Análise e Teste de Software

Ficha: Combinadores de Parsing

Universidade do Minho

Ano lectivo 2019/2020

João Saraiva

1 Exercícios

Nesta ficha pretende-se recordar os conhecimentos de gramáticas independentes de contexto e modelar estas gramáticas diretamente em Haskell usando combinadores de Parsing.

Resolva os exercícios re-utilizando a biblioteca de combinadores de Parsing `Parser.hs` desenvolvida nas aula teórica.

Considere a seguinte gramática independente de contexto que define a linguagem de expressões aritméticas, onde a prioridade dos respetivos operadores está expressa nas próprias produções da gramática:

```
exp   -> term
       | term '+' exp
       | term '-' exp
       ;
term  -> factor
       | factor '*' term
       | factor '/' term
       ;
factor -> number
       |  ident
       | '(' exp ')'
       ;
```

onde `number` regresenta uma sequência de digitos, e `ident` um sequência de letras miusculas ou miusculas.

Exercício 1.1 *Define combinadores de parsing para expressar os símbolos terminais* ***number*** *e* ***ident****.*

Considere que expressões aritméticas se definem abstratamente pelo seguinte tipo de dados algébrico:

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

Considere ainda que se definiu a seguinte função de pretty printing para estas expressões artiméticas.

```
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
```

Exercício 1.2 *Utilizando o tipo de dados* ***Exp*** *defina a expressão artimética* ***"e=(var+3)*5"****.*

```
e :: Exp
e =
```

Exercício 1.3 *Escreva o parser baseado em combinadores que reconhece a notação de expressões aritméticas produzida pela função de pretty printing anterior e devolve a árvore de syntaxe abstrata com tipo* ***Exp****.*

```
pExp ::  Parser Char Exp
```

Exercício 1.4 *Considere a seguinte expressão artimética:*

```
e1 = "2 * 4 - 34"
```

Verfique que o parser desenvolvido não processa este input. Atualize a gramática de modo a considerar a existência de espaços a separar símbolos das expressões. Sugestão: defina um parser `spaces`*, que define a linguage de zero, um ou mais espaços. Defina ainda uma parser* `symbol'` *que processa o símbolo dado e depois consome (ignorando) eventuais espaços que apareçam a seguir.*

Exercício 1.5 *O parser* `spaces` *descreve uma construção sintatica muito frequente em parsing: zero, uma ou mais vezes (o operador* `*` *das expressões regulares). Adicione à biblioteca* `Parser.hs` *o combinador*

```
zeroOrMore :: Parser s r -> Parser s [r]
```

que aplica um dado parser zero uma ou mais vezes e que devolve uma lista com os resultados das várias aplicações do parser.

Exercício 1.6 *Defina o parser* `spaces` *em termos de* `zeroOrMore`*.*

Exercício 1.7 *Defina (em* `Parser.hs`*) o parser* `oneOrMore` *em termos de* `zeroOrMore`*. Sugestão: Recorde que* $a^+ \equiv aa^*$*.*

Considere que definiu a seguinte linguagem de programação que é constituída por uma sequência de *statements* e em que um statement pode ser um cilo *while*, um condicional *if* ou uma atribuição. Esta linguagem é definida pelo seguinte tipo de dados abstrato:

```
data Prog  = Prog  Stats

data Stats = Stats [Stat]

data Stat  = While      Exp Stats
           | IfThenElse Exp Stats Stats
           | Assign     Id Exp
```

Considere ainda que foi escrita a seguinte função de prettu printing:

```
instance Show Prog where
  show = showProg

showProg (Prog sts) = showStats sts


instance Show Stats where
  show = showStats

showStats (Stats l) = showStatsList l
```

```
showStatsList []       = ""
showStatsList (st:[])  = showStat  st
showStatsList (st:sts) = showStat st ++ ";\n " ++ (showStatsList sts)


instance Show Stat where
  show = showStat

showStat (Assign n e) = n ++ " = " ++ showExp e
showStat (While  e sts) = "while (" ++ showExp e ++ ")\n " ++
                          "{ " ++ showStats sts ++ "}"
```

Exercício 1.8 *Escreva o parser baseado em combinadores para esta linguagem cuja notação é definida pela função* ***showProg****.*

```
pProg :: Parser Char Prog
```

Exercício 1.9 *No desenvolvimento do parser* ***pProg*** *foram utilizadas construções sintáticas muito frequentes em linguagem de programação:* ***separatedBy*** *(lista de elementos separados por um dado separador, neste exemplo ponto e virgula),* ***enclosedBy*** *(elementos delimitados por um símbolo inicial e final, neste exemplo parentesis curvos). Defina em* ***Parser.hs*** *estes combinadores que descartam o resultado de fazer parsing aos separadores/delimitadores.*

```
separatedBy :: Parser s a -> Parser s b -> Parser s [a]
enclosedBy  :: Parser s a -> Parser s b -> Parser s c -> parser s b
```

Exercício 1.10 *Re-escreva* ***pProg*** *utilizando* ***separatedBy*** *e* ***enclosedBy***

Exercício 1.11 *Adicione à biblioteca* ***Parser.hs*** *mais construções sintáticas frequentes em longuagens de programação, nomeadamente:*

```
followedBy :: Parser s a -> Parser s b -> Parser s [a]
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]
```