PFL Project 2 - Simple Compiler in Haskell

Class 5 Group 11

Name	Number	Contribution (%)
Félix Marcial Alves Martins	202108837	50
Marco Filipe Gonçalves Vilas Boas	202108774	50

Running the Project

To run the project, you should be inside the `src` folder. Then, you can run the following commands:

```
$ ghci
$ :l Main.hs
$ main                      # runs some example tests given in the
assignment
$ additionalTests           # runs tests produced by us
$ assemblerTests            # runs tests for the assembler
$ testParser "<input>"      # runs the parser on the given input
$ testAssembler "<input>"   # runs the assembler on the given input
```

Description

This project consists of the implementation of a compiler in Haskell for a simple imperative language. A possible input program is the following:

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

The compiler is divided into three main parts: the lexer, the parser and the assembler. The lexer is responsible for reading the input string and converting it into a list of tokens. The parser is responsible for reading the list of tokens and converting it into a list of Statements. After this there is a compile step where the list of Statements is translated into instructions of our virtual machine. Finally, the assembler takes this list of instructions and executes them one by one, effectively running the code and producing a result.

Our Solution

Lexer

Defining Tokens

We defined a data type for all the possible tokens in our language.

```
data Token = IntToken Integer       -- integer literal
           | PlusTok                -- +
           | TimesTok               -- *
           | MinusTok               -- -
           | OpenParenTok           -- (
           | ClosedParenTok         -- )
           | IfTok                  -- if
           | ThenTok                -- then
           | ElseTok                -- else
           | VarTok String          -- variable name
           | AssignTok              -- :=
           | WhileTok               -- while
           | DoTok                  -- do
           | TrueTok                -- True
           | FalseTok               -- False
           | AndTok                 -- and
           | NotTok                 -- not
           | IntEqTok               -- ==
           | BoolEqTok              -- =
           | LessOrEqTok            -- <=
           | SemiColonTok           -- ;
          deriving (Show, Eq)
```

Implementing Lexer

The lexer is a function responsible for reading the input string and converting it into a list of tokens. Since the language is very simple and has very few keywords, we simply check that the input string matches any keyword or token and return the corresponding token. In the special case of an integer literal or variable name, we use `span` to read the longest number or variable name possible. If the input string does not match any keyword or token, we return an error.

A snippet of the lexer is shown below:

```
lexer :: String -> [Token]
lexer [] = []
lexer ('+': rest) = PlusTok : lexer rest
lexer ('w': 'h': 'i': 'l': 'e': rest) = WhileTok : lexer rest
-- ...
lexer (c: rest)
  | isSpace c = lexer rest
```

```
    | isDigit c = IntToken (read num) : lexer rest'
    | isLower c = VarTok var : lexer rest''
    | otherwise = error ("Invalid character: " ++ [c] ++ " in " ++ (c:rest))
    where (num, rest') = span isDigit (c:rest)
          (var, rest'') = span isAlphaNum (c:rest)
```

Parser

Defining Arithmetic Expressions

We defined a data type for all the possible arithmetic expressions in our language. An arithmetic expression can be a variable, an integer literal, a sum, a multiplication or a subtraction.

```
data Aexp = NumExp Integer
          | VarExp String
          | AddExp Aexp Aexp
          | MultExp Aexp Aexp
          | SubExp Aexp Aexp
          deriving Show
```

Defining Boolean Expressions

We also defined a data type for all the possible boolean expressions in our language. A boolean expression can be a boolean literal (True or False), an integer comparison (less than or equal `<=` or equality `==`), a boolean equality `=`, a boolean negation `not` or a boolean conjunction `and`.

```
data Bexp = TrueExp
          | FalseExp
          | LeExp Aexp Aexp
          | IntEqExp Aexp Aexp
          | BoolEqExp Bexp Bexp
          | NotExp Bexp
          | AndExp Bexp Bexp
          deriving Show
```

Defining Statements

We defined a data type for all the possible statements in our language. A statement can be an assignment, a conditional or a while loop. Note that the conditional and while loop statements may receive a list of statements as their body.

```
data Stm =  AssignStm String Aexp
          | IfStm Bexp [Stm] [Stm]
          | WhileStm Bexp [Stm]
          deriving Show
```

Parsing Expressions and Statements

After defining the necessary data types above, we implemented the parser for the list of tokens provided by the lexer. This was the most challenging part of the project, since we had to deal with the precedence of the operators and the parenthesis.

Parsing Arithmetic Expressions

To parse arithmetic expressions, we studied and implemented the example provided in the lectures. The idea is to have many functions, each responsible for parsing a specific operator, and call them according to the precedence of the operators. In our project, we start with a function `parseSumOrHigher` that calls `parseProdOrHigher` which in turn calls `parseIntParenVar`, which is responsible for parsing integer literals, variables and expressions between parenthesis. These are the ones with highest precedence. To parse an expression between parenthesis, it can simply call the higher level function `parseSumOrHigher`. After parsing these expressions, the control returns to `parseProdOrHigher`, parsing all the products, and finally to `parseSumOrHigher`, parsing all the sums and subtractions.

The first level (sums and subtractions) of the parser is shown below:

```
parseSumOrHigher :: [Token] -> Maybe (Aexp, [Token])
parseSumOrHigher tokens = case parseProdOrHigher tokens of
    Just (expr1, PlusTok : restTokens1) ->
        case parseSumOrHigher restTokens1 of
            Just (expr2, restTokens2) -> Just (AddExp expr1 expr2,
restTokens2)
            Nothing -> Nothing
    Just (expr1, MinusTok : restTokens1) ->
        case parseSumOrHigher restTokens1 of
            Just (expr2, restTokens2) -> Just (SubExp expr1 expr2,
restTokens2)
            Nothing -> Nothing
    result -> result
```

Parsing Boolean Expressions

To parse boolean expressions, we used the same idea as for arithmetic expressions. These were even more challenging to parse since there can be arithmetic expressions inside boolean expressions. To solve this issue we used the `parseSumOrHigher` function mentioned above when we reached either the 'Integer Equality' or 'Less or Equal' precedence levels, and then continued parsing the result of this function as a boolean expression again.

The Integer Equality level of the parser is shown below:

```
parseIntEqOrHigher :: [Token] -> Maybe (Bexp, [Token])
parseIntEqOrHigher tokens = case parseSumOrHigher tokens of
    Just (expr1, IntEqTok : restTokens1) ->
```

```
        case parseSumOrHigher restTokens1 of
            Just (expr2, restTokens2) -> Just (IntEqExp expr1 expr2,
restTokens2)
            Nothing -> Nothing
    result -> parseLeOrHigher tokens
```

Parsing Statements

To parse statements we just needed to break the list of tokens into smaller lists of tokens, each corresponding to a part of a statement. For example, to parse a loop statement, we first need to parse the `while` token, then the boolean expression, then the `do` token and finally the body of the loop. The only challenge here are the parenthesis: what if there is an 'if' inside a 'while' loop? To solve this issue, we created a `getBetweenParenTokens` function that returns the tokens between parenthesis, and then we call the parser recursively to parse the tokens between parenthesis.

The function responsible for parsing statements is named `buildData`. Below is an example of this function to parse a loop statement:

```
buildData (WhileTok:tokens) = WhileStm (buildBexp bexp) (buildData
doTokens) : buildData rest
    where (bexp, withDoTokens) = break (== DoTok) tokens
          (doTokens, rest) =
                if head (tail withDoTokens) == OpenParenTok then
                    getBetweenParenTokens (tail withDoTokens)
                else
                    break (== SemiColonTok) (tail withDoTokens)
```

Compiling statements

After parsing the program and obtaining a list of statements, we need to convert them into a list of instructions.

To do this, we created a function `compile` that receives the complete program and returns a list of instructions. We chose to represent a program as a list of statements.

This function is responsible for compiling each type of statement. Additionally, we created auxiliary functions to compile arithmetic expressions and boolean expressions.

For example, to compile an assignment statement, we just need to compile the arithmetic expression and then add the `Store` instruction to store the result of the expression in the variable. Below, we can see the implementation of the `compile` function for the assignment statement:

```
type Program = [Stm]
-- ...
compile :: Program -> Code
compile (AssignStm var aexp:rest) = compA aexp ++ [Store var] ++ compile
rest
```

To compile an arithmetic expression, we just need to compile each subexpression and then add the corresponding instruction. Below, we can see the implementation of the `compA` function for the sum:

```
compA :: Aexp -> Code
compA (AddExp a1 a2) = compA a2 ++ compA a1 ++ [Add]
```

To compile a boolean expression, we just need to compile each subexpression and then add the corresponding instruction. Below, we can see the implementation of the `compB` function for the conjunction:

```
compB :: Bexp -> Code
compB (AndExp b1 b2) = compB b2 ++ compB b1 ++ [And]
```

Assembler

Defining Instructions and Code

The instruction set and the code definition were given in the project template:

```
data Inst = Push Integer
          | Add
          | Mult
          | Sub
          | Tru
          | Fals
          | Equ
          | Le
          | And
          | Neg
          | Fetch String
          | Store String
          | Noop
          | Branch Code Code
          | Loop Code Code
          deriving Show

type Code = [Inst]
```

Defining the machine

To define the machine, we had to define a Stack, that may contain numeric values or boolean values, and a State, that maps variable names to numeric values. We used the stack implementation presented in the lectures and, for the State, we used the Map implementation from the exercises sheets. This Map is

implemented as a BST, so the lookup and insert operations are `O(log n)`. Furthermore, it is inherently ordered, so we can easily print the state in alphabetical order.

```
import qualified Stack
import qualified Map
--...
data Node = Num Integer
          | Tval Bool
          deriving (Show, Eq)

type NodeStack = Stack.Stack Node

type State = Map.Map String Node
```

Running code

Having defined the Instruction Set, the NodeStack and the State, our machine can be represented by this triple. Now we just need to implement the function `run` that runs Code (a list of Instructions) on the machine. To simplify the process, we can define a function `execInst` that executes a single instruction on the machine. This function is responsible for updating the Code, the NodeStack and the State.

Here is the implementation of the `run` function:

```
run :: (Code, NodeStack, State) -> (Code, NodeStack, State)
run ([], stack, state) = ([], stack, state)
run (code, stack, state) = run (ncode, nstack, nstate)
  where (ncode, nstack, nstate) = execInst (code, stack, state)
```

As we can see, `execInst` will be called instruction by instruction until the Code is empty. To implement `execInst` we just need to implement a case for each instruction. Some instructions also require to define operations for the Node type, such as the `Add` instruction.

Below, we can see the implementation of the `Add` instruction, together with its auxiliary function to add two nodes:

```
add :: Node -> Node -> Node
add (Num n1) (Num n2) = Num (n1 + n2)
add _ _ = error "Run-time error"
-- ...
execInst (Add : code, stack, state) =
    (code, Stack.push (add n1 n2) nstack, state)
        where n1 = Stack.top stack
              n2 = Stack.top (Stack.pop stack)
              nstack = Stack.pop (Stack.pop stack)
```

Conclusion

This project was very interesting and challenging. We improved our Haskell skills while learning the basics of compilers, which will serve as preparation for the next semester.