp)] \\ &\qquad\qquad \mid otherwise = \quad [] \\ &\quad \textbf{where}\ n = length\ t \end{aligned}$$

2 Os Combinadores de Parsing

O tipo `Parser` foi definido de modo a permitir que os parser básicos definidos anteriormente possam ser facilmente combinados de modo a formar parser mais poderosos. Esta é precisamente a ideia dos combinadores de parsing: funções de ordem superior que combinam os parsers que recebem como argumento e produzem um parser mais poderoso.

$$\begin{aligned} &succeed :: a \rightarrow Parser\ s\ a \\ &succeed\ r\ xs = [(r, xs)] \end{aligned}$$

$$\begin{aligned} &(<|>) :: Parser\ s\ a \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a \\ &(p<|>q)\ xs = p\ xs +\!\!+ q\ xs \end{aligned}$$

2.1 *Re-escreva o parser* `expaco` *de modo a processar os símbolos espaço, tab, e newline.*

$espaco = symbol$ `' '` $<|> symbol$ `'\n'` $<|> symbol$ `'\t'`

2.2 *Considere a seguinte gramática independente do contexto* $G = (\{"while","decl","if"\}, \{X\}, X, P)$, *com*

$$
\begin{array}{lllll}
P = & \{ & P0: & X \rightarrow & "while" \\
 & , & P1: & \mid & "decl" \\
 & , & P2: & \mid & "if" \\
 & , & P3: & \mid & \epsilon \\
 & \} & & &
\end{array}
$$

Escreva uma função de parsing **`parserX`** *que modela o parsing da linguagem definida por* G. *Qual o tipo deste parser? Quantas soluções produz o parser ao processar a seguinte entrada* **`"decl x;"`**. *Porquê?*

```
parserX = token "while"
        <|>token "decl"
        <|>token "if"
        <|>succeed ""
```

2.3 *O combinador* **`<*>`** *expressa a ocorrência de uma sequência de símbolos no lado direito das produções. O parser* **`<$>`** *permite alterar o valor do resultado de um parser.*

```
(<*>) :: Parser s (a → r) → Parser s a → Parser s r
(p<*>q) inp = [(f v, zs)
                | (f, ys) ← p inp
               , ( v, zs) ← q ys
               ]

(<$>) :: (a → r) → Parser s a → Parser s r
(f<$>p) inp = [(f r, rstinp) | (r, rstinp) ← p inp]
```

Utilize este combinador para modelar as seguintes linguagens:

1. *Um símbolo* **`a`** *seguido de um símbolo* **`b`**

   ```
   alinea_1 = f<$>symbol 'a'<*>symbol 'b'
     where f a b = ()
   ```

2. *Dois espaços seguidos.*

   ```
   doisespaco = f<$>symbol ' '<*>symbol ' '
     where f a b = ()
   ```

3. *Implemente o parser* **`zeroOuMaisEspacos`** *para definir a linguagem cujas frases são zero ou mais espaços seguidos.*

   ```
   zeroOuMaisEspacos = p0<$>symbol ' '<*>zeroOuMaisEspacos
                     <|>succeed p1
     where p0 a b = ""
           p1     = ""
   ```

4. *Implemente o parser* **`umOuMaisEspacos`** *para definir a linguagem cujas frases são um ou mais espaços seguidos.*

$$
\begin{aligned}
umOuMespacos = \;& p0 <\$> symbol\ '\ ' <*> umOuMespacos \\
<|> & \; p1 <\$> symbol\ '\ ' \\
& \textbf{where}\ p0\ a\ b = a : b \\
& \qquad p1\ a \quad = [a]
\end{aligned}
$$

2.4 *Define dois novos combinadores para expressar as estrutura repetitivas "zero ou mais vezes"e "uma ou mais vezes". Isto é, defina os combinadores* **`zeroOuMais`** *e* **`umOuMais`** *que recebem como argumento um parser e expressam a sua repetição. Ou por outras palavras, generalize as funções* **`zeroEspacos`** *e* **`umOuMaisEspaco`***.*

1. *Expresse a função de parsing* **`umOuMaisEspacos`** *usando o combinador* **`umOuMais`**

$$
\begin{aligned}
zeroOuMais\ p = \;& p0 <\$> p <*> zeroOuMais\ p \\
<|> & \; succeed\ p1 \\
& \textbf{where}\ p0\ a\ b = a : b \\
& \qquad p1 = [\,]
\end{aligned}
$$

$$
\begin{aligned}
umOuMais\ p = \;& p0 <\$> p <*> umOuMais\ p \\
<|> & \; p1 <\$> p \\
& \textbf{where}\ p0\ a\ b = a : b \\
& \qquad p1\ a = [a]
\end{aligned}
$$

2. *Defina a função de parsing* **`parseInt`** *para processar a linguagem dos números inteiros.*

$$
\begin{aligned}
palavra &= umOuMais\ letra \\
iss &= umOuMais\ digito \\
espacos' &= umOuMais\ espaco
\end{aligned}
$$