Ficha1-ATS

Luís Braga

December 2019

Contents

1 Ficha 1

```haskell
module Ficha1 where

import Prelude hiding ((<*>),(<$>))

import Data.Char
import Parser

-- Exerc cios

{-
    Exerc cio 1.1) Define combinadores de parsing para expressar os s mbolos terminais number
        e ident.
-}

data Exp = AddExp Exp Exp
         | MulExp Exp Exp
         | SubExp Exp Exp
         | GThen Exp Exp
         | LThen Exp Exp
         | OneExp Exp
         | Var String
         | Const Int

instance Show Exp where
  show = showExp

showExp (AddExp e1 e2) = showExp e1 ++ " + " ++ showExp e2
showExp (SubExp e1 e2) = showExp e1 ++ " - " ++ showExp e2
showExp (MulExp e1 e2) = showExp e1 ++ " * " ++ showExp e2
showExp (GThen e1 e2)  = showExp e1 ++ " > " ++ showExp e2
showExp (OneExp e)     = "( " ++ showExp e ++ " )"
showExp (Const i)      = show i
showExp (Var a)        = a

--R:

number =   f <$> satisfy isDigit
       <|>  g <$> satisfy isDigit <*> number
  where f a = [a]
        g a b = a:b

ident =  oneOrMore (satisfy isAlpha)

{-
    Exerc cio 1.2) Utilizando o tipo de dados Exp defina a express o artim tica "e=(var+3)*5".
-}

e :: Exp
e = MulExp (OneExp (AddExp (Var "var") (Const 3))) (Const 5)
```

```
50
51 {-
52     Exerc cio 1.3) Escreva o parser baseado em combinadores que reconhece a nota  o de
53         express es aritm ticas produzida pela fun  o de pretty printing anterior e devolve
       a  rvore
54         de syntaxe abstrata com tipo Exp.
55 -}
56
57 --R:
58 pexp :: Parser Char Exp
59 pexp =   f <$> pterm
60      <|> g <$> pterm <*> symbol' '+' <*> pexp
61      <|> h <$> pterm <*> symbol' '-' <*> pexp
62    where f a = a
63          g a b c = AddExp a c
64          h a b c = SubExp a c
65
66 pterm :: Parser Char Exp
67 pterm =  f <$> pfactor
68      <|> g <$> pfactor <*> symbol' '*' <*> pterm
69    where f a = a
70          g a b c = MulExp a c
71
72 pfactor :: Parser Char Exp
73 pfactor =   f <$> number
74        <|>  g <$> ident
75        <|>  h <$> symbol' '(' <*> pexp <*> symbol' ')'
76   where f a = Const (read a)
77         g a = Var a
78         h a e f = OneExp e
79
80 {-
81     Exerc cio 1.4) Considere a seguinte express o artim tica:
82     e1 = "2 * 4 - 34"
83
84     Verfique que o parser desenvolvido n o processa este input. Atualize a gram tica de modo
85     a considerar a exist ncia de espa os a separar s mbolos das express es. Sugest o:
       defina
86     um parser spaces, que define a linguage de zero, um ou mais espa os. Defina ainda
87     uma parser symbol  que processa o s mbolo dado e depois consome (ignorando) eventuais
88     espa os que apare am a seguir.
89 -}
90
91 symbol' a = (\ a b c -> b) <$> spaces <*> symbol a <*> spaces
92
93 {-
94   Exerc cio 1.5) O parser spaces descreve uma constru  o sintatica muito frequente em
95     parsing: zero, uma ou mais vezes (o operador * das express es regulares). Adicione
96     biblioteca Parser.hs o combinador
97
98     zeroOrMore :: Parser s r -> Parser s [r]
99
100    que aplica um dado parser zero uma ou mais vezes e que devolve uma lista com os
101    resultados das v rias aplica  es do parser.
102 -}
103
104 zeroOrMore :: Parser s r -> Parser s [r]
105 zeroOrMore p = sf <$> p <*> zeroOrMore p
106              <|> succeed []
107   where sf x xs = x : xs
108
109 {-
110   Exerc cio 1.6) Defina o parser spaces em termos de zeroOrMore.
111 -}
112
113 spaces = zeroOrMore
114          (satisfy (\x -> x `elem` [' ','\t','\n']))
115
116 {-
117   Exerc cio 1.7) Defina (em Parser.hs) o parser oneOrMore em termos de zeroOrMore.
118     Sugest o: Recorde que a+     a a  .
119
120     Considere que definiu a seguinte linguagem de programa  o que    constitu da por uma
121     sequ ncia de statements e em que um statement pode ser um cilo while, um condicional if
122     ou uma atributi  o. Esta linguagem    definida pelo seguinte tipo de dados abstrato:
123 -}
```

```
124
125 data Prog = Prog Stats
126 type Id = String
127 data Stats = Stats [Stat]
128
129 data Stat = While Exp Stats
130           | IfThenElse Exp Stats Stats
131           | Assign Id Exp
132
133 --Considere ainda que foi escrita a seguinte fun  o de pretty printing:
134
135 instance Show Prog where
136   show = showProg
137
138 showProg (Prog sts) = showStats sts
139
140 instance Show Stats where
141   show = showStats
142
143 showStats (Stats l) = showStatsList l
144
145 showStatsList :: [Stat] -> [Char]
146 showStatsList [] = ""
147 showStatsList (st:[]) = showStat st
148 showStatsList (st:sts) = showStat st ++ ";\n " ++ (showStatsList sts)
149
150 instance Show Stat where
151   show = showStat
152
153 showStat :: Stat -> [Char]
154 showStat (Assign n e) = n ++ " = " ++ showExp e
155 showStat (While e sts) = "while (" ++ showExp e ++ ")\n " ++ "{ " ++ showStats sts ++ "}"
156 showStat (IfThenElse e s s1) = "if (" ++ showExp e ++ ")\nthen{" ++ showStats s ++ "}\nelse{"
    ++ showStats s1 ++ "}\n"
157
158 oneOrMore p =  sf1 <$> p <*> zeroOrMore p
159   where sf1 x xs = x : xs
160
161 {-
162   Exerc cio 1.8) Escreva o parser baseado em combinadores para esta linguagem cuja nota  o
163         definida pela fun  o showProg.
164 -}
165
166 -- R:
167
168 pProg :: Parser Char Prog
169 pProg = Prog <$> pStats
170
171
172 -- Funcoes antes da resolucao de 1.10
173 {--
174 pStats :: Parser Char Stats
175 pStats =  f <$> token ""
176       <|> g <$> pStat
177       <|> h <$> pStat <*> token ";\n" <*> pStats
178       where f a = Stats []
179             g a = Stats [a]
180             h a b c = Stats (a: (i c))
181             i = (\ (Stats b) -> b)
182
183 pStat :: Parser Char Stat
184 pStat =   h <$> token  "while (" <*> pexp <*> symbol' ')' <*> symbol' '{' <*> pStats <*>
    symbol' '}'
185       <|> g <$> token "if (" <*> pexp <*> token ")\nthen{" <*> pStats <*> token "}\nelse{" <*>
     pStats <*> token "}\n"
186       <|> f <$> ident <*> symbol' '=' <*> pexp
187       where g a b c d e f g = IfThenElse b d f
188             f a b c = Assign a c
189             h a b c d e f = While b e
190
191 --}
192
193 {-
194   Exerc cio 1.9) No desenvolvimento do parser pProg foram utilizadas constru  es
    sint ticas
195    muito frequentes em linguagem de programa  o: separatedBy (lista de elementos sepa-
```

```
    rados por um dado separador, neste exemplo ponto e virgula), enclosedBy (elementos
    delimitados por um s mbolo inicial e final, neste exemplo parentesis curvos). Defina em
    Parser.hs estes combinadores que descartam o resultado de fazer parsing aos separado-
    res/delimitadores.


separatedBy :: Parser s a -> Parser s b -> Parser s [a]
separatedBy p s =  f <$>  p <*> s <*> (separatedBy p s)
              <|> g <$>  p
             where f a b c = a : c
                   g a     = [a]
-}
separatedBy :: Parser s a -> Parser s b -> Parser s [a]
separatedBy d s =  f <$> d
               <|> g <$> d <*> s <*> separatedBy d s
             where f a = [a]
                   g a b c = a:c

enclosedBy :: Parser s a -> Parser s b -> Parser s c -> Parser s b
enclosedBy p1 p2 p3 =  g <$> p1 <*> p2 <*> p3
                   where g a b c = b


{-
  Exerc cio 1.10) Re-escreva pProg utilizando separatedBy e enclosedBy
-}

token' a = (\ a b c -> b) <$> spaces <*> token a <*> spaces

pStats :: Parser Char Stats
pStats = Stats <$> separatedBy pStat (symbol' ';')

pStat :: Parser Char Stat
pStat =  f <$> token' "while" <*> (enclosedBy (symbol' '(') pexp (symbol' ')'))
                              <*> (enclosedBy (symbol' '{') pStats (symbol' '}'))
     <|> g <$> token' "if"    <*> (enclosedBy (symbol' '(') pexp (symbol' ')'))
                              <*> token' "then" <*> (enclosedBy (symbol' '{') pStats (symbol'
    '}'))
                              <*> token' "else" <*> (enclosedBy (symbol' '{') pStats (symbol'
    '}'))
     <|> h <$> ident <*> symbol' '=' <*> pexp
    where f w c b = While c b
          g a b c d e f = IfThenElse b d f
          h a b c = Assign a c

{-
  Exerc cio 1.11) Adicione    biblioteca Parser.hs mais constru  es sint ticas frequentes
    em longuagens de programa  o , nomeadamente:
-}

followedBy :: Parser s a -> Parser s b -> Parser s [a]
followedBy d s =  g <$> d <*> s <*> followedBy d s
              <|>  succeed []
             where f a b = [a]
                   g a b c = a:c

{-
block :: Parser s a -- open delimiter
-> Parser s b -- syntactic symbol that follows statements
-> Parser s r -- parser of statements
-> Parser s f -- close delimiter
-> Parser s [r]
-}

block :: Parser s a -> Parser s b -> Parser s r -> Parser s f -> Parser s [r]
block od ss ps cd = enclosedBy od (followedBy ps ss) cd
```

2 Parser

```

module Parser where

import Prelude hiding ((<*>),(<$>))

```

```haskell
infixl 2 <|>
infixl 3 <*>

type Parser s r = [s] -> [(r , [s])]

symbola :: Parser Char Char
symbola [] = []
symbola (x:xs) = if x == 'a' then [('a',xs)]
                 else []

symbol :: Eq a => a -> Parser a a
symbol s []                = []
symbol s (x:xs) | s == x    = [(s,xs)]
                | otherwise = []

satisfy :: (s -> Bool) -> Parser s s
satisfy p []                  = []
satisfy p (x:xs) | p x        = [(x,xs)]
                 | otherwise = []


token :: Eq s => [s] -> Parser s [s]
token t [] = []
token t inp = if   take (length t) inp == t
              then [(t,drop (length t) inp)]
              else []

succeed :: r -> Parser s r
succeed r inp = [ ( r , inp) ]

(<|>) :: Parser s a -> Parser s a -> Parser s a
(p <|> q) inp = p inp ++ q inp

pS =   token "while"
  <|>  token "for"

{-
(<*>)::Parser s a-> Parser s b-> Parser s (a,b)
(p <*> r) inp = [ ((x,y),ys)
                | (x,xs)  <- p inp
                , (y,ys)  <- r xs
                ]
-}

(<$>) :: (a -> r) -> Parser s a -> Parser s r
(f <$> p) inp = [ (f v , xs)
                | (v   , xs)  <- p inp
                ]

{-
pS' =   f <$> (symbol 'a' <*> symbol 'b' <*> symbol 'c')
    <|> g <$> (symbol 'd')
  where f ((a,b),c) = [a,b,c]
        g d         = [d]
-}

(<*>) :: Parser s (a -> b)
      -> Parser s a
      -> Parser s b
(p <*> r) inp = [ (f v ,ys)
                | (f   ,xs)  <- p inp
                , (  v ,ys)  <- r xs
                ]
pS' =  f <$> symbol 'a' <*> symbol 'b' <*> symbol 'c'
   <|> g <$> symbol 'd'
  where f a b c  = [a,b,c]
        g d      = [d]
```