Functional and Logical Programming

2nd Practical Project - Haskell

> This project was divided into two halfs, and so we decided to also divide the report into two halfs. Descriptions and explanation of our thought process as well how we solved the problems are included in their respective part.

Team

Group: `T06_G13`

Name	Student Number	Contribution (%)
Diogo Tomás Valente Fernandes	up202108752	50%
Hélder Gabriel Silva Costa	up202108719	50%

Part 1

Description

For the first half the challenge was to consider a low-level machine with configurations of the form (c, e, s) where c is a list of instructions (or code) to be executed, e is the evaluation stack, and s is the storage, and the following instructions push-n, add, mult, sub, true, false, eq, le, and, neg, fetch-x, store-x, noop, branch(c1, c2) and loop(c1, c2).

In order to complete this, first we started by developing two new types, one to represent the machine's state and another to represent the machine's stack. These were done in the *Stack.hs* and *State.hs* files.

For the Stack this is the code that defines the new data type as well as the constructor for an empty stack:

```
-- Definition of new type for a stack of values.
newtype Stack = Stack [Val]

-- Function to create an empty stack.
createEmptyStack :: Stack
createEmptyStack = Stack []
```

For the machine's state this is the code that defines the new data type as well as the constructor for a new (empty) State. The code that defines a variable is also included:

```
-- Definition of the new type Var (variable)
type Var = String

-- Definition for the machine's State.
type State = Map.Map Var Val

-- Function to create an empty state.
createEmptyState :: State
createEmptyState = Map.empty
```

After this we developed the *stack2Str* function that is responsible for transforming the given stack to a string following the restrictions given in the project assignment paper. This is it's respective code:

```
-- Function responsible for converting the stack to a string.
stack2Str :: Stack -> String
stack2Str (Stack xs) = intercalate "," (map value2Str xs)
```

The same was done for the state with the *state2Str* function. We also made an helper function for tranforming a variable-value pair to a string. Here's the respective code:

```
-- Function to convert a state to a string.
state2Str :: State -> String
state2Str state
  | Map.null state = ""
  | otherwise = intercalate "," (map entryToStr (Map.toList state))

-- Function to convert a variable-value pair to a string.
entryToStr :: (Var, Val) -> String
entryToStr (var, val) = var ++ "=" ++ value2Str val
```

Finally, and the most challenging part of this half, we needed to develop an interpreter for programs in this machine that takes a list of instructions, an empty and a state and runs the given instructions returning an empty code list, a stack and the output values in the state.

For this in the *Inst.hs* file, firstly we defined a new data type that corresponds to individual instructions as well as an alias (Code) for a list of instructions. All the instructions suggested in the assignment were included, here's the code:

```
-- Definition of the 'Inst' data type representing individual instructions.
data Inst =
    Push Integer      -- Pushes an integer onto the stack
  | Add               -- Addition operation
  | Mult              -- Multiplication operation
  | Sub               -- Subtraction operation
  | Tru               -- Pushes True to the stack
  | Fals              -- Pushes False to the stack
  | Equ               -- Equality comparison
```

```
  | Le                -- Less than or equal comparison
  | And               -- Logical AND operation
  | Neg               -- Logical NOT operation
  | Fetch String      -- Fetches the value of a variable from the state
  | Store String      -- Stores a value into a variable in the state
  | Noop              -- No-operation
  | Branch Code Code  -- Conditional branching based on the top of the stack
  | Loop Code Code    -- Loop construction
  deriving Show

-- Definiton of alias for a list of instructions.
type Code = [Inst]
```

After this in the *Interpreter.hs* file we developed the **exec** function that is responsible for executing a single step (instruction), in that function we also implement all the code needed for the instructions to work. Here's the function:

```
-- Function to execute a step.
exec :: (Code, Stack, State) -> Maybe (Code, Stack, State)
exec ([], stack, state) = Nothing
exec (inst:xs, stack, state) =
    case inst of
        -- Code responsible for push a value on to the stack.
        Push val ->
            let newStack = Stack.push stack (Integer val)
            in Just (xs, newStack, state)
        -- Code responsible for taking two values from the stack and adding them.
        Add ->
            case (top stack, top (pop stack)) of
                (Integer a, Integer b) ->
                    let newStack = Stack.push (pop (pop stack)) (Integer (a + b))
                    in Just (xs, newStack, state)
                _ -> error "Run-time error"
        -- Code responsible for taking two values from the stack and multiplying them.
        Mult ->
            case (top stack, top (pop stack)) of
                (Integer a, Integer b) ->
                    let newStack = Stack.push (pop (pop stack)) (Integer (a * b))
                    in Just (xs, newStack, state)
                _ -> error "Run-time error"
        -- Code responsible for taking two values from the stakc and subtracting them.
        Sub ->
            case (top stack, top (pop stack)) of
                (Integer a, Integer b) ->
                    let newStack = Stack.push(pop(pop stack)) (Integer (a - b))
                    in Just (xs,newStack, state)
                _ -> error "Run-time error"
        -- Code responsible for pushing True on to the stack.
        Tru ->
            let newStack = Stack.push stack Tt
            in Just (xs, newStack, state)
        -- Code responsible for pushing False on to the stack.
        Fals ->
            let newStack = Stack.push stack Ff
            in Just (xs, newStack, state)
        -- Code responsible for checking if the top two values on the stack are equal. If so push True to the stack, otherw
        Equ ->
            case (top stack, top (pop stack)) of
                (val1, val2) | val1 == val2 ->
                    let newStack = Stack.push (pop (pop stack)) Tt
                    in Just (xs, newStack, state)
                _ ->
                    let newStack = Stack.push (pop (pop stack)) Ff
                    in Just (xs, newStack, state)
        -- Code responsible for checking if the top value on the stack is lesser or equals than the one under it. If it is
        Le ->
            case (top stack, top (pop stack)) of
                (Integer a, Integer b) ->
                    let newStack = Stack.push (pop (pop stack)) (if a <= b then Tt else Ff)
                    in Just (xs, newStack, state)
                _ -> error "Run-time error"
        -- Code responsible for taking the top two values from the stack and applying a logical AND to them. Pushes the res
        And ->
            case (top stack, top (pop stack)) of
                (Tt, Tt) ->
                    let newStack = Stack.push (pop (pop stack)) Tt
                    in Just (xs, newStack, state)
                (Tt, Ff) ->
                    let newStack = Stack.push (pop (pop stack)) Ff
                    in Just (xs, newStack, state)
                (Ff, Tt) ->
                    let newStack = Stack.push (pop (pop stack)) Ff
                    in Just (xs, newStack, state)
                (Ff, Ff) ->
                    let newStack = Stack.push (pop (pop stack)) Ff
                    in Just (xs, newStack, state)
                _ -> error "Run-time error"
        -- Code responsible for applying the logical NOT to the top value from the stack, pushes the result on to the stack
        Neg ->
            case top stack of
                Tt ->
                    let newStack = Stack.push (pop stack) Ff
                    in Just (xs, newStack, state)
                Ff ->
                    let newStack = Stack.push (pop stack) Tt
                    in Just (xs, newStack, state)
                _ -> error "Run-time error"
```

```
        -- Code responsible for retrieving the value associated with a variable and pushes that value on to the stack.
        Fetch var ->
            case get state var of
                Just val ->
                    let newStack = Stack.push stack val
                    in Just (xs, newStack, state)
                Nothing ->
                    error "Run-time error"
        -- Code responsible for saving the value on top of the stack to a variable. Said value is popped from the stack.
        Store var ->
            let val = top stack
                newState = State.push state var val
            in Just (xs, pop stack, newState)
        -- Dummy function that returns the stack and state without any change.
        Noop ->
            Just (xs, stack, state)
        -- Branch function takes the top value form the stack, if it is True it runs code1 if it is False, runs code2 inste
        Branch code1 code2 ->
            case top stack of
                Tt -> Just (code1 ++ xs, pop stack, state)
                Ff -> Just (code2 ++ xs, pop stack, state)
                _ -> error "Run-time error"
        -- Loop function executed code1 while the value on top of the stack is True, when it is False executes code2 once a
        Loop code1 code2 ->
            let code = code1 ++ [Branch (code2 ++ [Loop code1 code2]) [Noop]] ++ xs
            in Just (code, stack, state)
```

During the development of this function we defined some auxiliary functions in the *State* and *Stack* files, those were pushing a variable-value pair into the state, getting the value of a variable from the state, pushing a value on to the stack, popping a value from the stack and returning the stack's top value, here are the respective codes:

```
-- Function to push a variable-value pair into the state.
push :: State -> Var -> Val -> State
push state var val = Map.insert var val state

-- Function to get the value of a variable from the state.
get :: State -> Var -> Maybe Val
get state var = Map.lookup var state

-- Function responsible for pushing a value on to the stack.
push :: Stack -> Val -> Stack
push (Stack stack) value = Stack (value : stack)

-- Function responsible for popping a value from the stack.
pop :: Stack -> Stack
pop (Stack (_:xs)) = Stack xs
pop _ = error "Run-time error"

-- Function responsible for returning the value on top of the stack.
top :: Stack -> Val
top (Stack (x:_)) = x
top _ = error "Run-time error"
```

Finally we declared the function to run the interpreter, again following the restrictions in the assignment paper. Here's the respective code:

```
-- Function to run the interpreter.
run :: (Code, Stack, State) -> (Code, Stack, State)
run (code, stack, state) =
    case exec (code, stack, state) of
        Just nextInterpreter -> run nextInterpreter
        Nothing -> (code, stack, state)
```

Part 2

Description

For the second half the challenge was to consider a small imperative programming language with arithmetic and boolean expressions, and statements consisting of assignments of the form x := a, sequence of statements (instr1 ; instr2), if then else statements, and while loops, and develop a compiler from this language into lists of instructions in the previous machine.

In order to complete this, we started by declaring three new data types, *Aexp* for arithmetic expression, *Bexp* for boolean expression and *Stm* for statements. We also defined an alias for a list of statements, called a Program. Here's the respective code:

```
-- Data type for arithmetic expressions.
data Aexp
    = A_var String          -- Variable
    | A_num Integer         -- Integer constant
    | A_add Aexp Aexp       -- Addition
    | A_sub Aexp Aexp       -- Subtraction
    | A_mul Aexp Aexp       -- Multiplication
    deriving (Eq, Show)

-- Data type for boolean expressions.
data Bexp
    = B_true                -- True
    | B_false               -- False
    | B_aeq Aexp Aexp       -- Arithmetic equality
    | B_beq Bexp Bexp       -- Boolean equality
    | B_leq Aexp Aexp       -- Less than or equal to
    | B_not Bexp            -- Logical NOT
    | B_and Bexp Bexp       -- Logical AND
    deriving (Eq, Show)
```

```
-- Data type for statements.
data Stm
    = S_assign String Aexp   -- Assignment statement
    | S_if Bexp [Stm] [Stm]  -- If statement with true and false branches
    | S_while Bexp [Stm]     -- While loop
    deriving (Eq, Show)

-- Alias definition for a program, which is a list of statements.
type Program = [Stm]
```

After we developed the compiler that transform a program from this imperative language into machine instructions for the machine in the first half. The syntax enforced in the assignment paper was followed and here's the resulting code included in the *compiler.hs* file:

```
-- Function to compile arithmetic expressions (Aexp) to instructions (Code).
compA :: Aexp -> Code
compA (A_var varName) = [Fetch varName]                          -- Fetch value from a variable
compA (A_num num) = [Push num]                                   -- Push an integer on to the stack
compA (A_add aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Add]     -- Addition
compA (A_sub aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Sub]     -- Subtraction
compA (A_mul aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Mult]    -- Multiplication

-- Function to compile boolean expression (Bexp) to instructions (Code).
compB :: Bexp -> Code
compB (B_true) = [Tru]                                           -- Push True on to the stack
compB (B_false) = [Fals]                                         -- Push False on to the stack
compB (B_aeq aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Equ]     -- Arithmetic equality
compB (B_beq bexp1 bexp2) = compB bexp2 ++ compB bexp1 ++ [Equ]     -- Boolean equality
compB (B_leq aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Le]      -- Less than or equal
compB (B_not bexp1) = compB bexp1 ++ [Neg]                       -- Boolean negation
compB (B_and bexp1 bexp2) = compB bexp1 ++ compB bexp2 ++ [And]     -- Boolean AND

-- Function to compile a program to instructions (Code).
compile :: Program -> Code
compile [] = []                                                                  -- Empty program returns empty list of inst
compile (S_assign var aexp : prog) = compA aexp ++ [Store var] ++ compile prog  -- Compile arithmetic expression
compile (S_if bexp stm1 stm2 : prog) =                                           -- Compile Branch with condtional execution
    let
        on_true = compile stm1
        on_false = compile stm2
        condition = compB bexp
    in
        condition ++ [Branch on_true on_false] ++ compile prog
compile (S_while bexp stm : prog) =                                              -- Compile Loop with condition execution
    let
        on_true = compile stm
        condition = compB bexp
    in
        [Loop on_true condition] ++ compile prog
```

Finally, we needed to develop a parser that transforms the program given as a string into Statements. For that, we developed a lexer that takes the given string and splits it into tokens facilitating the following parsing functions. We also developed functions that transform variables and integer to their associated tokens. Here's the code for the lexer included in the *lexer.hs* file:

```
-- Defining the 'Token' data type that corresponds that serve as the input to the parsing stage.
data Token
    = T_while                -- | while
    | T_do                   -- | do
    | T_if                   -- | if
    | T_then                 -- | then
    | T_else                 -- | else
    | T_not                  -- | not
    | T_plus                 -- | +
    | T_less                 -- | -
    | T_times                -- | *
    | T_semicolon            -- | ;
    | T_lbracket             -- | (
    | T_rbracket             -- | )
    | T_assign               -- | :=
    | T_leq                  -- | <=
    | T_aeq                  -- | ==
    | T_beq                  -- | =
    | T_and                  -- | and
    | T_integer Integer      -- | Number
    | T_bool Bool            -- | Boolean
    | T_var String           -- | Variable
    deriving (Eq, Show)


-- Function that transforms a string to a list of tokens.
lexer :: String -> [Token]
lexer [] = []
lexer ('w':'h':'i':'l':'e':restStr) = T_while : lexer restStr
lexer ('d':'o':restStr) = T_do : lexer restStr
lexer ('i':'f':restStr) = T_if : lexer restStr
lexer ('t':'h':'e':'n':restStr) = T_then : lexer restStr
lexer ('e':'l':'s':'e':restStr) = T_else : lexer restStr
lexer ('n':'o':'t':restStr) = T_not : lexer restStr
lexer ('+':restStr) = T_plus : lexer restStr
lexer ('-':restStr) = T_less : lexer restStr
lexer ('*':restStr) = T_times : lexer restStr
lexer (';':restStr) = T_semicolon : lexer restStr
lexer ('(':restStr) = T_lbracket : lexer restStr
lexer (')':restStr) = T_rbracket : lexer restStr
lexer (':':'=':restStr) = T_assign : lexer restStr
lexer ('<':'=':restStr) = T_leq : lexer restStr
```

```
lexer ('=':'=':restStr) = T_aeq : lexer restStr
lexer ('=':restStr) = T_beq : lexer restStr
lexer ('a':'n':'d':restStr) = T_and : lexer restStr
lexer ('T':'r':'u':'e':restStr) = T_bool True : lexer restStr
lexer ('F':'a':'l':'s':'e':restStr) = T_bool False : lexer restStr
lexer (chr:restStr)
    | isSpace chr = lexer restStr
    | isDigit chr = lexNum (chr:restStr)
    | isAlpha chr = lexVar (chr:restStr)
    | otherwise = error ("lexer: unexpected character " ++ [chr])

-- Function responsible for transforming a integer to its respective token.
lexNum :: String -> [Token]
lexNum str = T_integer (read numStr) : lexer restStr
    where (numStr, restStr) = span isDigit str

-- Function responsible for transforming a variable to its respective token.
lexVar :: String -> [Token]
lexVar str = T_var varStr : lexer restStr
    where (varStr, restStr) = span isAlpha str
```

In order to develop the parser we decided to divide it into two parsers, *parseA* responsible for parsing arithmetical expressions and taking into account the arithmetic precedences, and *parseB* responsible for parsing boolean expressions taking into account it's own precedences and it may call *parseA*, in case there are arithmetical expressions nested into the boolean expression. Both these are included in the *parser.hs* file.

Here's the code for *parseA*:

```
-- Function to parse an arithmetic expression (Aexp) from a list of tokens.
parseA :: [Token] -> Aexp
parseA tokens =
    case parseSumOrSubOrProdOrIntOrVarOrPar tokens of
        Just (expr, []) -> expr
        _ -> error "Run-time error"

-- Parsing integer constants, variables, or expressions within parentheses.
parseIntOrVarOrPar :: [Token] -> Maybe (Aexp, [Token])
parseIntOrVarOrPar (T_var var : restTokens)
    = Just (A_var var, restTokens)
parseIntOrVarOrPar (T_integer num : restTokens)
    = Just (A_num num, restTokens)
parseIntOrVarOrPar (T_lbracket : restTokens1)
    = case parseSumOrSubOrProdOrIntOrVarOrPar restTokens1 of
        Just (expr, (T_rbracket : restTokens2)) ->
            Just (expr, restTokens2)
        Just _ -> Nothing -- no closing paren
        Nothing -> Nothing
parseIntOrVarOrPar tokens = Nothing

-- Parsing multiplication, integer constants, variables, or expressions within parentheses.
parseProdOrIntOrVarOrPar :: [Token] -> Maybe (Aexp, [Token])
parseProdOrIntOrVarOrPar tokens
    = case parseIntOrVarOrPar tokens of
        Just (expr1, (T_times : restTokens1)) ->
            case parseProdOrIntOrVarOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (A_mul expr1 expr2, restTokens2)
                Nothing -> Nothing
        result -> result

-- Parsing addition, subtraction, multiplication, integer constants, variables, or expressions within parentheses.
parseSumOrSubOrProdOrIntOrVarOrPar::[Token] -> Maybe (Aexp, [Token])
parseSumOrSubOrProdOrIntOrVarOrPar tokens
    = case parseProdOrIntOrVarOrPar tokens of
        Just (expr1, (T_plus : restTokens1)) ->
            case parseSumOrSubOrProdOrIntOrVarOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (A_add expr1 expr2, restTokens2)
                Nothing -> Nothing
        Just (expr1, (T_less : restTokens1)) ->
            case parseSumOrSubOrProdOrIntOrVarOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (A_sub expr1 expr2, restTokens2)
                Nothing -> Nothing
        result -> result
```

And the code for *parseB*:

```
-- Function to parse a boolean expression (Bexp) from a list of tokens.
parseB :: [Token] -> Bexp
parseB tokens =
    case parseAndOrBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar tokens of
        Just (expr, []) -> expr
        _ -> error "Run-time error"

-- Parsing True, False or expressions withing parentheses.
parseTrueOrFalseOrPar :: [Token] -> Maybe (Bexp, [Token])
parseTrueOrFalseOrPar (T_bool True : restTokens)
    = Just (B_true, restTokens)
parseTrueOrFalseOrPar (T_bool False : restTokens)
    = Just (B_false, restTokens)
parseTrueOrFalseOrPar (T_lbracket : restTokens1)
    = case parseAndOrBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar restTokens1 of
        Just (expr, (T_rbracket : restTokens2)) ->
            Just (expr, restTokens2)
        Just _ -> Nothing -- no closing paren
        Nothing -> Nothing
```

```
parseTrueOrFalseOrPar tokens = Nothing

-- Parsing less than or equal (Leq) expressions, True, False, or expressions within parentheses.
parseLeqOrTrueOrFalseOrPar :: [Token] -> Maybe (Bexp, [Token])
parseLeqOrTrueOrFalseOrPar tokens
    = case parseSumOrSubOrProdOrIntOrVarOrPar tokens of
        Just (expr1, (T_leq : restTokens1)) ->
            case parseSumOrSubOrProdOrIntOrVarOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (B_leq expr1 expr2, restTokens2)
                Nothing -> Nothing
        _ -> parseTrueOrFalseOrPar tokens

-- Parsing arithmetic equality (Aeq), less than or equal (Leq), True, False, or expressions within parentheses.
parseAeqOrLeqOrTrueOrFalseOrPar :: [Token] -> Maybe (Bexp, [Token])
parseAeqOrLeqOrTrueOrFalseOrPar tokens
    = case parseSumOrSubOrProdOrIntOrVarOrPar tokens of
        Just (expr1, (T_aeq : restTokens1)) ->
            case parseSumOrSubOrProdOrIntOrVarOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (B_aeq expr1 expr2, restTokens2)
                Nothing -> Nothing
        _ -> parseLeqOrTrueOrFalseOrPar tokens

-- Parsing boolean negation (Not), arithmetic equality (Aeq), less than or equal (Leq), True, False, or expressions within
parseNotOrAeqOrLeqOrTrueOrFalseOrPar :: [Token] -> Maybe (Bexp, [Token])
parseNotOrAeqOrLeqOrTrueOrFalseOrPar (T_not : restTokens)
    = case parseAeqOrLeqOrTrueOrFalseOrPar restTokens of
        Just (expr1, restTokens1) ->
            Just (B_not expr1, restTokens1)
        result -> parseAeqOrLeqOrTrueOrFalseOrPar restTokens
parseNotOrAeqOrLeqOrTrueOrFalseOrPar tokens = parseAeqOrLeqOrTrueOrFalseOrPar tokens

-- Parsing boolean equality (Beq), boolean negation (Not), arithmetic equality (Aeq),
-- less than or equal (Leq), True, False, or expressions within parentheses.
parseBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar :: [Token] -> Maybe (Bexp, [Token])
parseBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar tokens
    = case parseNotOrAeqOrLeqOrTrueOrFalseOrPar tokens of
        Just (expr1, (T_beq : restTokens1)) ->
            case parseBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (B_beq expr1 expr2, restTokens2)
                Nothing -> Nothing
        result -> result

-- Parsing logical AND (And), boolean equality (Beq), boolean negation (Not), arithmetic equality (Aeq),
-- less than or equal (Leq), True, False, or expressions within parentheses.
parseAndOrBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar :: [Token] -> Maybe (Bexp, [Token])
parseAndOrBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar tokens
    = case parseBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar tokens of
        Just (expr1, (T_and : restTokens1)) ->
            case parseAndOrBeqOrNotOrAeqOrLeqOrTrueOrFalseOrPar restTokens1 of
                Just (expr2, restTokens2) ->
                    Just (B_and expr1 expr2, restTokens2)
                Nothing -> Nothing
        result -> result
```

After we created two helper functions for parsing a body that appears enclosed in parentheses, the *parseBody* and *parseBodyAux*, here are the corresponding codes :

```
-- Parse the body of a block, enclosed in parentheses.
parseBody :: [Token] -> ([Token], [Token])
parseBody (T_lbracket : tokens) = parseBodyAux tokens [] [T_lbracket]
parseBody tokens = (takeWhile (\t -> t /= T_semicolon && t /= T_then && t /= T_else) tokens,
                    drop 1 (dropWhile (\t -> t /= T_semicolon && t /= T_then && t /= T_else) tokens))

-- Auxiliary function to help parse the body of a block
parseBodyAux :: [Token] -> [Token] -> [Token] -> ([Token], [Token])
parseBodyAux [] body stack = error "Run-time error"                                         -- Unmatched brackets.
parseBodyAux (T_lbracket : tokens) body stack = parseBodyAux tokens (body ++ [T_lbracket]) (T_lbracket : stack)
parseBodyAux (T_rbracket : tokens) body (T_lbracket : stack) =
    if stack == [] then (body, tokens)                                                      -- Successfully matched brackets
    else parseBodyAux tokens (body ++ [T_rbracket]) stack
parseBodyAux (token : tokens) body stack = parseBodyAux tokens (body ++ [token]) stack
```

And then finally we developed the main parser function that includes the code to parse if->then / else and while loops as well as the callers for all the functions developed previously, here's the respective code:

```
-- Function to parse a program from a string input.
parse :: String -> Program
parse input = parseStatements (lexer input)

-- Parse a list of tokens into a list of statements.
parseStatements :: [Token] -> [Stm]
parseStatements [] = []
parseStatements (T_var var : T_assign : tokens) =
    let
        aexp = parseA (takeWhile (/= T_semicolon) tokens)  -- Parse arithmetic expression.
    in
        S_assign var aexp : parseStatements (drop 1 (dropWhile (/= T_semicolon) tokens))

-- Parsing of a while loop
parseStatements (T_while : tokens) =
    let
        bexp = parseB (takeWhile (/= T_do) tokens)                          -- Parse boolean expression.
        (body, restTokens) = parseBody (drop 1 (dropWhile (/= T_do) tokens))  -- Parse while loop body.
```

```
    in
        S_while bexp (parseStatements body) : parseStatements (drop 1 (dropWhile (/= T_semicolon) restTokens))

-- Parsing of a if->then / else
parseStatements (T_if : tokens) =
    let
        bexp = parseB (takeWhile (/= T_then) tokens)                                        -- Parse boolean expression.
        (trueBranch, restTokens1) = parseBody (drop 1 (dropWhile (/= T_then) tokens))       -- Parse true branch of if stat
        (falseBranch, restTokens2) = parseBody (drop 1 (dropWhile (/= T_else) restTokens1)) -- Parse false branch of if sta
    in
        S_if bexp (parseStatements trueBranch) (parseStatements falseBranch) : parseStatements restTokens2

parseStatements (T_semicolon : tokens) = parseStatements tokens
```

In this code, the Program variable relates to a list of Statements.

To test all the code developed we ran several tests, for the Assembler and the Parser, all of the results were positive assuring us we developed the code and solved the problems correctly.

The tests we ran:

```
testAssembler [Push 10,Push 4,Push 3,Sub,Mult] == ("-10","")
testAssembler [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"] == ("","a=3,someVar=False,var=True")
testAssembler [Fals,Store "var",Fetch "var"] == ("False","var=False")
testAssembler [Push (-20),Tru,Fals] == ("False,True,-20","")
testAssembler [Push (-20),Tru,Tru,Neg] == ("False,True,-20","")
testAssembler [Push (-20),Tru,Tru,Neg,Equ] == ("False,-20","")
testAssembler [Push (-20),Push (-21), Le] == ("True","")
testAssembler [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"] == ("","x=4")
testAssembler [Push 10,Store "i",Push 1,Store "fact",Loop [Push 1,Fetch "i",Equ,Neg] [Fetch "i",Fetch "fact",Mult,Store "fa
testParser "x := 5; x := x - 1;" == ("","x=4")
testParser "x := 0 - 2;" == ("","x=-2")
testParser "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;" == ("","y=2")
testParser "x := 42; if x <= 43 then x := 1; else (x := 33; x := x+1;);" == ("","x=1")
testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1;" == ("","x=2")
testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1; z := x+x;" == ("","x=2,z=4")
testParser "x := 44; if x <= 43 then x := 1; else (x := 33; x := x+1;); y := x*2;" == ("","x=34,y=68")
testParser "x := 42; if x <= 43 then (x := 33; x := x+1;) else x := 1;" == ("","x=34")
testParser "if (1 == 0+1 = 2+1 == 3) then x := 1; else x := 2;" == ("","x=1")
testParser "if (1 == 0+1 = (2+1 == 4)) then x := 1; else x := 2;" == ("","x=2")
testParser "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);" == ("","x=2,y=-10,z=6")
testParser "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i - 1;);" == ("","fact=3628800,i=1")
```

And the results we got:

```
ghci> :l Main.hs
[1 of 9] Compiling Inst             ( Inst.hs, interpreted )
[2 of 9] Compiling Lexer            ( Lexer.hs, interpreted )
[3 of 9] Compiling Parser           ( Parser.hs, interpreted )
[4 of 9] Compiling Compiler         ( Compiler.hs, interpreted )
[5 of 9] Compiling Value            ( Value.hs, interpreted )
[6 of 9] Compiling State            ( State.hs, interpreted )
[7 of 9] Compiling Stack            ( Stack.hs, interpreted )
[8 of 9] Compiling Interpreter      ( Interpreter.hs, interpreted )
[9 of 9] Compiling Main             ( Main.hs, interpreted )
Ok, 9 modules loaded.
ghci> main
Assembler test 1: True
Assembler test 2: True
Assembler test 3: True
Assembler test 4: True
Assembler test 5: True
Assembler test 6: True
Assembler test 7: True
Assembler test 8: True
Assembler test 9: True
Parser test 1: True
Parser test 2: True
Parser test 3: True
Parser test 4: True
Parser test 5: True
Parser test 6: True
Parser test 7: True
Parser test 8: True
Parser test 9: True
Parser test 10: True
Parser test 11: True
Parser test 12: True
```

Conclusion

In conclusion, after extensive testing, we are confident in the accurate development of the Interpreter, Compiler, and Parser for the targeted machine.

The primary challenges encountered were rooted in initially navigating a programming language that was unfamiliar to us and different that all the others we studied. Furthermore, while constructing the parser, we struggled with the implementation of if-then/else statements and while loops, having to take into consideration the precendences in both the boolean and arithmetic expressions and the fact that some boolean expressions may have arithmetical expressions included in them.

In the end, we managed to solve all the problems and we feel this helped a lot in understanding some concepts in Haskell as well as consolidating the knowledge acquired in the theoretical and practical classes.