Haskell Compiler

Group Members

Nome	Up	Contribuição
Inês Alexandre Queirós Matos Macedo de Oliveira	202103343	50%
João Pedro Cruz Moreira de Oliveira	202108737	50%

Installation and Execution

To be able to run the preject, you need to install GHCi (version 9.0 or later) and download the source code.

To be able to execute, you just need to type `:load Main.hs` and then `main`. Additional tests for execution can be added in the "Testers" section.

Stack and State

The first task of our program was to decide what types the stack and state should support. Since the stack can support either integers for arithmetic calculations and also boolean values, we have decided to use stack type as follows:

```
type StackType = Either Integer String
type Stack = [StackType]
```

String was the choice for us instead of Bool since **tru** and **fals** instructions push the constants tt and ff, respectively, into the stack. When deciding the type of our state, we considered the rules present in **fetch-x** and **store-x** instructions. Since they can push and pop values from the stack, respectively, we have decided to use state type as follows:

```
type State = [(String, StackType)]
```

Machine Instructions

The first part of the project consists of operations of a low-level machine defined by configurations (c, e, s), where 'c' is the code to be executed, 'e' the evaluation stack and 's' the storage. We use the evaluation stack to evaluate arithmetic (composed of integer numbers only, which can positive or negative) and boolean expressions. Focused on processing arithmetic and Boolean expressions, the machine uses a set of basic operations such as **add**, **mult** and **sub** for arithmetic calculations and **eq** and **le** for Boolean evaluations. These operations, together with stack manipulation instructions such as **push-n**, **fetch-x**, **store-x**, **branch(c1, c2)**, **loop(c1, c2)**, **tru**, **fals** and **noop** form the core of the machine's functionality. And finally, there are two intructions mentioned before responsable for changing the flow of controll, **branch(c1, c2)** and **loop(c1, c2)**.

Add

The add function takes the two top integers from the stack, adds them, and pushes their sum back into the stack. This operation affects only the top elements of the stack, leaving the rest of the stack and the machine's state as it is.

```
add :: (Code, Stack, State) -> (Code, Stack, State)
add (Add:tailcode, (Left i1):(Left i2):tailstack, state) = (tailcode, Left (i1 +
i2):tailstack, state)
```

Mult

The mult function multiplies the two topmost integers on the stack and places the product back on the stack. This operation affects only the top elements of the stack, leaving the rest of the stack and the machine's state as it is.

```
mult :: (Code, Stack, State) -> (Code, Stack, State)
mult (Mult:tailcode, (Left i1):(Left i2):tailstack, state) = (tailcode, Left (i1 *
i2):tailstack, state)
```

Sub

The sub function subtracts the topmost element of the stack with the second topmost element and pushes the result back into the stack. This operation affects only the top elements of the stack, leaving the rest of the stack and the machine's state as it is.

```
sub :: (Code, Stack, State) -> (Code, Stack, State)
sub (Sub:tailcode, (Left i1):(Left i2):tailstack, state) = (tailcode, Left (i1 -
i2):tailstack, state)
```

Equ

The equ function compares either two integers or two boolean values from the top of the stack. If the values are equal, it pushes tt (string value that binds to true) into the stack; otherwise, it pushes ff (string value that binds to false). This operation updates the stack with the comparison result, while the rest of the machine's state remains unaffected.

```
equ :: (Code, Stack, State) -> (Code, Stack, State)
equ (Equ:tailcode, (Left i1):(Left i2):tailstack, state) = (tailcode, Right(
boolToStr(i1 == i2)):tailstack, state)
equ (Equ:tailcode, (Right i1):(Right i2):tailstack, state) = (tailcode, Right (
boolToStr( strToBool i1 == strToBool i2)):tailstack, state)
```

Le

The le function evaluate the less-than-or-equal-to (<=) condition between two integers on the stack. It checks if the topmost stack element is less or equal to the second topmost element. If true, it pushes tt (string value that binds to true) into the stack; otherwise, ff (string value that binds to false) is pushed. This operation updates the stack with the comparison result, while the rest of the machine's state remains unaffected.

```haskell
le :: (Code, Stack, State) -> (Code, Stack, State)
le (Le:tailcode, (Left i1):(Left i2):tailstack, state) = (tailcode, Right(
boolToStr(i1 <= i2)):tailstack, state)
```

Tru

The true function simply pushes the boolean value tt (string value that binds to true) into the top of the stack. This function doesn't alter the remaining stack or the machine's state, but adds a true boolean value to be used in subsequent operations or comparisons.

```haskell
true :: (Code, Stack, State) -> (Code, Stack, State)
true (Tru:tailcode, stack, state) = (tailcode, Right "tt":stack, state)
```

Fals

The false function pushes the boolean value ff (string value that binds to false) into the top of the stack. Like the true function, it doesn't modify the rest of the stack or the state of the machine, but simply adds a false boolean value for use in later computations or logical evaluations.

```haskell
false :: (Code, Stack, State) -> (Code, Stack, State)
false (Fals:tailcode, stack, state) = (tailcode, Right "ff":stack, state)
```

Push

The push function add a new integer value to the stack. When it encounters the Push-i instruction, it places the specified integer i into the top of the stack. This function only alters the stack by adding the new element, leaving the remaining stack and the machine's state unchanged. Its primary role is to introduce new data into the machine's computations.

```haskell
push :: (Code, Stack, State) -> (Code, Stack, State)
push (Push i:tailcode, stack, state) = (tailcode, Left i:stack, state)
```

Store

The store function in the machine's instruction operates by taking the top element from the stack and associating it with a variable in the machine's state. When the Store var instruction is executed, the function removes the topmost element el from the stack and updates the machine's state by binding this element to the variable var. This operation modifies the state by storing a new value, but it doesn't change the subsequent code execution flow. The function ensures data persistence by updating the state and is essential for maintaining and manipulating stored values within the machine's operations.

```haskell
store :: (Code, Stack, State) -> (Code, Stack, State)
store ((Store var):tailcode, el:tailstack, state) = (tailcode, tailstack,
updateState var el state)
```

```haskell
updateState :: String -> StackType -> State -> State
updateState var el state = (var, el) : filter ((/= var) . fst) state
```

Fetch

The fetch function is an essential part of the machine's instruction set, focusing on retrieving stored data. When the Fetch var instruction is executed, this function looks up the value associated with the variable var in the machine's state. It then places this value on top of the stack, without modifying the rest of the stack or the machine's state. This operation is vital for accessing and utilizing previously stored values within the machine's computational processes, enabling dynamic data retrieval and use in ongoing operations.

```haskell
fetch :: (Code, Stack, State) -> (Code, Stack, State)
fetch ((Fetch var):tailcode, stack, state) = (tailcode, searchVar var state :
stack, state)


searchVar :: String -> State -> StackType
searchVar var state = case lookup var state of
    Just value -> value
    Nothing    -> error "Run-time error"
```

Noop

The noop function in the machine's instruction set is a "no operation" command. It leaves the stack and the state unchanged, effectively doing nothing but moving to the next instruction in the code sequence.

```haskell
noop :: (Code, Stack, State) -> (Code, Stack, State)
noop (Noop:tailcode, stack, state) = (tailcode, stack, state)
```

Branch

The branch function is a control flow operation in the machine's instruction set. It decides the next set of instructions to execute based on the top value of the stack. If the top value is tt (true), the function continues with code1 followed by the remaining code (tailcode). Conversely, if the top value is ff (false), it proceeds with code2 and then tailcode. In both cases, the top boolean value is removed from the stack, and the machine's state remains unchanged.

```
branch :: (Code, Stack, State) -> (Code, Stack, State)
branch ((Branch code1 cod2):tailcode, (Right "tt"):tailstack, state) = (code1 ++
tailcode, tailstack, state)
branch ((Branch code1 code2):tailcode, (Right "ff"):tailstack, state) = (code2 ++
tailcode, tailstack, state)
```

Loop

The loop function enables looping in the machine's operations. It takes two sets of instructions, code1 and code2. The machine first executes code1, and then code2 determines whether to repeat code1 or move on. This setup allows for continuous execution of a set of instructions (code1) based on a condition defined in code2. The stack and the state of the machine are unaffected during this initial setup of the loop.

For building our loop instruction we followed the hint: `"For example, loop(c1, c2) may be transformed into c1 ++ [branch([c2, loop(c1, c2)], [noop])]".`

```
loop :: (Code, Stack, State) -> (Code, Stack, State)
loop ((Loop code1 code2):tailcode, stack, state) = (code1 ++ [Branch (code2 ++
[Loop code1 code2]) [Noop]] ++ tailcode, stack, state)
```

Neg

The neg function is used for negating a boolean value. When it encounters the Neg instruction, it takes the top boolean value on the stack, inverts it (true becomes false, and vice versa), and then places this negated value back on the stack. This operation only affects the top element of the stack and leaves the rest of the stack and the machine's state as they were.

```
neg :: (Code, Stack, State) -> (Code, Stack, State)
neg (Neg:tailcode, (Right boolval):tailstack, state) = (tailcode, Right (boolToStr
(not (strToBool boolval))):tailstack, state)
```

AndF

The andF function performs a logical 'AND' operation. It operates on the top two boolean values on the stack. When the And instruction is executed, it evaluates the logical 'AND' of these two values (both true results in true, otherwise false) and pushes this result back into the stack. This operation modifies only the top two elements of the stack, replacing them with the outcome of the 'AND' operation, while the rest of the stack and the machine's state remain unchanged.

```
andF :: (Code, Stack, State) -> (Code, Stack, State)
andF (And:tailcode, (Right boolval1):(Right boolval2):tailstack, state) =
(tailcode, Right (boolToStr (strToBool boolval1 && strToBool boolval2)):tailstack,
state)
```

Error handling

All operations feature **error handling**, ensuring that each operation validates its data before proceeding. For example, in the add function, the operation only proceeds if the first two elements of the stack are integers (Left i1 and Left i2). If they are not, or if the stack is not correctly formed, an error is displayed. This strict input validation pattern protects against runtime errors that can arise from invalid or unexpected states, such as incompatible types, incorrect stack formats or empty variables.

For displaying the error in machine instructions, we used the following code:

```
error "Run-time error"
```

Machine Top-Level Functions

For this project, we developed 5 Top-Level functions such as **createEmptyStack**, **createEmptyState**, **stack2St**, **state2Str** and **run**.

For creating an empty stack and state before starting any program we used the following code snippets:

```
createEmptyStack :: Stack
createEmptyStack = []

createEmptyState :: State
createEmptyState = []
```

Since we needed to transform a given stack into a string, we used the following code:

Note: The string represents the stack as an ordered list of values, separated by commas and without spaces, with the leftmost value representing the top of the stack.

```
stack2Str :: Stack -> String
stack2Str stack = intercalate "," $ map stackElementToStr stack

stackElementToStr :: StackType -> String
stackElementToStr (Left integer) = show integer
stackElementToStr (Right string)
    | string == "tt" = "True"
    | string == "ff" = "False"
    | otherwise = error "Run-time error"
```

Since we needed to transform a given state into a string, we used the following code:

Note: The string represents the state as an list of pairs variable-value, separated by commas and without spaces, with the pairs ordered in alphabetical order of the variable name. Each variable-value pair is represented without spaces and using an "=".

```
state2Str :: State -> String
state2Str state = intercalate "," $ map statePairToStr $ sortOn fst state

statePairToStr :: (String, StackType) -> String
statePairToStr (var, value) = var ++ "=" ++ stackElementToStr value
```

Finally, to run our program we created the **run** function that receives a list of instructions, a stack that is initially empty and a state. Runs the list of instructions returning as ouput an empty code list, a stack and the output values in the state.

For that, we used the following code:

```
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack, state) = ([], stack, state)
run (Add:tailcode, stack, state) = run (add (Add:tailcode, stack, state))
run (Mult:tailcode, stack, state) = run (mult (Mult:tailcode, stack, state))
run (Sub:tailcode, stack, state) = run (sub (Sub:tailcode, stack, state))
run (Equ:tailcode, stack, state) = run (equ (Equ:tailcode, stack, state))
run (Le:tailcode, stack, state) = run (le (Le:tailcode, stack, state))
run (Tru:tailcode, stack, state) = run (true (Tru:tailcode, stack, state))
run (Fals:tailcode, stack, state) = run (false (Fals:tailcode, stack, state))
run ((Push i):tailcode, stack, state) = run (push ((Push i):tailcode, stack,
state))
run ((Store var):tailcode, stack, state) = run (store ((Store var):tailcode,
stack, state))
run ((Fetch var):tailcode, stack, state) = run (fetch ((Fetch var):tailcode,
stack, state))
run (Noop:tailcode, stack, state) = run (noop (Noop:tailcode, stack, state))
run ((Branch code1 code2):tailcode, stack, state) = run (branch ((Branch code1
code2):tailcode, stack, state))
run ((Loop code1 code2):tailcode, stack, state) = run (loop ((Loop code1
code2):tailcode, stack, state))
run (Neg:tailcode, stack, state) = run (neg (Neg:tailcode, stack, state))
run (And:tailcode, stack, state) = run (andF (And:tailcode, stack, state))
```

Compiler

The Compiler module's core function is to translate high-level programming constructs into machine code, a process known as compilation. For this compiler, we 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.

For this compiler we have the following rules:

- Arithmetic and boolean expressions will be evaluated on the evaluation stack of the machine and the code to be generated must effect this. Thus, the code generated for binary expressions consists of the code for the right argument followed by that for the left argument and, finally, the appropriate instruction for the operator. In this way, it is ensured that the arguments appear on the evaluation stack in the order required by the instructions.
- The code generated for x := a appends the code for the arithmetic expression a with the instruction store−x. This instruction assigns x the appropriate value and additionally pops the stack.
- For a sequence of two statements, we just concatenate the two instruction sequences.
- When generating code for the conditional, the code for the boolean expression will ensure that a truth value will be placed on top of the evaluation stack, and the branch instruction will then inspect (and pop) that value and select the appropriate piece of code.
- Finally, the code for the while statement uses the loop-instruction.

Our first step was to define three data types to represent expressions and statements:

- Aexp for arithmetic expressions
- Bexp for boolean expressions
- Stm for statements

For that, we used the following code:

```haskell
data Aexp = Number Integer | Variable String | AAdd Aexp Aexp | ASub Aexp Aexp |
AMul Aexp Aexp
   deriving (Show, Eq)

data Bexp = TrueExp | FalseExp | BEqu Aexp Aexp | BLe Aexp Aexp | BAnd Bexp Bexp |
BNeg Bexp | BEquality Bexp Bexp
    deriving (Show, Eq)

data Stm = Assign String Aexp | Seq [Stm] | IfThenElse Bexp Stm Stm | While Bexp
Stm | Skip
  deriving (Show, Eq)

type Program = [Stm]
```

Finally, we defined a compiler to parse a list of instructions in this imperative language into machine instructions as we mentioned before.

For that, we used three functions:

- compile :: Program → Code, which is the main compiler
- compA :: Aexp → Code, which compile arithmetic
- compB :: Bexp → Code, which boolean expressions

For this, we used the following code:

```haskell
compile :: Program -> Code
compile [] = []
compile (s:stms) = compStm s ++ compile stms

compStm :: Stm -> Code
compStm (Assign str aexp) = compA aexp ++ [Store str]
compStm (Seq stms) = concatMap compStm stms
compStm (IfThenElse bexp stm1 stm2) = compB bexp ++ [Branch (compStm stm1)
(compStm stm2)]
compStm (While bexp stm) = [Loop (compB bexp) (compStm stm)]
compStm Skip = [Noop]

compA :: Aexp -> Code
compA (Number n) = [Push n]
compA (Variable str) = [Fetch str]
compA (AAdd aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Add]
compA (ASub aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Sub]
compA (AMul aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Mult]

compB :: Bexp -> Code
compB TrueExp = [Tru]
compB FalseExp = [Fals]
compB (BEqu aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Equ]
compB (BEquality bexp1 bexp2) = compB bexp2 ++ compB bexp1 ++ [Equ]
compB (BLe aexp1 aexp2) = compA aexp2 ++ compA aexp1 ++ [Le]
compB (BAnd bexp1 bexp2) = compB bexp2 ++ compB bexp1 ++ [And]
compB (BNeg bexp) = compB bexp ++ [Neg]
```

Parser

Having the compiler and machine instructions, we developed a parser responsable for transforming an imperative program represented as a string into its corresponding representation in the Stm data (a list of statements Stm).

For this Parser, we have the following rules:

- All statements end with a semicolon (**;**).
- The string may contain spaces between each token.
- Variables begin with a lowercase letter and can´t contain a reserved keyword as a substring.
- Operator precedence in arithmetic expressions is the usual.
- Parentheses may be used to add priority to an operation.
- • In boolean expressions, two instances of the same operator are also computed from left to right. The order of precedence for the operations is (with the first one being executed first): integer inequality (≤), integer equality (==), logical negation (not), boolean equality (=) and logical conjunction (and).

For this parser, we decided to use a lexer that splits the string into a list of words, tokens. For that, we used the following list of tokens:

```haskell
data Token = PlusToken | MinusToken | MultToken | OpenPToken | ClosePToken |
IntToken Integer | VarToken String | SemiColonToken | AssignToken | BLeToken |
BEqAritToken | BEqBoolToken | IfToken | ThenToken | ElseToken | TrueToken |
FalseToken | NotToken | AndToken | WhileToken | DoToken
  deriving (Show, Eq)
```

For this parse program we have the following structure:

- **parse :: String -> Program**, which receives a string and returns a list of stms (Program) to be compiled.
- **lexer :: String -> [Token]**, which receives a string and returns a list of tokens.
- **parseTokens :: [Token] -> Program**, which receives a list of tokens and returns a list of stms (Program).
- **parseStm :: [Token] -> (Stm, [Token])**, which receives a list of tokens and is responsible to parse each stm.
- **parseStms :: [Token] -> ([Stm], [Token])**, which receives a list of tokens and is responsible to parse multiple stms. For example, used in nested stms in if-then-else and while loops.
- **parseIfThenElse :: [Token] -> Maybe (Stm, [Token])**, which receives a list of tokens and is responsible to parse an if-then-else stm.
- **parseWhileLoop :: [Token] -> Maybe (Stm, [Token])**, which receives a list of tokens and is responsible to parse a while loop.
- **parseParentSumsOrMultOrBasicExpr :: [Token] -> Maybe (Aexp, [Token])**, which receives a list of tokens and is responsible to parse sums or products or parenthesised expressions.
- **parseParenMulOrBasicExpr :: [Token] -> Maybe (Aexp, [Token])**, which receives a list of tokens and is responsible to parse products or parenthesis expressions.
- **parseParenOrBasicExpr :: [Token] -> Maybe (Aexp, [Token])**, which receives a list of tokens and is responsible to parse parenthesised expressions.
- **parseBexp :: [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse boolean expressions.
- **parseNotOrBoolEqualityOrAnd :: [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse not or boolean equality or and tokens.
- **parseNotOrBoolEquality:: [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse not or boolean equality tokens.
- **parseNot:: [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse not tokens.
- **parseArithmeticComparison :: [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse arithmetic comparison.
- **parseIntegerEquality :: Aexp -> [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse integer equality.
- **parseIntegerLessThanOrEqual :: Aexp -> [Token] -> Maybe (Bexp, [Token])**, which receives a list of tokens and is responsible to parse integer less than or equal.
- **printToken:: String -> IO ()**, which is just an helper function to verify which tokens we get from the lexer.

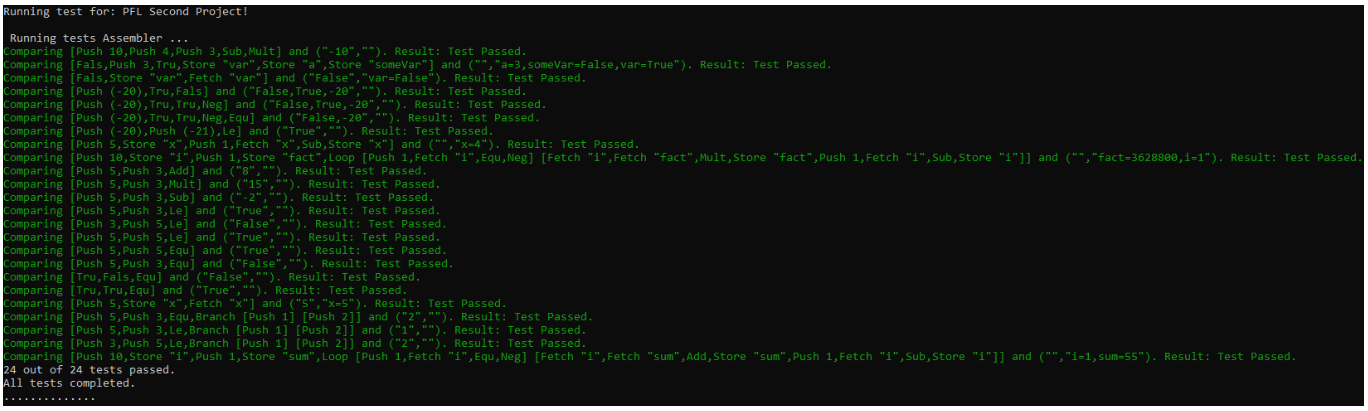
Tester

To validate our application, we created the following code:

```
testAssembler :: Code -> (String, String) -- Tests the assembler
AtestParser :: String -> (String, String) -- Tests the parser
testAssemblerCases :: [([Inst], (String, String))] -- Test cases for the assembler
```

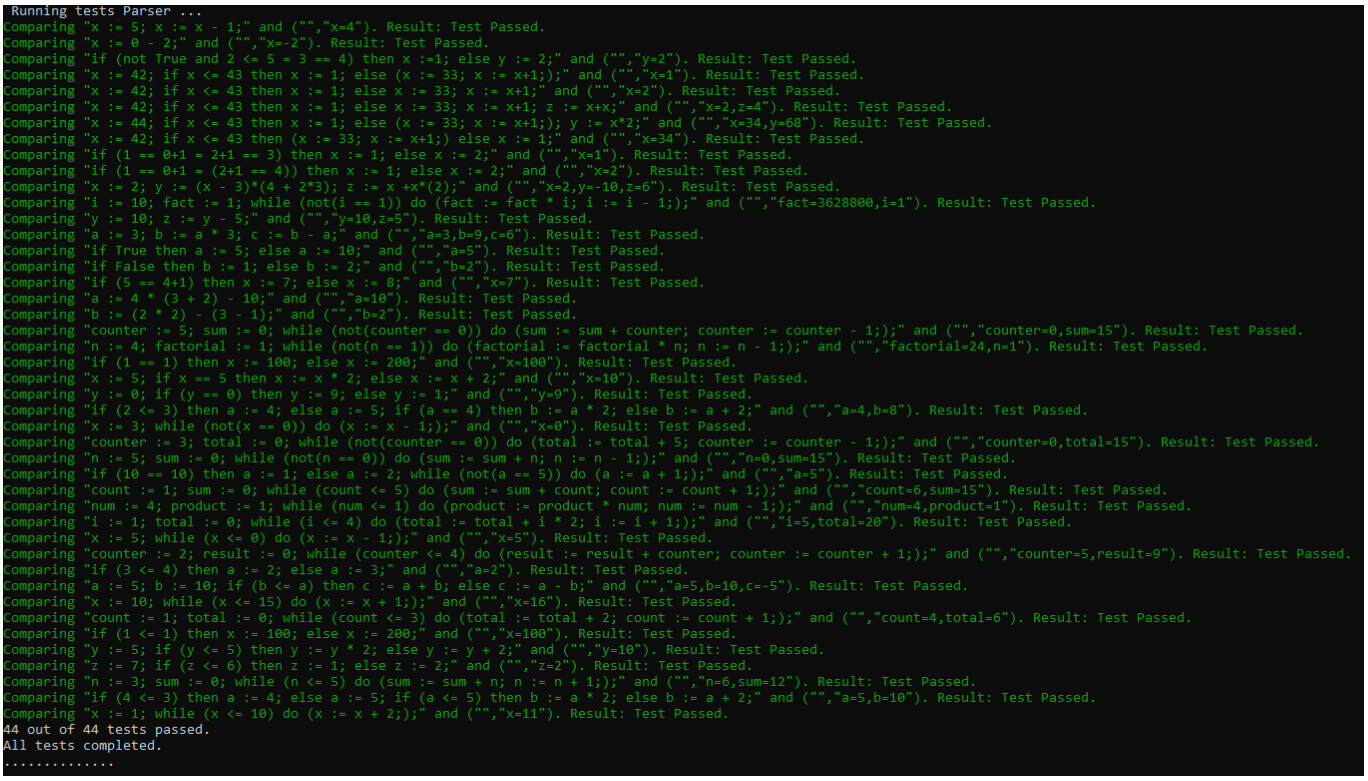
```haskell
testParserCases :: [(String, (String, String))] -- Test cases for the parser
runTests :: IO () -- Runs all the tests
runTestAssembler :: ([Inst], (String, String)) -> IO Bool -- Runs assembler tests
runTestParser :: (String, (String, String)) -> IO Bool -- Runs parser tests
```

And we tested the scenarios from the template with additional personal cases:



```
Running test for: PFL Second Project!

 Running tests Assembler ...
Comparing [Push 10,Push 4,Push 3,Sub,Mult] and ("-10",""). Result: Test Passed.
Comparing [Fals,Push 3,Tru,Store "var",Store "a",Store "someVar"] and ("","a=3,someVar=False,var=True"). Result: Test Passed.
Comparing [Fals,Store "var",Fetch "var"] and ("False","var=False"). Result: Test Passed.
Comparing [Push (-20),Tru,Fals] and ("False,True,-20",""). Result: Test Passed.
Comparing [Push (-20),Tru,Tru,Neg] and ("False,True,-20",""). Result: Test Passed.
Comparing [Push (-20),Tru,Tru,Neg,Equ] and ("False,-20",""). Result: Test Passed.
Comparing [Push (-20),Push (-21),Le] and ("True",""). Result: Test Passed.
Comparing [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"] and ("","x=4"). Result: Test Passed.
Comparing [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"]] and ("","fact=3628800,i=1"). Result: Test Passed.
Comparing [Push 5,Push 3,Add] and ("8",""). Result: Test Passed.
Comparing [Push 5,Push 3,Mult] and ("15",""). Result: Test Passed.
Comparing [Push 5,Push 3,Sub] and ("-2",""). Result: Test Passed.
Comparing [Push 5,Push 3,Le] and ("True",""). Result: Test Passed.
Comparing [Push 3,Push 5,Le] and ("False",""). Result: Test Passed.
Comparing [Push 5,Push 5,Le] and ("True",""). Result: Test Passed.
Comparing [Push 5,Push 5,Equ] and ("True",""). Result: Test Passed.
Comparing [Push 5,Push 3,Equ] and ("False",""). Result: Test Passed.
Comparing [Tru,Fals,Equ] and ("False",""). Result: Test Passed.
Comparing [Tru,Tru,Equ] and ("True",""). Result: Test Passed.
Comparing [Push 5,Store "x",Fetch "x"] and ("5","x=5"). Result: Test Passed.
Comparing [Push 5,Push 3,Equ,Branch [Push 1] [Push 2]] and ("2",""). Result: Test Passed.
Comparing [Push 5,Push 3,Le,Branch [Push 1] [Push 2]] and ("1",""). Result: Test Passed.
Comparing [Push 3,Push 5,Le,Branch [Push 1] [Push 2]] and ("2",""). Result: Test Passed.
Comparing [Push 10,Store "i",Push 1,Store "sum",Loop [Push 1,Fetch "i",Equ,Neg] [Fetch "i",Fetch "sum",Add,Store "sum",Push 1,Fetch "i",Sub,Store "i"]] and ("","i=1,sum=55"). Result: Test Passed.
24 out of 24 tests passed.
All tests completed.
..............
```



```
 Running tests Parser ...
Comparing "x := 5; x := x - 1;" and ("","x=4"). Result: Test Passed.
Comparing "x := 0 - 2;" and ("","x=-2"). Result: Test Passed.
Comparing "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;" and ("","y=2"). Result: Test Passed.
Comparing "x := 42; if x <= 43 then x := 1; else (x := 33; x := x+1;);" and ("","x=1"). Result: Test Passed.
Comparing "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1;" and ("","x=2"). Result: Test Passed.
Comparing "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1; z := x+x;" and ("","x=2,z=4"). Result: Test Passed.
Comparing "x := 44; if x <= 43 then x := 1; else (x := 33; x := x+1;); y := x*2;" and ("","x=34,y=68"). Result: Test Passed.
Comparing "x := 42; if x <= 43 then (x := 33; x := x+1;) else x := 1;" and ("","x=34"). Result: Test Passed.
Comparing "if (1 == 0+1 = 2+1 == 3) then x := 1; else x := 2;" and ("","x=1"). Result: Test Passed.
Comparing "if (1 == 0+1 = (2+1 == 4)) then x := 1; else x := 2;" and ("","x=2"). Result: Test Passed.
Comparing "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);" and ("","x=2,y=-10,z=6"). Result: Test Passed.
Comparing "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i - 1;);" and ("","fact=3628800,i=1"). Result: Test Passed.
Comparing "y := 10; z := y - 5;" and ("","y=10,z=5"). Result: Test Passed.
Comparing "a := 3; b := a * 3; c := b - a;" and ("","a=3,b=9,c=6"). Result: Test Passed.
Comparing "if True then a := 5; else a := 10;" and ("","a=5"). Result: Test Passed.
Comparing "if False then b := 1; else b := 2;" and ("","b=2"). Result: Test Passed.
Comparing "if (5 == 4+1) then x := 7; else x := 8;" and ("","x=7"). Result: Test Passed.
Comparing "a := 4 * (3 + 2) - 10;" and ("","a=10"). Result: Test Passed.
Comparing "b := (2 * 2) - (3 - 1);" and ("","b=2"). Result: Test Passed.
Comparing "counter := 5; sum := 0; while (not(counter == 0)) do (sum := sum + counter; counter := counter - 1;);" and ("","counter=0,sum=15"). Result: Test Passed.
Comparing "n := 4; factorial := 1; while (not(n == 1)) do (factorial := factorial * n; n := n - 1;);" and ("","factorial=24,n=1"). Result: Test Passed.
Comparing "if (1 == 1) then x := 100; else x := 200;" and ("","x=100"). Result: Test Passed.
Comparing "x := 5; if x == 5 then x := x * 2; else x := x + 2;" and ("","x=10"). Result: Test Passed.
Comparing "y := 0; if (y == 0) then y := 9; else y := 1;" and ("","y=9"). Result: Test Passed.
Comparing "if (2 <= 3) then a := 4; else a := 5; if (a == 4) then b := a * 2; else b := a + 2;" and ("","a=4,b=8"). Result: Test Passed.
Comparing "x := 3; while (not(x == 0)) do (x := x - 1;);" and ("","x=0"). Result: Test Passed.
Comparing "counter := 3; total := 0; while (not(counter == 0)) do (total := total + 5; counter := counter - 1;);" and ("","counter=0,total=15"). Result: Test Passed.
Comparing "n := 5; sum := 0; while (not(n == 0)) do (sum := sum + n; n := n - 1;);" and ("","n=0,sum=15"). Result: Test Passed.
Comparing "if (10 == 10) then a := 1; else a := 2; while (not(a == 5)) do (a := a + 1;);" and ("","a=5"). Result: Test Passed.
Comparing "count := 1; sum := 0; while (count <= 5) do (sum := sum + count; count := count + 1;);" and ("","count=6,sum=15"). Result: Test Passed.
Comparing "num := 4; product := 1; while (num <= 1) do (product := product * num; num := num - 1;);" and ("","num=4,product=1"). Result: Test Passed.
Comparing "i := 1; total := 0; while (i <= 4) do (total := total + i * 2; i := i + 1;);" and ("","i=5,total=20"). Result: Test Passed.
Comparing "x := 5; while (x <= 0) do (x := x - 1;);" and ("","x=5"). Result: Test Passed.
Comparing "counter := 2; result := 0; while (counter <= 4) do (result := result + counter; counter := counter + 1;);" and ("","counter=5,result=9"). Result: Test Passed.
Comparing "if (3 <= 4) then a := 2; else a := 3;" and ("","a=2"). Result: Test Passed.
Comparing "a := 5; b := 10; if (b <= a) then c := a + b; else c := a - b;" and ("","a=5,b=10,c=-5"). Result: Test Passed.
Comparing "x := 10; while (x <= 15) do (x := x + 1;);" and ("","x=16"). Result: Test Passed.
Comparing "count := 1; total := 0; while (count <= 3) do (total := total + 2; count := count + 1;);" and ("","count=4,total=6"). Result: Test Passed.
Comparing "if (1 <= 1) then x := 100; else x := 200;" and ("","x=100"). Result: Test Passed.
Comparing "y := 5; if (y <= 5) then y := y * 2; else y := y + 2;" and ("","y=10"). Result: Test Passed.
Comparing "z := 7; if (z <= 6) then z := 1; else z := 2;" and ("","z=2"). Result: Test Passed.
Comparing "n := 3; sum := 0; while (n <= 5) do (sum := sum + n; n := n + 1;);" and ("","n=6,sum=12"). Result: Test Passed.
Comparing "if (4 <= 3) then a := 4; else a := 5; if (a <= 5) then b := a * 2; else b := a + 2;" and ("","a=5,b=10"). Result: Test Passed.
Comparing "x := 1; while (x <= 10) do (x := x + 2;);" and ("","x=11"). Result: Test Passed.
44 out of 44 tests passed.
All tests completed.
..............
```

Conclusion

The Haskell Compiler has been successfully implemented in Haskell, using the GHCi compiler. The creation of a low-level machine with a variety of instructions for arithmetic and boolean operations demonstrates significant learning of Haskell knowledge and functional programming.

Throughout this report, we detail the strategies used for both parts of the project, detailing the decisions we made and exemplifying with code excerpts.

Finally, we explored each component, from the establishment of stack and state types and the intricate design of machine instructions and error handling mechanisms to the development of a parser, capable of transforming program strings into a structured format.

Bibliography

To develop this project we used the material given to us on the UC moodle. Including the Parsing in Haskell, DataTypes and Módulos e Abstract Data TypesFicheiro slides.