PFL Project 2

3LEIC02 G12

Gonçalo Gonçalves Miranda - up202108773 (50%)

Pedro Emanuel Faria da Fonseca - up202108653 (50%)

How to run the project

```
$ ghci
ghci> :l main.hs
ghci> main
ghci> testParser "<input>"
```

General Description

In this project, we were asked to design an Haskell compiler for a simple imperative language.

Part 1 - Data Definition and Assembler

In the first part of the assignment, we must define how our data will be organized and create the assembler of the language.

Data Types

In this first step of our project, we needed to define the possible instructions of a stack, the possible values in that same data structure and the stack itself, which is just a list of those same values.

After this, we also defined the data type State, which is basically a list of tuples where a string is associated with a value.

Running the Program

In order to run the program, we created a function that received a list of instructions as an argument. For each instruction it read at the top of the list, the stack and/or storage were updated as needed. This was followed by the recursive call of the run function which now contained the tail of the list of instructions. When achieving a state where this list is empty, we made it so the program halted execution, returning a truple with en empty instructions list, the final stack state and the final storage state. This could now be used to compare execution fidelity against known results for a given input or to simply print out the result of the program.

Part 2 - Parsing and Building Statements

In this second part of the project, we need to design a lexer and a parser for our language.

Data types

For this, we started by defining the data types we'll need for this implementation to be possible.

In this case we're using a few data types to represent the expressions we have, that can be either boolean or arithmetic. Besides that, we also defined our possible statements and that a program is a list of statements.

```
 -- Possible Arithmetic Operations and Expressions
data Aexp = CONST Integer
          | VAR String
          | ADD Aexp Aexp
          | SUB Aexp Aexp
          | MUL Aexp Aexp
          deriving (Show, Eq)

-- Possible Boolean Operations and Expressions
data Bexp = TRUE
          | FALSE
          | AND Bexp Bexp
          | NOT Bexp
          | INTEQ Aexp Aexp
          | LEINT Aexp Aexp
          | BOOLEQ Bexp Bexp
          deriving (Show, Eq)

-- Possible Statements
data Stm = ASS String Aexp
         | IF Bexp [Stm] [Stm]
         | WHILE Bexp [Stm]
         deriving (Show, Eq)

type Program = [Stm]
```

In order to build the lexer, we first defined the tokens that must be originated once certain symbols or words have been inserted into our program.

```
 -- Possible tokens
data Token = IntegerToken Integer
           | PlusToken            -- +
           | MultToken           -- *
           | MinusToken           -- -
           | OpenPToken           -- (
           | ClosedPToken         -- )
           | IfToken              -- if
           | ThenToken            -- then
           | ElseToken            -- else
           | VarToken String      -- variable
           | AssignToken          -- :=
           | WhileToken           -- while
           | DoToken              -- do
           | TrueToken            -- True
           | FalseToken           -- False
           | AndToken             -- and
           | NotToken             -- not
           | NumeralEqToken       -- ==
           | BoolEqToken          -- =
           | LessOrEqToken        -- <=
           | SemiColonToken       -- ;
            deriving (Show, Eq)
```

Lexer

The lexer is used to read the user's input string and to translate it into a list of tokens that our parser can later use to totally translate the string into instructions. For this to work as intended, we test if the string provided to our function is any of the special characters or instructions present in our language and then convert it into its corresponding token. All the spaces are ignored.

```
 lexer :: String -> [Token]
lexer [] = []
lexer ('+': rest) = PlusToken : lexer rest
lexer ('*': rest) = MultToken : lexer rest
lexer ('-': rest) = MinusToken : lexer rest
lexer ('(': rest) = OpenPToken : lexer rest
lexer (')': rest) = ClosedPToken : lexer rest
lexer ('n': 'o': 't': rest) = NotToken : lexer rest
lexer ('a': 'n': 'd': rest) = AndToken : lexer rest
lexer ('i': 'f': rest) = IfToken : lexer rest
lexer ('t': 'h': 'e': 'n': rest) = ThenToken : lexer rest
lexer ('e': 'l': 's': 'e': rest) = ElseToken : lexer rest
lexer ('w': 'h': 'i': 'l': 'e': rest) = WhileToken : lexer rest
lexer ('d': 'o': rest) = DoToken : lexer rest
lexer ('=': '=': rest) = NumeralEqToken : lexer rest
lexer ('=': rest) = BoolEqToken : lexer rest
lexer ('<': '=': rest) = LessOrEqToken : lexer rest
lexer (':': '=': rest) = AssignToken : lexer rest
lexer ('T': 'r': 'u': 'e': rest) = TrueToken : lexer rest
lexer ('F': 'a': 'l': 's' : 'e': rest) = FalseToken : lexer rest
lexer (';': rest) = SemiColonToken : lexer rest
lexer (c: rest)
  | isSpace c = lexer rest  -- ignore spaces
  | isDigit c = IntegerToken (read num) : lexer rest'   -- get digits and convert to integer
  | isLower c = VarToken var : lexer rest''           -- starts w/ lowercase letter -> variable
  | otherwise = error ("Bad character: " ++ [c])
  where (num, rest') = span isDigit (c:rest)        -- get all digits
        (var, rest'') = span isAlphaNum (c:rest)    -- get all alphanumeric characters
```

Parser

For our parser implementation, in order to deal with the order of precedence of operators and the different type of expressions available, we decided to handle each one seperately.

Arithmetic Expressions

In order to parse arithmetic expressions, we decided to have a function for each operator, and call the respective functions according to the operator's precedence. In this case, the order of precedence of the functions we needed to implement was

`parseSumOpUp -> parseMultOpUp -> parseAtom (which is the function that has the operator with highest precedence)`

When parsing parenthesis, we can call parseSumOpUp and, after parsing the expressions comprehended between parenthesis, it returns to parseMulOpUp, parsing all the products, and finally to parseSumOpUp, parsing all the sums and subtractions.

Boolean Expressions

When parsing boolean expressions, we decided to use a similar approach to the one used for arithmetic expressions, which helps us deal with both at the same time, since arithmetic expressions can be part of boolean expressions. In order to evaluate arithmetic expressions, we called the function that deals with the operator with least precedence in arithmetic expressions (parseSumOpUp) when we reach the "integer equality" or the "less than equals" precedence levels and then keep parsing the result of this function as a boolean expression again.

The precedence sequence of the functions used to parse boolean expressions is:

`parseAndOpUp -> parseBoolEqOpUp -> parseNotOpUp -> parseIntEqOpUp -> parseLeOpUp`

Building Statements

When parsing statements, we only need to parse the list of tokens and create smaller token lists containing each part of the statement. However, due to the possible presence of parenthesis inside statements, we needed to create a function to fetch the tokens present in between parenthesis (getBetweenParenthesisTokens) and, after it returns we call the parser recursively to parse the previous output.