Simple language compiler and interpreter

This project was developed for the course **Functional and Logic Programming**, with the intent to write a machine for a low level set of instructions and the appropriate compiler for the language that is intended to operate with it.

Group Information

Class: 3LEIC14

Group: T14_G06

Members:

Student Number	Name	Contribution
202105385	Luís Du	34 %
202005832	Luís Sousa	33 %
202108783	Tiago Gouveia	33 %

1. Instructions and Structures

1.1. Data types

In order to make sure the language had employed the correct type restrictions, we created a type called `Value`, employed in both the `Stack` and the `State`.

```
data Value = I Integer | B Bool
```

Operators were overloaded to filter out incorrect usage, and restrict them to the correspondent types, i.e. trying to `add` Booleans or trying to `neg` an Integer.

1.1.1. Stack

The **stack** is one of the two fundamental structures of any machine, used to store values to perform operations, different from the state because these arent permanent, and are constantly under change.

It is defined by `[Value]`, a list of our custom type, `Value`. Despite being a list, we only interact with it through stack-like operations: `push` and `pop`, therefore making it essentially a stack.

In addition to the aforementioned functions, some utility functions are present to help the usage of the stack, without violating the base concept:

Function	Description
`createEmptyStack`	Creates an empty stack
`top`	Displays the top element of the stack
`size`	Displays the number of elements on the stack
`pop`	Removes the top element from the stack
`push`	Adds an element to the top of the stack
`stack2Str`	Auxiliary function used to display the stack, useful for debugging and testing.

1.1.2. State

The **state** is the other fundamental structure, used to store keys and values of variables the code instructs it to store for later, not removing them upon consulting the values, and suffering less mutations on average than the stack.

It is defined with the help of the library `Data.Map`, mapping `String`, the keys, to `Value`, the values.

We implemented a series of functions that help us interact and navegate the stack as we need it, always basing ourselves on the library's supplied functions:

Function	Description
`createEmptyState`	Creates an empty state calling `Data.Map.empty`
`push`	Calls `Data.Map.insert` to insert a new key value pair into the map, or updating an existing one
`find`	Calls `Data.Map.lookup` to return the value held by the desired key
`state2Str`	Auxiliary function used to display the state, useful for debugging and testing. Accomplished by usage of the function `Data.Map.mapAccumWithKey` and a lambda function to navegate through the map and synthesise it in one string

1.2. Operations

The following are operations that our machine is able to process and execute, with the help of the stack and the state as defined above:

Operation	Description
`add`	Adds two integers together and pushes the result to the stack
`mult`	Multiplies two integers and pushes the result to the stack
`sub`	Subtracts two integers and pushes the result to the stack
`tru`	Pushes `True` to the stack, part of the statement assessment
`fals`	Pushes `False` to the stack, part of the statement assessment

Operation	Description
`equ`	Statement assessment that checks if two `Value`s of the same type are equal and pushes the result to the stack
`le`	Statement assessment that checks if a `Value Integer` is less or equal to another and pushes the result to the stack
`bAnd`	Boolean operation that checks if both operands are `True` and pushes the result to the stack
`neg`	Boolean operation that reverses a boolean and pushes the result to the stack
`fetch`	Statement that retrieves a `Value` from the state via a key and pushes it to the stack
`store`	Statement that stores a `Value` under a key in the state
`noop`	Dummy instruction that returns the input stack and store
`branch`	Statement that selects which code to execute based on the top value of the stack, mandatorily a boolean
`loop`	Statement that checks a supplied condition and executes the code until the condition isn't true

2. Imperative Programming Language

Once we have our machine defined, it becomes possible to develop a compact imperative language. This language operates with arithmetic and boolean expressions, allowing statements in the form of assignments such as x := a, sequences of statements denoted by (instr1 ; instr2), conditional statements if-then-else, and iterative constructs with while loops.

2.1. Expressions and Statements

The following data types were used to handle both arithmetic and boolean expressions, as well as statements.

2.1.1. Arithmetic expressions - *Aexp*

`ALit` is a sum type that represents the literals used in arithmetic expressions, that can be either integers (`IntValue`) or variables (`IntVariable`).

```
data ALit = IntValue Integer | IntVariable String
```

`Aexp` is a recursive algebraic data type that encompasses various forms of arithmetic expressions, including literals (`IntLit`), addition (`IntAdd`), multiplication (`IntMult`), and subtraction (`IntSub`).

```
data Aexp
  = IntLit    ALit
  | IntAdd    Aexp Aexp
  | IntMult   Aexp Aexp
  | IntSub    Aexp Aexp
```

2.1.2. Boolean expressions - *Bexp*

Similarly to arithmetic expressions, boolean expression are defined by `Bexp`, a recursive data type that encompasses various forms of boolean expressions, including boolean literals `BoolLit`, negation `NOT`, logical `AND` operations, equality comparisons for both boolean (`BoolEqual`) and arithmetic expressions (`IntEqual`), as well as a comparison for less-or-equal-to (`IntLe`) between arithmetic expressions.

```
data Bexp
  = BoolLit    Bool
  | BoolNeg    Bexp
  | BoolAnd    Bexp Bexp
  | BoolEqual  Bexp Bexp
  | IntEqual   Aexp Aexp
  | IntLe      Aexp Aexp
```

2.1.3. Statements - *Stm*

In our language, statements could be:

- `IfStm` - if then else statements, consisting of a boolean expression, followed by two distinct lists of statements: one for the if condition and another for the else condition;
- `AssignStm` - Assignments of the form variable := arithmetic expression;
- `LoopStm` - while loop, consisting of boolean expression followed by a sequence of statements, forming the body of the while loop;
- `SequenceOfStm` - a list of statements

```
data Stm
  = IfStm Bexp [Stm] [Stm]
  | LoopStm Bexp [Stm]
  | AssignStm String Aexp
  | SequenceOfStm [Stm]
```

2.1.4. Program

`Program` is simply a list of statements:

type Program = [Stm]

2.2. Compiler

To define a compiler from a program in this small imperative language into a list of machine instructions, we can use two auxiliary functions which compile arithmetic and boolean expressions, respectively.

2.2.1. Compiler of arithmetic expressions - *compA*

In arithmetic expressions, 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`. Hence, we can implement the following recursive structure:

```
compA (IntAdd exp1 exp2) = compA exp2 ++ compA exp1 ++ [Add]
compA (IntMult exp1 exp2) = compA exp2 ++ compA exp1 ++ [Mult]
compA (IntSub exp1 exp2) = compA exp2 ++ compA exp1 ++ [Sub]
```

As our base case, we could either have a variable or an integer, leading to a `Fetch` or `Push` instruction, respectively:

```
compA (IntLit (IntValue n)) = [Push n]
compA (IntLit (IntVariable var)) = [Fetch var]
```

2.2.2. Compiler of boolean expressions - *compB*

Analogously, boolean expressions can be compiled to code using recursion:

```
compB (BoolNeg exp) = compB exp ++ [Neg]
compB (BoolAnd exp1 exp2) = compB exp2 ++ compB exp1 ++ [And]
compB (BoolEqual exp1 exp2) = compB exp2 ++ compB exp1 ++ [Equ]
compB (IntEqual exp1 exp2) = compA exp2 ++ compA exp1 ++ [Equ]
compB (IntLe exp1 exp2) = compA exp2 ++ compA exp1 ++ [Le]
```

As our base case, we have:

```
compB (BoolLit n)
  | n         = [Tru]
  | otherwise = [Fals]
```

2.2.3. Compiler of program - *compile*

As we have implemented compilers for both arithmetic and boolean expressions, we just need to compile the statements. Since our `Program` is a list of statements, we can recursively traverse through each statement, compiling them into machine-executable code by employing specific compilation rules for each statement type.

```
compile :: Program -> Code
compile [] = []
compile (statement : rest) =
  case statement of
    AssignStm var aExp -> compA aExp ++ [Store var] ++ compile rest
    IfStm cond ifBlock elseBlock -> compB cond ++ [Branch (compile ifBlock)
(compile elseBlock)] ++ compile rest
    LoopStm cond loopBody -> Loop (compB cond) (compile loopBody) : compile
rest
```

2.3 Parser

The parser was developed leveraging the **Parsec** library, accessible through this link. This library served as the foundation for constructing the lexer and the parser, enabling the creation of robust parsing functionality within our program.

2.3.1. Lexer

The `lexer` was created using the constructor `emptyDef` from `Text.ParserCombinators.Parsec.Language`, that sets up the syntax and reserved keywords for parsing purposes:

```
languageDefinition =
   emptyDef { Token.identStart      = lower
            , Token.identLetter     = alphaNum
            , Token.reservedNames   = [ "if"
                                      , "then"
                                      , "else"
                                      , "while"
                                      , "do"
                                      , "True"
                                      , "False"
                                      , "not"
                                      , "and"
                                      ]
            , Token.reservedOpNames = ["+", "-", "*"
                                     ,"==", "=", "<=", "and", "not"
                                      , ":="
                                      ]
            }
```

The lexer can be created by simply doing:

```
lexer = Token.makeTokenParser languageDefinition
```

2.3.2. Token Parsers

To facilitate parsing, the lexer's token parsers were extracted:

```
variable   = Token.identifier lexer
reserved   = Token.reserved   lexer
reservedOp = Token.reservedOp lexer
parens     = Token.parens     lexer
integer    = Token.integer    lexer
semiColon  = Token.semi       lexer
whiteSpace = Token.whiteSpace lexer
```

2.3.3. Expression Parsers

Regarding `Aexp` and `Bexp`, the expression parsers were built using the function `buildExpressionParser` provided by the Parsec library.

```
aritExp :: Parser Aexp
aritExp = buildExpressionParser aOperators aritParser

boolExp :: Parser Bexp
boolExp = buildExpressionParser bOperators boolParser
```

The operator precedence and associativity were specified as follow:

```
aOperators :: [[Operator Char st Aexp]]
aOperators = [ [Infix  (reservedOp "*"   >> return IntMult) AssocLeft]
             , [Infix  (reservedOp "+"   >> return IntAdd) AssocLeft,
                Infix  (reservedOp "-"   >> return IntSub) AssocLeft]
              ]

bOperators :: [[Operator Char st Bexp]]
bOperators = [ [Prefix (reservedOp "not" >> return BoolNeg)          ],
                [Infix (reservedOp "=" >> return BoolEqual) AssocLeft
],
              [Infix  (reservedOp "and" >> return BoolAnd) AssocLeft]
             ]
```

Finally, we have to define the terms:

```
intParser :: Parser ALit
intParser = fmap IntValue integer Parsec.<|> fmap IntVariable variable

aritParser :: Parser Aexp
aritParser =  parens aritExp Parsec.<|> fmap IntLit intParser
```

```
boolParser :: Parser Bexp
boolParser =  parens boolExp
     Parsec.<|> (reserved "True"  >> return (BoolLit True) )
     Parsec.<|> (reserved "False" >> return (BoolLit False) )
     Parsec.<|> intCompareParser

intCompareParser :: Parser Bexp
intCompareParser =
   do a1 <- aritExp
      op <- comp
      a2 <- aritExp
      return $ op a1 a2

comp :: Parser (Aexp -> Aexp -> Bexp)
comp = (reservedOp "<=" >> return IntLe) Parsec.<|> (reservedOp "==" >>
return IntEqual)
```

2.3.4. Statement Parsers

A statement is parsed using the function `statementParser` that specific parsers are called depending on the type of the statement:

```
statementParser :: Parser Stm
statementParser =  parens statementParser
           Parsec.<|> ifParser
           Parsec.<|> loopParser
           Parsec.<|> assignParser
```

2.3.5. Statements Parser

To process a sequence of statements, the `statementParser` was applied zero or more times using `many` from `Text.Parsec.Combinator`:

```
statementsParser :: Parser [Stm]
statementsParser = parens statementsParser Parsec.<|> Parsec.many
statementParser
```

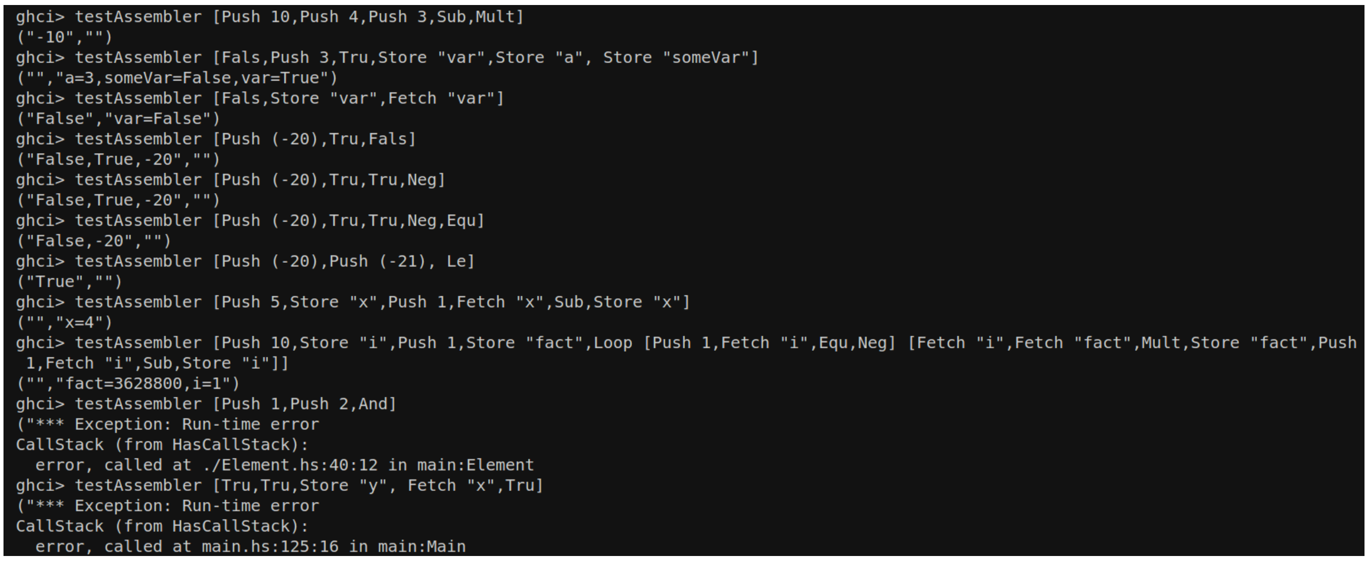
2.3.6. Parse

At last, we can construct our parse function as follows. Here, we utilize `statementsParser` to process the input, ensuring that the entirety of the input is consumed using `Parsec.eof` for validation.

```haskell
parse :: String -> Program
parse str =
  case Parsec.parse (whiteSpace >> statementsParser <* Parsec.eof) "" str
of
    Left e -> error "Run-time error"
    Right r -> r
```

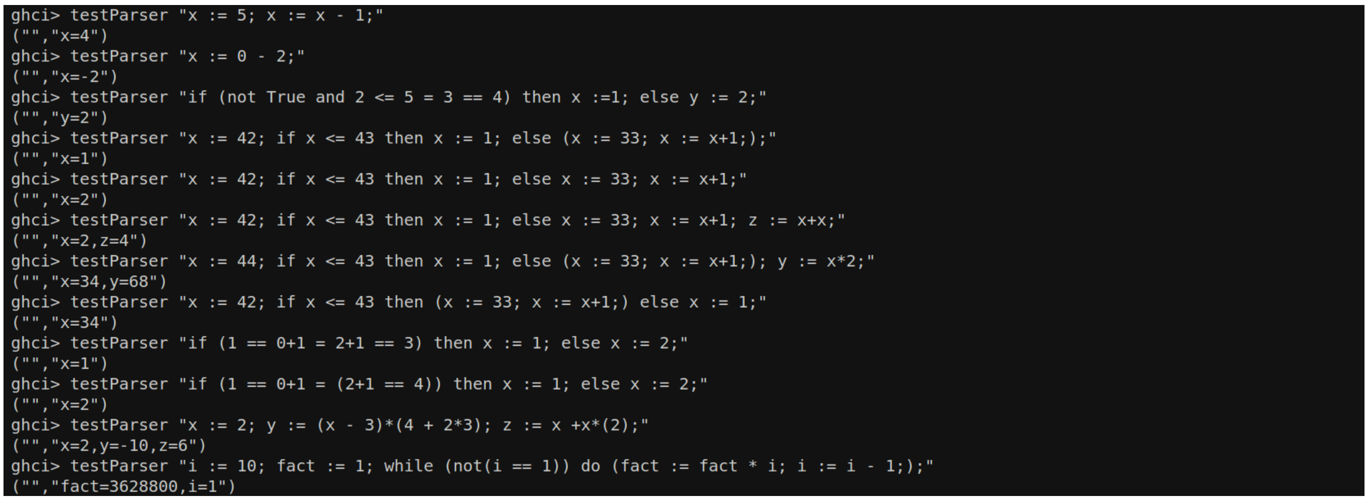
Examples of execution

TestAssembler



```
ghci> testAssembler [Push 10,Push 4,Push 3,Sub,Mult]
("-10","")
ghci> testAssembler [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"]
("","a=3,someVar=False,var=True")
ghci> testAssembler [Fals,Store "var",Fetch "var"]
("False","var=False")
ghci> testAssembler [Push (-20),Tru,Fals]
("False,True,-20","")
ghci> testAssembler [Push (-20),Tru,Tru,Neg]
("False,True,-20","")
ghci> testAssembler [Push (-20),Tru,Tru,Neg,Equ]
("False,-20","")
ghci> testAssembler [Push (-20),Push (-21), Le]
("True","")
ghci> testAssembler [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"]
("","x=4")
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")
ghci> testAssembler [Push 1,Push 2,And]
("*** Exception: Run-time error
CallStack (from HasCallStack):
  error, called at ./Element.hs:40:12 in main:Element
ghci> testAssembler [Tru,Tru,Store "y", Fetch "x",Tru]
("*** Exception: Run-time error
CallStack (from HasCallStack):
  error, called at main.hs:125:16 in main:Main
```

TestParser



```
ghci> testParser "x := 5; x := x - 1;"
("","x=4")
ghci> testParser "x := 0 - 2;"
("","x=-2")
ghci> testParser "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;"
("","y=2")
ghci> testParser "x := 42; if x <= 43 then x := 1; else (x := 33; x := x+1;);"
("","x=1")
ghci> testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1;"
("","x=2")
ghci> testParser "x := 42; if x <= 43 then x := 1; else x := 33; x := x+1; z := x+x;"
("","x=2,z=4")
ghci> testParser "x := 44; if x <= 43 then x := 1; else (x := 33; x := x+1;); y := x*2;"
("","x=34,y=68")
ghci> testParser "x := 42; if x <= 43 then (x := 33; x := x+1;) else x := 1;"
("","x=34")
ghci> testParser "if (1 == 0+1 = 2+1 == 3) then x := 1; else x := 2;"
("","x=1")
ghci> testParser "if (1 == 0+1 = (2+1 == 4)) then x := 1; else x := 2;"
("","x=2")
ghci> testParser "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);"
("","x=2,y=-10,z=6")
ghci> testParser "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i - 1;);"
("","fact=3628800,i=1")
```