Ficha2-ATS

Luís Braga

December 2019

Contents

1 ASTandPP

```haskell
module ASTandPP where

data P = R Its

type Its = [It]

data It = Block Its
        | Decl String
        | Use String

instance Eq It where
    Use a   == Use b   =  a == b
    Decl a  == Decl b  =  a == b
    Block a == Block b =  a == b
    _       == _       =  False

instance Show P where
    show = pp_P

pp_P (R its) = "[" ++ pp_Its its ++ "]"

instance Show It where
    show = pp_It

pp_Its [] = ""
pp_Its [it] = pp_It it
pp_Its (it:its) = pp_It it ++ " , " ++ pp_Its its

pp_It (Decl n) = "Decl " ++ n
pp_It (Use n) = "Use " ++ n
pp_It (Block is) = "[" ++ pp_Its is ++ "]"
```

2 LangParser

```haskell
module LangParser where

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

pP :: Parser Char P
pP = R <$> pIts

pIts :: Parser Char Its
```

```
pIts =  enclosedBy (symbol' '[') (separatedBy pIt (symbol' ',')) (symbol' ']')

pIt :: Parser Char It
pIt =   f <$> token' "Decl" <*> ident
   <|>  g <$> token' "Use" <*> ident
   <|>  Block <$> pIts
   where f a b = Decl b
         g a b = Use b
```

3 Parser

```

module Parser where

import Prelude hiding ((<*>),(<$>))
import Data.Char

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]


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

ident =  oneOrMore (satisfy isAlpha)

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

zeroOrMore :: Parser s r -> Parser s [r]
zeroOrMore p = sf <$> p <*> zeroOrMore p
             <|> succeed []
  where sf x xs = x : xs

oneOrMore p =  sf1 <$> p <*> zeroOrMore p
          where sf1 x xs = x : xs


spaces = zeroOrMore
  (satisfy (\x -> x `elem` [' ','\t','\n']))


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 = (\ a b c -> b) <$> p1 <*> p2 <*> p3


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

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

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
```

4 SemanticAnalyzer

```
module SemanticAnalyzer where

import Parser
import ASTandPP
import LangParser

isDecl (Decl _) = True
isDecl _ = False

isUse (Use _) = True
isUse _ = False

isBlock (Block _) = True
isBlock _ = False

(<||>) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
(p <||> q) inp = (p inp) || (q inp)

```

```
-- Verifica de existe um Decl de um determinado Use
-- Exemplo:
-- declOfUse (Use "w") [Decl "w", Decl "q"]
-- True
-- declOfUse (Use "d") [Decl "w", Decl "q"]
-- False
declOfUse :: It -> [It] -> Bool
declOfUse (Use a) b = elem (Decl a) b

-- Verifica os erros de uma n vel
-- 1 Argumento: lista de declara  es herdadas
-- 2 Argumento: lista do que j  foi processado
-- 3 Argumento: lista do que falta processar
--
-- Classifica  o dos erros:
-- 1 condicional: se for "Use" e n o for declarado no que vai ser
--                processado ou no que foi processado ou nas declaracoes herdadas <-    um erro **
-- 2 condicional: se for uma declaracao e existir uma declarcao dela no futuro <-    um erro
-- 3 condicional: caso contr rio est  correto
-- ** por an lise do enunciado n o se considera erro se existir no passado
levelerrors :: Its -> Its -> Its -> Its
levelerrors _ _ []     = []
levelerrors pe t (x:xs) | isUse x && not (declOfUse x xs || declOfUse x t || declOfUse x pe)
    = x : levelerrors pe (t ++ [x]) xs
                        | isDecl x && (elem x xs)
    = x : levelerrors pe (t ++ [x]) xs
                        | otherwise
    = levelerrors pe (t ++ [x]) xs

-- Funcao que detecta os erros gerais
-- Aplica a "levelerrors" a um nivel e aplica o mesmo aos blocks
-- "leva" consigo a lista de declaracoes do n vel presente para os pr ximos
-- Aplica a "levelerrors" s  a objectos do tipo "Use" e "Decl"
errors :: Its -> Its -> Its
errors pdecl its = levelerrors pdecl [] f ++ concat( map ( error . (\ (Block a) -> a) )  h)
             where f     = filter (isDecl <||> isUse )  its          -- filtra objectos do tipo "Decl" e "Use"
                   error = errors ((filter isDecl its) ++ pdecl )
                   h     = filter isBlock its                        -- filtra objectos do tipo Block

-- Funcao criada a partir da necessidade de cobrir o caso em que    gerada uma lista vazia,
-- pois a fun  o head do Prelude d  exception
head' :: [(P,String)] -> (P,String)
head' [] = (R [], "")
head' (x:xs) = x

-- Funcao principal
-- 1 Realiza o Parse da linguagem
-- 2 Retira o primeiro elemento que consumiu todo o input (fez parse correctamente)
-- 3 Analisa os erros semanticos
semanticAnalyzer = errors [] . (\ (R a) -> a) . fst . head' . filter ( (==) "" . snd) . pP
```