Functional and Logic Programming - 2nd Pratical Assignment

Group Description

Group Name: **T03_G10**

Group Members:

- **João Filipe Oliveira Ramos** - up202108743 - 33,(3)% contribution
- **Marco André Pereira da Costa** - up202108821 - 33,(3)% contribution
- **Tiago Filipe Castro Viana** - up201807126 - 33,(3)% contribution

Problem Description

The goal of this pratical assignment was to consider a low-level machine with configurations of the form (Code, Stack, State) that executes a given program, and to implement the corresponding assembler, parser and compiler for this machine. The machine has a **Code** component that consists of a list of instructions, an evaluation **Stack** and a **State** component to function as storage.

In the first part of the assignment, we were asked to implement an assembler for the low-level machine. The assembler is responsible for processing the instructions and returning the resulting **Stack** and **State**. The second part required us to implement a parser and a compiler for the same machine. The parser will parse the program, converting it into a list of **statements**, and the compiler will process the list of **statements** and return the resulting **Code**.

Part I - Assembler Implementation Description

Data Types

The low-level machine we are implementing consists of a tuple of three elements, where the first one is the **code**, the second is the **stack** and the third is the **state**.

The **code** type was already defined by the pratical assignment template.

For the **stack**, we decided to create a new type called **StackData** which can hold either an Integer or a Boolean value. The **stack** would then be defined as a list of **StackData**.

These types are defined as follows:

```
-- StackData represents the different data types that can be held in the Stack
type
data StackData
  = I Integer
  | B Bool
  deriving (Show, Eq)
```

```haskell
-- Stack represents a custom stack-like data structure
type Stack = [StackData]
```

This definition allows us to process the different types of data that can be stored in the **stack** abstractly, without having to worry about the type of data that is being processed. This is useful because it allows us to implement the **stack** operations in a more generic fashion, without having to implement them for each data type.

However, in order to implement the <= operation, we needed to be able to compare the values of the **stack** elements. Therefore, we established the ordering rules for the **StackData** type, by implementing an **Ord** typeclass instance, as follows:

```haskell
-- Ord StackData establishes how the StackData type is ordered
instance Ord StackData where
  compare :: StackData -> StackData -> Ordering
  compare (I n1) (I n2) = compare n1 n2
  compare (B b1) (B b2) = compare b1 b2
  compare _ _ = error "Run-time error"
```

This approach allows us to compare the values of the **stack** elements. If the values are of different types, a *Run-time error* is thrown.

As for the **state**, we decided to create a new type called **StateData**, which consists of a tuple containing a **String** and **StackData**. The **state** would then be defined as a list of **StateData** that represents the storage, through which we can store or retrieve variable values.

These types are defined as follows:

```haskell
-- StateData is a tuple that contains a String and a StackData, where the String
represents a variable's name, and the StackData corresponds to its value
type StateData = (String, StackData)

-- State is a list of StateData. It represents the storage
type State = [StateData]
```

This definition allows us to identify the tuples by the name of the variable, which is useful when we need to retrieve or update the value of a variable. This is also useful for the implementation of the *state2Str* function, which will need to print the **state** ordered by the name of the variables.

Functions

In order to create an empty **stack** and **state**, we implemented the following functions:

```haskell
-- Creates an empty Stack
createEmptyStack :: Stack
createEmptyStack = []
```

```haskell
-- Creates an empty State
createEmptyState :: State
createEmptyState = []
```

In order to convert a **stack** into a **string**, where the elements of the **stack** are separated by a comma (without spaces), we implemented the following function:

```haskell
-- Converts a Stack into a String
stack2Str :: Stack -> String
stack2Str [] = ""
stack2Str [h] = case h of
  I i -> show i
  B b -> show b
stack2Str (h : t) = case h of
  I i -> show i ++ "," ++ stack2Str t
  B b -> show b ++ "," ++ stack2Str t
```

In order to convert a **state** into a **string**, where the elements of the **state** are separated by a comma (without spaces), we implemented the following function:

```haskell
-- Converts a State into a String
state2StrAux :: State -> String
state2StrAux [] = ""
state2StrAux [(var, value)] = case value of
  I i -> var ++ "=" ++ show i
  B b -> var ++ "=" ++ show b
state2StrAux ((var, value) : state) = case value of
  I i -> var ++ "=" ++ show i ++ "," ++ state2StrAux state
  B b -> var ++ "=" ++ show b ++ "," ++ state2StrAux state

-- Sorts a State and converts it into a String
state2Str :: State -> String
state2Str = state2StrAux . sort
```

In this function, we use the *sort* function to sort the **state** by the name of the variables, so that the **state** is displayed in alphabetical order.

Finally, in order to process the **code**, we implemented the following function:

```haskell
-- Executes a program (a list of instructions) for a given stack and state,
returning the resulting stack and state
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack, state) = ([], stack, state)
run (inst : code, stack, state) =
  case inst of
    Push n -> run (code, I n : stack, state)
```

```
    Add -> run (code, arithmeticOp (+) stack, state)
    Mult -> run (code, arithmeticOp (*) stack, state)
    Sub -> run (code, arithmeticOp (-) stack, state)
    Tru -> run (code, B True : stack, state)
    Fals -> run (code, B False : stack, state)
    Equ -> run (code, comparisonOp (==) stack, state)
    Le -> run (code, comparisonOp (<=) stack, state)
    And -> run (code, logicalOp (&&) stack, state)
    Neg -> run (code, unaryOp not stack, state)
    Fetch var -> run (code, fetch var state : stack, state)
    Store var -> run (code, tail stack, store var (head stack) state)
    Noop -> run (code, stack, state)
    Branch code1 code2 -> if head stack == B True then run (code1 ++ code, tail
stack, state) else run (code2 ++ code, tail stack, state)
    Loop code1 code2 -> run (code1 ++ [Branch (code2 ++ [Loop code1 code2])
[Noop]] ++ code, stack, state)

  where
    -- Auxiliary functions

    -- Executes an arithmetic operation on the top two elements of the stack
    arithmeticOp :: (Integer -> Integer -> Integer) -> Stack -> Stack
    arithmeticOp op (I n1 : I n2 : stack) = I (n1 `op` n2) : stack
    arithmeticOp op _ = error "Run-time error"

    -- Executes a comparison operation on the top two elements of the stack
    comparisonOp :: (StackData -> StackData -> Bool) -> Stack -> Stack
    comparisonOp op (v1 : v2 : stack) = case (v1, v2) of
      (I _, B _) -> error "Run-time error"
      (B _, I _) -> error "Run-time error"
      _ -> B (v1 `op` v2) : stack

    -- Executes a binary logical operation on the top two elements of the stack
    logicalOp :: (Bool -> Bool -> Bool) -> Stack -> Stack
    logicalOp op (B b1 : B b2 : stack) = B (b1 `op` b2) : stack
    logicalOp op _ = error "Run-time error"

    -- Executes an unary logical operation on the top element of the stack
    unaryOp :: (Bool -> Bool) -> Stack -> Stack
    unaryOp op (B b : stack) = B (op b) : stack
    unaryOp op _ = error "Run-time error"

    -- Stores a variable and its corresponding value in the storage. If the
variable is already stored, its value is updated
    store :: String -> StackData -> State -> State
    store var value [] = [(var, value)]
    store var value ((var', value') : state)
      | var == var' = (var', value) : state
      | otherwise = (var', value') : store var value state

    -- Fetches a given variable's value from the storage. If the variable does not
exist in the storage, raises an exception "Run-time error"
    fetch :: String -> State -> StackData
```

```
    fetch var [] = error "Run-time error"
    fetch var ((var', value) : t) = if var == var' then value else fetch var t
```

This function receives a tuple of 3 elements, where the first element is the **code** (a list of instructions), the second element is the **stack** and the third element is the **state**. It then processes the **code** and returns a tuple of 3 elements, where the first element is the remaining **code** (should be empty), the second element is the resulting **stack** and the third element is the resulting **state**.

Each instruction is processed one at a time, and the resulting **code**, **stack** and **state** are passed to the next instruction. This is done recursively until the **code** is empty. Each case of the *run* function corresponds to a different instruction, identified by pattern matching. The *run* function also uses auxiliary functions to process the different types of instructions.

The *op* auxiliary functions are used to process the arithmetic, comparison and logical operations. These functions receive a function that corresponds to the operation to be performed, and the **stack**. The *op* functions then apply the operation to the top elements of the **stack** and return the resulting **stack**. If the **stack** does not have enough elements to perform the operation, a *Run-time error* is thrown.

The *store* auxiliary function is used to store a variable and its corresponding value (top of the stack) in the **state**. If the variable is already stored, its value is updated. This function receives the variable name, the value to be stored and the **state**. The *store* function then searches for the variable in the **state** and updates its value if it exists, or adds the variable and its value to the **state** if it does not exist.

The *fetch* auxiliary function is used to fetch a variable's value from the **state**. If the variable does not exist in the **state**, a *Run-time error* is thrown. This function receives the variable name and the **state**. The *fetch* function then searches for the variable in the **state** and returns its value if it exists, or throws a *Run-time error* if it does not exist.

Test Examples

```
ghci> testAssembler [Push 10,Push 4,Push 3,Sub,Mult] == ("-10","")
True
ghci> testAssembler [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"] ==
("","a=3,someVar=False,var=True")
True
ghci> testAssembler [Fals,Store "var",Fetch "var"] == ("False","var=False")
True
ghci> testAssembler [Push (-20),Tru,Fals] == ("False,True,-20","")
True
ghci> testAssembler [Push (-20),Tru,Tru,Neg] == ("False,True,-20","")
True
ghci> testAssembler [Push (-20),Tru,Tru,Neg,Equ] == ("False,-20","")
True
ghci> testAssembler [Push (-20),Push (-21), Le] == ("True","")
True
ghci> testAssembler [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"] ==
("","x=4")
True
ghci> testAssembler [Push 10,Store "i",Push 1,Store "fact",Loop [Push 1,Fetch
```

```
"i",Equ,Neg] [Fetch "i",Fetch "fact",Mult,Store "fact",Push 1,Fetch "i",Sub,Store
"i"]] == ("","fact=3628800,i=1")
True
ghci> testAssembler [Push 1,Push 2,And]
("*** Exception: Run-time error
CallStack (from HasCallStack):
  error, called at main.hs:128:22 in main:Main
ghci> testAssembler [Tru,Tru,Store "y", Fetch "x",Tru]
("True,*** Exception: Run-time error
CallStack (from HasCallStack):
  error, called at main.hs:144:20 in main:Main
```

Part II - Parser and Compiler Implementation Description

Data Types

In order to process the code provided in string format, we created a **Token** data type, which represents all the possible keywords and symbols that can be used in the language. This type is defined as follows:

```haskell
-- Token represents the type of tokens used by the lexer to parse the program
data Token
  = TokAssign
  | TokSemicolon
  | TokOpenBracket
  | TokCloseBracket
  | TokIf
  | TokThen
  | TokElse
  | TokWhile
  | TokDo
  | TokNot
  | TokAnd
  | TokOr
  | TokTrue
  | TokFalse
  | TokAEq
  | TokBEq
  | TokLeq
  | TokPlus
  | TokMinus
  | TokMult
  | TokNum Integer
  | TokVar String
  deriving (Show, Eq)
```

The **Token** data type is used by the *lexer* to parse the program and convert it into a list of **Tokens**.

In order to represent any arithmetic or boolean expression, we created the following data types:

```haskell
-- Aexp represents all possible arithmetic expressions
data Aexp
  = AddAx Aexp Aexp
  | MultAx Aexp Aexp
  | SubAx Aexp Aexp
  | NumAx Integer
  | VarAx String
  deriving (Eq, Show)

-- Bexp represents all possible boolean expressions
data Bexp
  = AndBx Bexp Bexp
  | EqBx Bexp Bexp
  | EqAx Aexp Aexp
  | NegBx Bexp
  | LeqAx Aexp Aexp
  | Bx Bool
  | VarBx String
  deriving (Eq, Show)
```

These types are evaluated by the functions *compA* and *compB*, which are used by the *compile* to process the list of **Stms** and convert it into a list of **Instructions**.

To facilitate the compilation of the program, we created the following data type:

```haskell
-- Stm represents all available statements
data Stm
  = AssignBx String Bexp
  | AssignAx String Aexp
  | Conditional Bexp Stm Stm
  | While Bexp Stm
  | Seq [Stm]
  deriving (Show)

-- Program is a list of Stm. It represents a set of statements
type Program = [Stm]
```

The **Stm** data type allows us to represent all the possible statements that can be used in the language:

- Assignment of a boolean expression (**Bexp**), to a variable (**String**)
- Assignment of an arithmetic expression (**Aexp**) to a variable (**String**)
- Conditional, which takes a boolean expression (**Bexp**) and two statements (**Stm**) as arguments
- Loop, which takes a boolean expression (**Bexp**) and a statement (**Stm**) as argument
- Sequence of statements, which takes a list of statements (**[Stm]**) as argument

This type is used by the *parse* function to parse the string into the corresponding **Program**, a list of **Stms**. The *compile* function then uses the **Program** to convert it into **Code**, a list of **Instructions**.

Functions

In order to streamline the parsing of the program, we started by implementing a *lexer* auxiliary function that splits the given string into a list of **Tokens**. This function will then be used along with the *buildData* function to build the corresponding list of **Stms**.

The *lexer* function is defined as follows:

```
-- Converts a given string into a list of tokens
lexer :: String -> [Token]
lexer [] = []
lexer (':' : '=' : rest) = TokAssign : lexer rest
lexer ('+' : rest) = TokPlus : lexer rest
lexer ('-' : rest) = TokMinus : lexer rest
lexer ('*' : rest) = TokMult : lexer rest
lexer (';' : rest) = TokSemicolon : lexer rest
lexer ('(' : rest) = TokOpenBracket : lexer rest
lexer (')' : rest) = TokCloseBracket : lexer rest
lexer ('=' : '=' : rest) = TokAEq : lexer rest
lexer ('<' : '=' : rest) = TokLeq : lexer rest
lexer ('=' : rest) = TokBEq : lexer rest
lexer ('i' : 'f' : rest) = TokIf : lexer rest
lexer ('t' : 'h' : 'e' : 'n' : rest) = TokThen : lexer rest
lexer ('e' : 'l' : 's' : 'e' : rest) = TokElse : lexer rest
lexer ('w' : 'h' : 'i' : 'l' : 'e' : rest) = TokWhile : lexer rest
lexer ('d' : 'o' : rest) = TokDo : lexer rest
lexer ('n' : 'o' : 't' : rest) = TokNot : lexer rest
lexer ('a' : 'n' : 'd' : rest) = TokAnd : lexer rest
lexer ('o' : 'r' : rest) = TokOr : lexer rest
lexer ('T' : 'r' : 'u' : 'e' : rest) = TokTrue : lexer rest
lexer ('F' : 'a' : 'l' : 's' : 'e' : rest) = TokFalse : lexer rest
lexer (c : rest)
  | isDigit c = TokNum (read (c : takeWhile isDigit rest)) : lexer (dropWhile
isDigit rest)
  | isSpace c = lexer rest
  | isLower c && isAlpha c = TokVar (c : takeWhile isAlphaNum rest) : lexer
(dropWhile isAlphaNum rest)
  | otherwise = error "Run-time error"
```

The *buildData* function is defined as follows:

```
-- Builds a program from a given list of tokens
buildData :: [Token] -> Program
buildData [] = []
buildData tokens =
  case tokens of
    (TokVar var : TokAssign : restTokens) ->
      case currentStatement restTokens [] of
        Just (stm, restTokens1) -> buildAssignment (init stm) var : buildData
restTokens1
        Nothing -> error "Run-time error"
    (TokOpenBracket : restTokens) ->
```

```
      case currentStatement restTokens [TokOpenBracket] of
        Just (stm, restTokens1) -> Seq (buildData (init (init stm))) : buildData
restTokens1
        Nothing -> error "Run-time error"
    (TokIf : restTokens) ->
      case getCondition TokThen restTokens of
        (bexp, restTokens1) -> case getCurrentStatement restTokens1 of
          Just (stm1, restTokens2) -> case getCurrentStatement restTokens2 of
            Just (stm2, restTokens3) -> Conditional bexp (head stm1) (head stm2) :
buildData restTokens3
            Nothing -> error "Run-time error"
          Nothing -> error "Run-time error"
    (TokWhile : restTokens) ->
      case getCondition TokDo restTokens of
        (bexp, restTokens1) -> case getCurrentStatement restTokens1 of
            Just (stm, restTokens2) -> While bexp (head stm) : buildData
restTokens2
            Nothing -> error "Run-time error"
    _ -> error "Run-time error"

  where
    -- Auxiliary functions

    -- Retrieves the respective list of tokens for the first (current) statement,
along with the list of the following tokens
    currentStatement :: [Token] -> [Token] -> Maybe ([Token], [Token])
    currentStatement (TokSemicolon : restTokens) [] = Just ([TokSemicolon],
restTokens)
    currentStatement (TokOpenBracket : restTokens) stack =
      case currentStatement restTokens (TokOpenBracket : stack) of
        Just (current, next) -> Just (TokOpenBracket : current, next)
        Nothing -> Nothing
    currentStatement (TokCloseBracket : restTokens) (TokOpenBracket : stack) =
      case currentStatement restTokens stack of
        Just (current, next) -> Just (TokCloseBracket : current, next)
        Nothing -> Nothing
    currentStatement (TokCloseBracket : restTokens) _ = Nothing
    currentStatement (TokIf : restTokens) stack =
      case currentStatement restTokens (TokIf : stack) of
        Just (current, next) -> Just (TokIf : current, next)
        Nothing -> Nothing
    currentStatement (TokThen : restTokens) (TokIf : stack) =
      case currentStatement restTokens (TokThen : TokIf : stack) of
        Just (current, next) -> Just (TokThen : current, next)
        Nothing -> Nothing
    currentStatement (TokThen : restTokens) _ = Nothing
    currentStatement (TokElse : restTokens) (TokThen : TokIf : stack) =
      case currentStatement restTokens stack of
        Just (current, next) -> Just (TokElse : current, next)
        Nothing -> Nothing
    currentStatement (TokElse : restTokens) _ = Nothing
    currentStatement (TokWhile : restTokens) stack =
      case currentStatement restTokens (TokWhile : stack) of
        Just (current, next) -> Just (TokWhile : current, next)
```

```
        Nothing -> Nothing
    currentStatement (TokDo : restTokens) (TokWhile : stack) =
      case currentStatement restTokens stack of
        Just (current, next) -> Just (TokDo : current, next)
        Nothing -> Nothing
    currentStatement (TokDo : restTokens) _ = Nothing
    currentStatement (token : restTokens) stack =
      case currentStatement restTokens stack of
        Just (current, next) -> Just (token : current, next)
        Nothing -> Nothing
    currentStatement [] stack = Nothing

    -- Retrieves the first (current) boolean expresion for a given list of tokens,
along with the list of the following tokens
    getCondition :: Token -> [Token] -> (Bexp, [Token])
    getCondition diffToken tokens = (buildBool (takeWhile (/= diffToken) tokens),
dropWhile (/= diffToken) tokens)

    -- Retrieves the first (current) statement for a given list of tokens, along
with the list of the following tokens
    getCurrentStatement :: [Token] -> Maybe ([Stm], [Token])
    getCurrentStatement tokens =
      case currentStatement (tail tokens) [] of
        Just (current, next) -> Just (buildData current, next)
        Nothing -> Nothing
```

This function receives a list of **Tokens** and converts it into a list of **Stms**. It achieves this by performing pattern matching of the first **Tokens** of the list. It then determines, for each type of statement, the respective arguments through the use of the auxiliary functions: *currentStatement*, *getCondition* and *getCurrentStatement*.

The *currentStatement* function is responsible for retrieving the respective list of **Tokens** for the first statement in the list, along with the list of the following and remaining **Tokens**. It achieves this by recursively calling itself, while keeping track of the **Tokens** that have been processed, and the **Tokens** that are still to be processed. It uses a stack to keep track of statements that are inside brackets, if statements or while statements, in a PDA-like fashion, thus allowing it to handle nested statements.

The *getCondition* function is responsible for retrieving the first boolean expression in the list of **Tokens**, along with the list of the following and remaining **Tokens**. It achieves this by taking the **Tokens** until it finds a token that corresponds to the end of the boolean expression, which is passed as an argument to the function.

The *getCurrentStatement* function is tasked with obtaining the initial statement from a list of **Tokens**, along with the subsequent and remaining **Tokens**. It accomplishes this by invoking the *currentStatement* function with an empty stack and the list of **Tokens** excluding the first one. In this particular context, the omission of the first token corresponds to either a **TokThen**, **TokElse**, or **TokDo**, each of which serves as an indicator for specific branching or looping conditions within the subsequent statements. Following this, the *buildData* function is called to construct the corresponding **Stms**.

For the case of the assignment statement, we implemented the *isBooleanExp* and the *buildAssignment* functions to handle the assignment of different types of expressions. These functions are defined as follows:

```haskell
-- Checks if a given list of tokens represents a boolean expression
isBooleanExp :: [Token] -> Bool
isBooleanExp = any (\x -> x `elem` [TokTrue, TokFalse, TokNot, TokAnd, TokAEq,
TokBEq, TokLeq])
```

```haskell
-- Builds an assignment statement from a given list of tokens
buildAssignment :: [Token] -> String -> Stm
buildAssignment tokens var
  | isBooleanExp tokens = AssignBx var (buildBool tokens)
  | otherwise = AssignAx var (buildArithmetic tokens)
```

The *isBooleanExp* function checks if the given list of **Tokens** represents a boolean expression. This is done by checking if the list of **Tokens** contains any of the boolean operators or operands.

The *buildAssignment* function receives a list of **Tokens** and a variable name. It then checks if the list of **Tokens** represents a boolean expression, by calling the *isBooleanExp* function. If it does, it calls the *buildBool* function to build the corresponding boolean expression. Otherwise, it calls the *buildArithmetic* function to build the corresponding arithmetic expression.

The *buildArithmetic* and *buildBool* functions are defined as follows:

```haskell
-- Builds an arithmetic expression from a given list of tokens
buildArithmetic :: [Token] -> Aexp
buildArithmetic tokens =
  case parseSumOrDiffOrProdOrIntOrPar (reverse tokens) of
    Just (expr, []) -> expr
    _ -> error "Run-time error"
```

```haskell
-- Builds a boolean expression from a given list of tokens
buildBool :: [Token] -> Bexp
buildBool tokens =
  case parseAndOrBEqOrNotOrBoolOrPar (reverse tokens) of
    Just (expr, []) -> expr
    _ -> error "Run-time error"
```

These functions allow us to parse the provided list of **Tokens** respectively into an arithmetic expression or boolean expression. They achieve this by recursively calling a sequence of functions, in which each function is responsible for parsing a specific operation or operand, by pattern matching.

The sequence is ordered from the lowest to the highest level of precedence and each function creates a call chain all the way to the function responsible for the highest precedence level, before trying to effectively match the resulting expression of the recursive call, with the current parsing function pattern. The parsing sequence starts by calling the function responsible for parsing the lowest level of precedence.

Because the operand parsing functions execute the pattern matching by checking the head of the list and then parsing with the same function (same level of precedence) the rest of the tokens, the parsing process resolved the expressions with right associativity. To solve this, we decided to reverse the list of tokens before parsing, so that the expression was parsed with left associativity.

By using this approach, we can simply define the *parse* function as follows:

```
-- Parses a given string into a program (list of statements) for the compiler to
execute
parse :: String -> Program
parse = buildData . lexer
```

This function receives a string and converts it into a list of **Tokens** by calling the *lexer* function. It then converts the list of **Tokens** into a list of **Stms** by calling the *buildData* function, which uses the *buildArithmetic* and *buildBool* functions to handle the parsing of expressions, according to the precedence of operations.

In order to compile the arithmetic expressions, we implemented the following function:

```
-- Compiles an arithmetic expression into a list of instructions
compA :: Aexp -> Code
compA aexp =
  case aexp of
    AddAx aexp1 aexp2 -> compA aexp2 ++ compA aexp1 ++ [Add]
    SubAx aexp1 aexp2 -> compA aexp2 ++ compA aexp1 ++ [Sub]
    MultAx aexp1 aexp2 -> compA aexp2 ++ compA aexp1 ++ [Mult]
    NumAx n -> [Push n]
    Va
    rAx v -> [Fetch v]
```

This function converts the arithmetic expression into a list of instructions, which can then be processed by the assembler.

For the boolean expressions, we implemented the following function:

```
-- Compiles a boolean expression into a list of instructions
compB :: Bexp -> Code
compB bexp =
  case bexp of
    AndBx b1 b2 -> compB b2 ++ compB b1 ++ [And]
    EqBx b1 b2 -> compB b2 ++ compB b1 ++ [Equ]
    EqAx a1 a2 -> compA a2 ++ compA a1 ++ [Equ]
    NegBx b -> compB b ++ [Neg]
    LeqAx a1 a2 -> compA a2 ++ compA a1 ++ [Le]
    Bx b -> if b then [Tru] else [Fals]
    VarBx v -> [Fetch v]
```

This is responsible for converting boolean expressions into a list of instructions, which can then be processed by the assembler.

Finally, in order to compile statements, and consequently the program, we implemented the following function:

```
-- Compiles a program into a list of instructions
compile :: Program -> Code
compile [] = []
compile (stm : program) =
  case stm of
    AssignBx var bexp -> compB bexp ++ [Store var] ++ compile program
    AssignAx var aexp -> compA aexp ++ [Store var] ++ compile program
    Conditional bexp stm1 stm2 ->
      compB bexp ++ [Branch (compile [stm1]) (compile [stm2])] ++ compile program
    While bexp stm -> Loop (compB bexp) (compile [stm]) : compile program
    Seq stms -> compile stms ++ compile program
```

This function matches all types of statements and converts them into a list of instructions, which can then be processed by the assembler.

Test Examples

```
ghci> testParser "x := 5; x := x - 1;" == ("","x=4")
True
ghci> testParser "x := 0 - 2;" == ("","x=-2")
True
ghci> testParser "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;" ==
("","y=2")
True
ghci> testParser "x := 42; if x <= 43 then x := 1; else (x := 33; x := x+1;);" ==
("","x=1")
True
ghci> testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1;" ==
("","x=2")
True
ghci> testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1; z :=
x+x;" == ("","x=2,z=4")
True
ghci> testParser "x := 44; if x <= 43 then x := 1; else (x := 33; x := x+1;); y :=
x*2;" == ("","x=34,y=68")
True
ghci> testParser "x := 42; if x <= 43 then (x := 33; x := x+1;); else x := 1;" ==
("","x=34")
True
ghci> testParser "if (1 == 0+1 = 2+1 == 3) then x := 1; else x := 2;" ==
("","x=1")
True
ghci> testParser "if (1 == 0+1 = (2+1 == 4)) then x := 1; else x := 2;" ==
("","x=2")
```

```
True
ghci> testParser "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);" ==
("","x=2,y=-10,z=6")
True
ghci> testParser "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i
:= i - 1;);" == ("","fact=3628800,i=1")
True
ghci> testParser "x := not True;" == ("", "x=False")
True
ghci> testParser "x := True and False;" == ("", "x=False")
True
ghci> testParser "x := True and not False;"      == ("", "x=True")
True
ghci> testParser "x := not (True and False);" == ("", "x=True")
True
ghci> testParser "x := (True and not False) and (not False);" == ("", "x=True")
True
ghci> testParser "x := 2 <= 5;" == ("", "x=True")
True
ghci> testParser "x := 2 == 5;" == ("", "x=False")
True
ghci> testParser "x := 2 + 3 == 5;" == ("", "x=True")
True
ghci> testParser "x := not (3 <= 1) and 4 == 2+2;" == ("", "x=True")
True
ghci> testParser "x := not (3 <= 1);" == ("", "x=True")
True
ghci> testParser "x := True = False;" == ("", "x=False")
True
ghci> testParser "x := True = False and True = False;" == ("", "x=False")
True
ghci> testParser "x := True = (1 <= 2);" == ("", "x=True")
True
ghci> testParser "x := 1+2-3+10;" == ("","x=10")
True
ghci> testParser "x := ((1)+(2) * 3 - (4 * 5) + ((((6)))) * 7;" == ("","x=-49")
True
ghci> testParser "x := (1 + 2 * 3 - 4 * 5 + 6) * 7;" == ("","x=-49")
True
ghci> testParser "x := (1 + (2 * 3) - (4 * 5) + 6) * 7;" == ("","x=-49")
True
ghci> testParser "x := ((1)+(2) * 3 - ((4 * 5) + (((((6))))) * 7;" == ("","x=-133")
True
ghci> testParser "x := (1 + 2 * 3 - (4 * 5 + 6)) * 7;" == ("","x=-133")
True
ghci> testParser "if True then if False then x := 1; else x := 2; else x := 3;" ==
("","x=2")
True
ghci> testParser "x := 0; while x <= 5 do (y := 0; while y <= 5 do (y := y + 1;);
x := x + 1;);" == ("","x=6,y=6")
True
ghci> testParser "x := 1; while x <= 5 do if True then x := x + 1; else x := x +
2;" == ("","x=6")
True
```

```
ghci> testParser "x := 1; if False then x := 1; else while x <= 5 do x := x + 1;"
== ("","x=6")
True
ghci> testParser "x := 1; if True then while x <= 5 do x := x + 1; else x := 1;"
== ("","x=6")
True
ghci> testParser "(x := 1; y := 1; while (x <= 3) do (y := 2 * y; x := x + 1; z :=
0; while (z <= 2) do (y := y + 1; z := z + 1;);););" == ("","x=4,y=29,z=3")
True
ghci> testParser "if True then (if False then x:=1; else ((x := 1; y := 1; while
(x <= 3) do (y := 2 * y; x := x + 1; z := 0; while (z <= 2) do (y := y + 1; z := z
+ 1;);););w:=1;);); else y:=1;" == ("","w=1,x=4,y=29,z=3")
True
testParser "x := 1; if True then (if False then x:=1; else ((x := 1; y := 1; while
(x <= 3) do (if True then a:=1; else b:=2; y := 2 * y; x := x + 1; z := 0; while
(z <= 2) do (y := y + 1; z := z + 1;);););w:=1;);); else y:=1;" ==
("","a=1,w=1,x=4,y=29,z=3")
True
```

Conclusions

The assembler, compiler and parser were fully implemented as expected. The Haskell program is able to process the provided code and produce the expected results, both for the tests provided by the pratical assignment template, and for the tests we implemented.

During the development of this assignment, we were able to learn more about the Haskell programming language, as well as the functional programming paradigm. We were also able to learn more about the implementation of assemblers, compilers and parsers, which allowed us to better understand how programming languages are processed and executed. The components that stood out as the most challenging was the parser, due to the complexity of the parsing functions and the need to handle the precedence of the operations, along with the implementation of nested loops and conditional statements, due to the challenge of determining the respective nested statements, while keeping track of the remaining ones and ensuring correct parsing.