Functional and Logic Programming

Practical Assignment 2

Turma 05 - Grupo 05

João Guimarães Mota (up202108677) Contribuição - 50%

Pedro Jorge Mendes Jesus Landolt (up202103337) Contribuição - 50%

Execution

To execute the project, as it is all inside the same *main.hs* file, you only need to run, inside GHCi terminal, the command **':l main.hs'** and then **main**, to run all the sample tests.

Project Description

In this project we implemented a compiler for a simple imperative language, with arithmetic and boolean expressions, statements consisting of assignments of the form x:= a, sequence of statements (instr1 ; instr2), if then else statements, and while loops.

As suggested in the assignment description we subdivided our implementation into four main parts:

Lexer - splits the input string into a list of words (tokens)

Parser - reads the list of tokens and transforms it into a list of statements

Compiler - reads the list of statements and transforms it into a list of instructions

Assembler - reads the list of instructions and executes them

Lexer

As mentioned above, the lexer function splits the input string, of the form

" i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i - 1;); "

into a list of possible tokens we defined for this language.

Due to the language's simplicity and limited set of keywords, our approach involves verifying that the input string aligns with any given keyword, and subsequently returning the associated token. In the event that the input string doesn't align with any predefined keyword or token, an error is returned.

```
-- Given a string, returns the corresponding list of tokens
lexerHelper :: String -> [Token]
lexerHelper word =
 case () of
   _ | isBoolOp word -> [BoolOpToken word]
     | isIntBoolOp word -> [IntBoolOpToken word]
     | isArithmeticOp word -> [ArithmeticOpToken word]
     | isReserved word -> [ReservedToken word]
     | isSemiColon word -> [SemiColonTok]
     | isOpenParen word -> [OpenParenTok]
     | isCloseParen word -> [CloseParenTok]
     | all isDigit (takeWhile (/=';') word) -> [IntConstToken (read (takeWhile (/=';')
  word))]
     | all isAlphaNum (take 1 word) -> [VarToken word]
```

```
    | otherwise -> error ("Cannot parse " ++ word ++ " on lexer")

-- Helper functions to check if a string is a token

-- Checks if a string is the boolean operator '='
isBoolOp :: String -> Bool
isBoolOp s = s == "="

-- Checks if a string is one of the boolean operators '==', '<='
isIntBoolOp :: String -> Bool
isIntBoolOp s = s `elem` ["==", "<="]

-- Checks if a string is one of the arithmetic operators ':=', '+', '-', '*'
isArithmeticOp :: String -> Bool
isArithmeticOp s = s `elem` [":=", "+", "-", "*"]

-- Checks if a string is one of the reserved words 'if', 'then', 'else',
'while', 'do', 'True', 'False', 'not', 'and', 'true', 'false'
isReserved :: String -> Bool
isReserved s = s `elem` ["if", "then", "else", "while", "do", "True", "False",
"not", "and", "true", "false"]
```

Parser

Following the instructions in the assignment description, we defined three data types to represent expressions and statements of this imperative language:

- **Aexp** for arithmetic expressions

```
data Aexp = Var String          -- Variable
          | IntConst Integer    -- Integer
          | AddP Aexp Aexp      -- Sum
          | SubtractP Aexp Aexp -- Subtraction
          | MultiplyP Aexp Aexp -- Multiplication
          deriving (Show)
```

- **Bexp** for boolean expressions

```
data Bexp = TrueP Bool          -- Boolean True
          | FalseP Bool         -- Boolean False
          | LessEqP Aexp Aexp   -- Less Than or Equal
          | IntEqP Aexp Aexp    -- Integer Equality
          | BoolEqP Bexp Bexp   -- Boolean Equality
          | NotP Bexp           -- Boolean negation
```

```
                | AndP Bexp Bexp        -- Logical Operation
                deriving (Show)
```

• **Stm** for statements

```
data Stm = Assign String Aexp    -- Assignment
         | If Bexp [Stm] [Stm]   -- Conditional
         | While Bexp [Stm]      -- Loop
         deriving (Show)
```

Using the list of tokens provided by the lexer and the new data types we defined we implemented the function ***statementBuild***, that splits the list of tokens into smaller parts, each corresponding to part of a statement.

Here's a snippet of the function (just the assignments statements) :

```
-- Given a list of tokens, returns the corresponding list of statements
statementBuild :: [Token] -> [Stm]
statementBuild [] = []
statementBuild (SemiColonTok:tokens) = statementBuild tokens

-- Assignments
statementBuild ((VarToken var):ArithmeticOpToken ":=":tokens) =
 Assign var (stmAexp aexp) : statementBuild rest
 where
   (aexp, rest) = break (== SemiColonTok) tokens
```

The challenge lies in handling parentheses, especially when scenarios involve conditional statements within a while loop for example. To address this concern, we implemented a ***getBetweenParenTokens*** function, which extracts tokens enclosed within parentheses. Subsequently, we recursively invoke the parser to analyze the tokens obtained from within the parentheses.

```
-- Extracts tokens enclosed with parentheses, returning them and the
remaining tokens after the closing parenthesis
getBetweenParenTokens :: [Token] -> ([Token], [Token])
getBetweenParenTokens tokens = (elseTokens, restTokens)
 where (restTokens, _, elseTokens) = getBetweenParenTokensAux tokens "" []

-- Auxiliary function that ensures the correct handling of nested
parentheses using a "stack" (string)
getBetweenParenTokensAux :: [Token] -> String -> [Token] -> ([Token],
String, [Token])
getBetweenParenTokensAux [] stk res = ([], "", reverse res)
```

To parse arithmetic expressions we created many functions, each responsible for a specific operation, and regarding the precedence of the operators.

We start with the function *parseIntVarParens*, responsible for parsing integer literals, variables and parentheses, operations that have the highest precedence. Then we have the function *parseMult*, responsible for the multiplication operation, with a medium precedence. Finally, we have the function *parseSumSub*, that has the lowest precedence and is responsible for the sum and subtraction operations.

The lowest precedence function calls the medium precedence function that does the same to the highest one.

```
-- Handles Integers, Variables and Parenthesis (highest precedence)
parseIntVarParens :: [Token] -> Maybe (Aexp, [Token])
parseIntVarParens (IntConstToken int : tokensLeft) = Just (IntConst
int, tokensLeft)
parseIntVarParens (VarToken var : tokensLeft) = Just (Var var,
tokensLeft)
parseIntVarParens (OpenParenTok:tokensLeft1) =
 case parseSumSub tokensLeft1 of
   Just (aexp, CloseParenTok:tokensLeft2) -> Just (aexp, tokensLeft2)
   Just _ -> Nothing
   Nothing -> Nothing
parseIntVarParens _ = Nothing
```

To parse boolean expressions we used the same strategy used in the arithmetic ones. We created different functions to handle each boolean operation regarding the precedence of the operators.
Here's some examples:

```
-- Handles Integer Equality Operation (highest precedence)
parseIntEq :: [Token] -> Maybe (Bexp, [Token])
parseIntEq tokens =
 case parseSumSub tokens of
   Just (aexp1, IntBoolOpToken "==":tokensLeft1) ->
     case parseSumSub tokensLeft1 of
       Just (aexp2, tokensLeft2) -> Just (IntEqP aexp1 aexp2,
tokensLeft2)
       Nothing -> Nothing
   result -> parseLessEq tokens

-- Handles Not Logical Operation (highest precedence)
parseNot :: [Token] -> Maybe (Bexp, [Token])
parseNot (ReservedToken "not": tokensLeft) =
 case parseNot tokensLeft of
   Just (bexp, tokensLeft') -> Just (NotP bexp, tokensLeft')
   Nothing -> Nothing
parseNot tokens = parseIntEq tokens
```

Compiler

Our compiler has a simple implementation, regarding the statement definition we used. Basically it divides each statement into smaller parts and compiles them separately, depending on their type, arithmetic or boolean.

For its definition we created three functions, the main one:

```
-- Compiler function that, given a sequence of statements, returns the
corresponding list of instructions
compile :: [Stm] -> Code
compile [] = []
compile (Assign var aexp : xs) = compA aexp ++ [Store var] ++ compile xs
compile (If bexp stm1 stm2 : xs) = compB bexp ++ [Branch (compile stm1)
(compile stm2)] ++ compile xs
compile (While bexp stm : xs) = Loop (compB bexp) (compile stm) : compile xs

-- Compiler for arithmetic expressions
compA :: Aexp -> Code
compA (Var var) = [Fetch var]
compA (IntConst a) = [Push a]
compA (AddP aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Add]
compA (SubtractP aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Sub]
compA (MultiplyP aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Mult]

-- Compiler for boolean expressions
compB :: Bexp -> Code
compB (TrueP bool) = if bool then [Tru] else [Fals]
compB (FalseP bool) = if bool then [Fals] else [Tru]
compB (LessEqP aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Le]
compB (IntEqP aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Equ]
compB (BoolEqP bexp1 bexp2) = compB bexp2 ++ compB bexp1 ++ [Equ]
compB (NotP bexp) = compB bexp ++ [Neg]
compB (AndP bexp1 bexp2) = compB bexp2 ++ compB bexp1 ++ [And]
```

and the two auxiliary functions, *compA* and *compB*, which compile arithmetic and boolean expressions, respectively.

Assembler

In the assembler part, we defined the **Stack** and **State** types, as well as all the required functions such as *createEmptyStack*, *createEmptyState* and the ones needed to verify the output, *stack2Str* and *state2Str*.

```haskell
-- Possible values in the stack/state
data Values = IntV Integer | BoolV Bool
   deriving Show

-- Stack definition
type Stack = [Values]

-- State definition
type State = [(String, Values)]
```

We also implemented all the functions that represent each one of the instructions. Here are some examples:

```haskell
-- Pushes a constant value onto the stack
push :: Values -> Stack -> Stack
push x xs = x:xs

-- Adds the two top integer values of the stack and then pushes
the result
add :: Stack -> Stack
add (IntV x : IntV y : xs) = IntV(x + y):xs
add (BoolV _ : _ : _) = error "Run-time error"
add _ = error "Run-time error"

-- Compares the top two values (integers or booleans) of the stack
for equality and then pushes the result
eq :: Stack -> Stack
eq (IntV x : IntV y : xs) = BoolV(x == y):xs
eq (BoolV x : BoolV y : xs) = BoolV(x == y):xs
eq (BoolV _ : _ : _) = error "Run-time error"
eq _ = error "Run-time error"
```

And the main function, *run*, that iterates through the list of instructions (**Code**) and returns the resulting **Stack** and **State**.

```haskell
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack, state) = ([], stack, state)
run (x:xs, stack, state) =
   case x of
   Push a -> run (xs, push (IntV a) stack, state)
   Add -> run (xs, add stack, state)
   Mult -> run (xs, mult stack, state)
```

```
Sub -> run (xs, sub stack, state)
Equ -> run (xs, eq stack, state)
Le -> run (xs, le stack, state)
Tru -> run (xs, push (BoolV True) stack, state)
Fals -> run (xs, push (BoolV False) stack, state)
And -> run (xs, myAnd stack, state)
Neg -> run (xs, neg stack, state)
Fetch a -> run (xs, fetch a stack state, state)
Store a -> run (xs, tail stack, store a stack state)
Noop -> run (xs, stack, state)
Branch c1 c2 -> branch c1 c2 stack state xs
Loop c1 c2 -> loop c1 c2 stack state xs
```

Conclusion

This project proved to be more of a challenge than it looked at the beginning but was actually beneficial and interesting. It helped enhance our proficiency in Haskell while introducing fundamental concepts of compiler construction.