U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

PFL – 2nd Project Report

Group T08_G06

João Torre Onofre Pereira – 202108848 (50%)
Madalena Ye – 202108795 (50%)

December 2023

1 Introduction

This project aimed to implement a low-level machine and a compiler for a small imperative programming language in Haskell. The machine has instructions for arithmetic, boolean operations, and flow control. The imperative language includes expressions, assignments, sequences of statements, if-then-else constructs, and while loops. The goal is to create functions to manipulate the machine's state, interpret programs, and compile imperative language programs into machine code. A parser was also developed to convert imperative language programs from strings to literal representations.

2 Installation and Execution

To run this project, you need to:

- Install the Haskell compiler GHCi
- Open a terminal, navigate to the src folder and run ghci
- Load the main file – `':l main.hs'`

3 Code Structure

The source code is organized into distinct modules:

- Stack – implements the Stack data type, providing operations for stack manipulation.
- State – defines the State data type representing the machine's storage and includes functions for state manipulation.
- Interpreter – implements the interpreter for the low-level machine, executing a list of instructions and updating the state and the stack accordingly.

- Lexer – contains the lexer function to tokenize input strings, helping in the parsing of imperative language programs.
- Parser – implements the parser function to convert strings representing imperative language programs into their corresponding internal representations.
- Compiler – defines the compiler function to convert programs in the imperative language into lists of machine instructions for the low-level machine.

3.1 Stack

To implement the internal workings of the Stack, we defined a data type called `Stack`, implemented as a list. This list exclusively accepts elements of the `StackElement` type, limited to integers or boolean values.

```
data StackElement = IntElement Integer | BoolElement Bool
    deriving(Show, Eq)

type Stack = [StackElement]
```

In terms of stack manipulation, we defined the following functions:

```
    createEmptyStack :: Stack -- creates an empty stack
    showValue :: StackElement -> String -- shows the value of x
       depending on if x is an IntElement or BoolElement
    stack2Str :: Stack -> String -- iterates through the stack and
       prints its content
```

3.2 State

To represent the data structure State, we introduced a new data type `State` which comprises a mapping where each key is a string and the corresponding values are of the type `StackElement`, introduced in the Stack module.

```
    newtype State = State { getSate :: Map.Map String StackElement}
```

`getState` is a field accessor function that allows us to extract the map from a State data type. We also defined some helper functions:

```
    createEmptyState :: State -- creates an empty state
    insertValue :: String -> StackElement -> State -> State -- adds a
       new (key, value) to the state
    readValue :: String -> State -> StackElement -- reads the value
       mapped to a certain key from the state
    state2Str :: State -> String -- iterates through state and prints
       its content in the form key=value. We used the intercalate
       function in the Data.List module to facilitate this.
```

3.3 Interpreter

The `Inst` data type in the Interpreter module represents instructions for a low-level machine. Each constructor corresponds to an operation, such as pushing an integer onto the stack, performing addition, or evaluating a boolean equality. The `Code` type is a list of these instructions, representing a program. In this module, we have the function `interpret` that takes an instruction, a triple of code, stack, and state, and updates them based on the operation. For instance, the 'Add' construct performs addition on the top two elements of the stack.

```
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]

interpret :: Inst -> (Code, Stack, State) -> (Code, Stack, State)
```

3.4 Lexer

The Lexer module defines a set of tokens using the `Token` data type. Each constructor represents a distinct token. The function `lexer` takes a string as input and transforms it into a list of tokens based on the recognized patterns in the string, facilitating the subsequent parsing of the language.

```
data Token = TInt Integer
       | TVar String
       | TPlus | TMinus | TMult
       | TLParen | TRParen
       | TTrue | TFalse | TEquBool | TEqu | TLe | TAnd | TNot
       | TAssign | TSeq
       | TIf | TThen | TElse | TWhile | TDo
       deriving (Show, Eq)

lexer :: String -> [Token]
```

3.5 Parser

`parse` is the main function of this module and takes a string representing an imperative language program and returns its internal representation as a list of statements. It calls the `parseStm` function after the program is tokenized, which parses a sequence of tokens into a list of statements. It then uses `parseAexp` and `parseBexp` to handle arithmetic and boolean expressions, respectively.

There are also a few helper functions, like `getCode` which extracts a code block from a list of tokens or `joinStms` which joins a list of statements into a single statement, representing a sequence.

```
parse :: String -> Program
parseStm :: [Token] -> Program
parseAexp :: [Token] -> Maybe (Aexp, [Token])
parseBexp :: [Token] -> Maybe (Bexp, [Token])
getCode :: [Token] -> ([Token], [Token])
joinStms :: [Stm] -> Stm
```

3.6 Compiler

The internal representation of the compiler involves abstract syntax trees for arithmetic expressions (Aexp), boolean expressions (Bexp), and statements (Stm). The compilation functions recursively traverse these ASTs, generating machine instructions based on the structure of the input program.

```
data Aexp = Num Integer
```

```
          | Var String
          | AddExp Aexp Aexp
          | SubExp Aexp Aexp
          | MultExp Aexp Aexp
          deriving (Show)

data Bexp = TrueExp
          | FalseExp
          | EquExp Aexp Aexp -- Equality
          | EquBoolExp Bexp Bexp  -- Boolean equality
          | LeExp Aexp Aexp
          | AndExp Bexp Bexp
          | NotExp Bexp
          deriving (Show)

data Stm = Assign String Aexp
         | If Bexp Stm Stm
         | While Bexp Stm
         | Seq Stm Stm -- Sequence: stm1; stm2
         deriving (Show)

-- Definition of the Program type
type Program = [Stm]

compA :: Aexp -> Code
compB :: Bexp -> Code
compileStm :: Stm -> Code
compile : Program -> Code
```

The `compile` function takes a statement as input and compiles it into a machine of instructions. It uses `compileStm` to handle each individual statement. When `compileStm` encounters a statement that involves an arithmetic expression, it uses `compA`. Similarly, when encountering a statement that involves a boolean expression, it uses `compB`.

4 Implementation

This assignment is divided into two parts to ensure a systematic and organized development process.

4.1 First part

In the first part, we focused on implementing the low-level machine components and building the interpreter. To achieve this, we created the necessary data structures and developed the interpretation function based on the machine instructions (Inst).
For the `interpreter` function, we employed **pattern matching** to efficiently handle various instructions and states within the machine. This approach allowed us to define specific behaviours for different instruction types. Each case in the pattern matching corresponds to a different constructor of the `Inst` data type, and the associated code is executed based on the specific instructions encountered during interpretation. We also applied the pattern-matching approach to verify whether the stack contained invalid arguments for a specific instruction, and in such cases, we raised a run-time error.
For example:

```
interpret Add (code, IntElement x : IntElement y : xs, state) =
  (code, IntElement (x+y) : xs, state)
interpret Add (code, stack, state) = error $ "Run-time error"
```

Here, the first line checks whether the constructor matches the pattern for an `Add` instruction and if the top two elements of the stack are integers (IntElement). If the condition is met, it performs the addition and updates the stack with the new value. The second line is a catch-all pattern that raises a run-time error if the above conditions are not satisfied.
The stack and state are updated accordingly until the code list is empty. For operations involving fetch-x and store-x instructions, we make use of the previously defined helper functions (that take advantage of the map function in Haskell) to efficiently manage state modifications. To update the state during fetch-x and store-x instructions, we make use of the helper functions we defined, while also taking advantage of the map function in Haskell.

```
insertValue key value (State state) = State $ Map.insert key value
   state

readValue key (State state) =
  case Map.lookup key state of
    Just value -> value
    Nothing    -> error "Run-time error"

-- Fetch
interpret (Fetch key) (code, stack, state)
  | valueIsInt = (code, value : stack, state)
  | valueIsBool = (code, value : stack, state)
  | otherwise = error "Run-time error"
  where
    value = readValue key state
    valueIsInt = case value of
      IntElement _ -> True
      _ -> False
    valueIsBool = case value of
      BoolElement _ -> True
      _ -> False

-- Store
interpret (Store key) (code, value : stack, state) =
  (code, stack, insertValue key value state)
```

4.2 Second part

In the second part of the project, our attention shifted towards developing a compiler for an imperative programming language. This involved the creation of modules for lexical analysis (lexer), syntactic analysis (parser) and code generation (compiler).
First, we tokenized input programs by implementing the `lexer` function:

```
lexer [] = []
lexer input@(c:cs)
  | isDigit c = let (num, restNum) = span isDigit input
                in TInt (read num) : lexer restNum
  | isAlpha c = case keyword of
                  "and" -> TAnd : lexer restKeyword
                  "not" -> TNot : lexer restKeyword
                  "True" -> TTrue : lexer restKeyword
                  "False" -> TFalse : lexer restKeyword
                  "if" -> TIf : lexer restKeyword
                  "then" -> TThen : lexer restKeyword
                  "while" -> TWhile : lexer restKeyword
                  "do" -> TDo : lexer restKeyword
                  "else" -> TElse : lexer restKeyword
                  _      -> TVar var : lexer restVar
  | c == '+' = TPlus : lexer cs
  | c == '-' = TMinus : lexer cs
  | c == '*' = TMult : lexer cs
  | c == '(' = TLParen : lexer cs
  | c == ')' = TRParen : lexer cs
  | c == '=' && not (null cs) && c2 == '=' = TEqu : lexer newCs
  | c == '<' && not (null cs) && c2 == '=' = TLe : lexer newCs
  | c == '=' = TEquBool : lexer cs
  | c == ':' && not (null cs) && c2 == '=' = TAssign : lexer newCs
  | c == ';' = TSeq : lexer cs
  | otherwise = lexer cs -- Ignoring spaces, for example
  where (var, restVar) = span isAlpha input
        (keyword, restKeyword) = span isAlpha input
        (c2:newCs) = cs
        (followWord, restFollowWord) = span isAlpha newCs
```

The lexer recognizes different token types such as integers, variables, operators, and keywords. It uses pattern matching and guards to handle various cases. For instance, if the input starts with a digit, it recognizes an integer and continues transforming the remaining string. If the input starts with an alphabetic character, it checks for keywords and variables manually. The lexer also recognizes operators, parentheses, and other symbols. It skips spaces and handles multi-character operators like '==' appropriately, as shown in the code. The lexer processes the input recursively until the entire string is tokenized.
The resulting list of tokens is then processed by the `parseStm` function, which recursively generates a list of statements based on the recognized tokens. This function handles different types of statements such as conditional statements `TIf`, while loops `TWhile`, variable assignments `TVar`, `TAssign`, and sequence statements `TSeq`. Here's a snippet of code for a while loop:

```
parseStm (TWhile : t) =
    case parseBexp cond of
        Just (parsedCond, []) ->
            let code = joinStms (parseStm extractedCode)
            in While parsedCond code : parseStm remaining
        _ -> error "Run-time error"
    where cond = takeWhile (/= TDo) t
          afterDo = drop 1 $ dropWhile (/= TDo) t
          (extractedCode, remaining) = getCode afterDo
```

When encountering a `TWhile` token in the list of tokens `t`, we proceed to parse the boolean condition `cond` following the `TWhile`. If the parsing is successful and there are no remaining tokens in the condition, we extract the code within the while loop using the `getCode` helper function. The `While` statement is then constructed with the parsed condition and the parsed code, and the process continues with the remaining tokens (`remaining`) after the while loop. If the parsing of the boolean condition fails, it raises an error.
The `joinStms` function is used to concatenate multiple statements into a single sequence statement. Additionally, there are functions `parseAexp` and `parseBexp` for parsing arithmetic and boolean expressions, respectively.

```
parseAexp tokens = do
  (left, rest) <- parseAexpOp tokens
  parseAexpInReverse left rest
```

`parseAexpInReverse` and `parseAexpOp` contribute to parsing arithmetic expressions with operators, considering the precedence and associativity of these operators.

Furthermore, regarding boolean expressions, parseBexp serves as the entry point, initiating the parsing process by invoking parseAndExp. This function sequentially delegates to other functions (parseEquBoolExp, parseBasicExp, parseRelationalAexp to handle different aspects of boolean expressions, ensuring proper parsing of logical And, equality, and relational expressions while maintaining precedence rules. The structure follows a bottom-up approach, building up to the boolean expression from basic components. Considering:

```
parseBasicBexp :: [Token] -> Maybe (Bexp, [Token])
parseBasicBexp (TTrue : tokens) = Just (TrueExp, tokens)
parseBasicBexp (TFalse : tokens) = Just (FalseExp, tokens)
parseBasicBexp (TNot : tokens) = do
  (exp, rest) <- parseBasicBexp tokens
  return (NotExp exp, rest)
parseBasicBexp (TLParen : tokens) =
  case parseBexp tokens of
    Just (exp, TRParen : restTokens) -> Just (exp, restTokens)
    _ -> Nothing
parseBasicBexp tokens = parseRelationalAexp tokens
```

Let's now imagine the boolean expression 'not (True and False)'. If we tokenize this expression, we get the following list: [TNot, TLParen, TTrue, TAnd, TFalse, TRParen]. Now, if we apply the function to these tokens: the first token parsed is TNot, then the function calls itself recursively on the remaining ones and returns (NotExp (TrueExp AndExp FalseExp), []).

Additional note: Several parsing functions have an auxiliary function that reverses the order of the parsing. The decision to compute boolean expressions with the operators in reverse order is driven by the semantics of the operation and the nature of the stack-based interpretation of expressions.

After parsing all the tokens, we proceed to translate the AST into low-level machine instructions. We also use the pattern matching strategy for each type of expression. Below is an example of the compB function, responsible for compiling boolean expressions after the code has been parsed:

```
compB TrueExp            = [Tru]
compB FalseExp           = [Fals]
compB (EquExp x y)       = compA y ++ compA x ++ [Equ]
compB (EquBoolExp x y)   = compB y ++ compB x ++ [Equ]
compB (LeExp x y)        = compA y ++ compA x ++ [Le]
compB (AndExp x y)       = compB y ++ compB x ++ [And]
compB (NotExp x)         = compB x ++ [Neg]
```

This function takes a boolean expression and associates it with the appropriate instruction for execution in the interpreter. Since the stack has a LIFO nature, it's important to compute the second expression before the first. Moreover, due to the distinction in operators for checking equality between integer values (==) and booleans (=), separate expressions were created. Although both expressions invoke the [Equ] interpreter function, EquExp is designed for comparing integers while EquBoolExp is used for comparing boolean expressions.
The main functions of the Compiler module are:

```
compile [] = []
compile (stm:stms) = compileStm stm ++ compile stms

compileStm (Assign x a)          = compA a ++ [Store x]
compileStm (Seq stm1 stm2)       = compileStm stm1 ++ compileStm stm2
compileStm (While cond stm)      = Loop (compB cond) (compileStm stm) :
    []
compileStm (If cond stm1 stm2)  = compB cond ++ [Branch (compileStm
    stm1) (compileStm stm2)]
```

`compile` processes each statement recursively, using `compileStm` for individual statement compilation, producing machine code for the entire program.

5 Conclusions

We successfully implemented a low-level machine and a compiler for a small imperative programming language in Haskell. Our project showcased proficiency in parsing, lexing, and execution of arithmetic and boolean expressions, variable assignments, and control flow structures. It has been shown through this report that all the project's goals have been achieved. The project was developed in compliance with the given specifications and is working as intended. Lastly, the development of this assignment as a whole has proved to be a hands-on, practical learning experience.