T12_G07 | Assignment 2

Authors and contribution:

Miguel Pedrosa, up202108809: 50%
António Rama, up202108801: 50%

Part 1

Data and Types

To define the data and types for this part, we started by defining a **StackItem** data, that can be either an Integer or a Boolean:

```
data StackItem = IntItem Integer | BoolItem Bool
```

After this we defined the **Stack** type, that is a list of StackItems (a list of Booleans and Integers), and the **State** type that will store the variables and their values:

```
type Stack = [StackItem]
type State = [(String, StackItem)]
```

We also defined two very simple functions that create an empty Stack and an empty State:

```
createEmptyStack :: Stack
createEmptyStack = []

createEmptyState :: State
createEmptyState = []
```

Conversion to strings

The next step was to define a function that would convert a Stack to a String with the values represented in the following structure: separated by commas and without spaces, with the leftmost value representing the top of the stack.

The **item2Str** auxiliary function takes a StackItem and converts it into a string using the show function.

The **stack2Str** function then converts an entire stack into a string. It does this by converting the first stack item using item2Str and then concatenating the rest of the items, separated by commas, using concatMap.

```
item2Str :: StackItem -> String
item2Str (IntItem i) = show i
item2Str (BoolItem b) = show b

stack2Str :: Stack -> String
stack2Str [] = ""
stack2Str (x:xs) = item2Str x ++ concatMap ((',' :) . item2Str) xs
```

We also needed to convert State into a String. 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 ”=”.

The **storageTuple2Str** auxiliary function takes a State tuple and converts it into a string in the format "key=value", where the key is the string and the value is the string representation of the StackItem, obtained using the previously defined item2Str function.

The **state2Str** function converts a State into a string. It first sorts the state by the string keys of the tuples using sortBy (comparing fst). Then, it converts the sorted list of storage tuples into a string. If the state is empty, it returns an empty string. Otherwise, it uses the storageTuple2Str function to convert the first storage tuple and then concatenates the rest of the tuples, separated by commas, using concatMap.

```
storageTuple2Str :: (String, StackItem) -> String
storageTuple2Str (a, b) = a ++ "=" ++ item2Str b

state2Str :: State -> String
state2Str [] = ""
state2Str (x:xs) =
    let sortedStorage = sortBy (comparing fst) (x:xs)
    in case sortedStorage of
        [] -> ""
        (x:xs) -> storageTuple2Str x ++ concatMap ((',' :) . storageTuple2Str) xs
```

Program interpreter - run function

The goal of this function is, given a list of instructions (Code), a stack that is initially empty, and a State (a.k.a Storage) , to run the list of instructions returning as output an empty code list, a stack and the output values in the State.

Each case handles the current stack and state according to the operation specified by the instruction and then recursively calls run with the rest of the instructions, updated stack, and updated state. If an operation encounters an invalid state (like trying to add two values when there aren't enough values on the stack), it results in a run-time error.

```
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack, state) = ([], stack, state) -- Base case, no Code left
run ((inst:rest), stack, state) = case inst of

  Push n ->
    run (rest, IntItem n : stack, state)
```

```
Add ->
  case stack of
    (IntItem n1 : IntItem n2 : stackRest) -> run (rest, IntItem (n1 + n2) : stackRest, state)
    _ -> error "Run-time error"

Sub ->
  case stack of
    (IntItem n1 : IntItem n2 : stackRest) -> run (rest, IntItem (n1 - n2) : stackRest, state)
    _ -> error "Run-time error"

Mult ->
  case stack of
    (IntItem n1 : IntItem n2 : stackRest) ->  run (rest, IntItem (n1 * n2) : stackRest, state)
    _ -> error "Run-time error"

Fals ->
  run (rest, BoolItem False : stack, state)

Tru ->
  run (rest, BoolItem True : stack, state)

Store var ->
  case stack of
    (val : stackRest) ->
      let updatedState = case lookup var state of
            Just _  -> map (\(k, v) -> if k == var then (k, val) else (k, v)) state
            Nothing -> (var, val) : state
      in run (rest, stackRest, updatedState)
    _ -> error "Run-time error"

Fetch var ->
  case lookup var state of
    Just val -> run (rest, val : stack, state)
    Nothing -> error "Run-time error"

Neg ->
  case stack of
    (BoolItem b : stackRest) -> run (rest, BoolItem (not b) : stackRest, state)
    _ -> error "Run-time error"

Equ ->
  case stack of
    (IntItem a : IntItem b : stackRest) -> run (rest, BoolItem (a == b) : stackRest, state)
    (BoolItem a : BoolItem b : stackRest) -> run (rest, BoolItem (a == b) : stackRest, state)
    _ -> error "Run-time error"

Le ->
  case stack of
    (IntItem a : IntItem b : stackRest) -> run (rest, BoolItem (a <= b) : stackRest, state)
    _ -> error "Run-time error"

Noop ->
  run (rest, stack, state)

Branch code1 code2 ->
  case stack of
    (BoolItem True : stackRest) -> run (code1 ++ rest, stackRest, state)
    (BoolItem False : stackRest) -> run (code2 ++ rest, stackRest, state)
    _ -> error "Run-time error"

Loop code1 code2 ->
  run (code1 ++ [Branch (code2 ++ [Loop code1 code2]) [Noop]] ++ rest, stack, state)

And ->
  case stack of
    (BoolItem b1 : BoolItem b2 : stackRest) -> run (rest, BoolItem (b1 && b2) : stackRest, state)
    _ -> error "Run-time error"
```

Tests

We tested our program with the provided **testAssembler** function:

```
testAssembler :: Code -> (String, String)
testAssembler code = (stack2Str stack, state2Str state)
  where (_,stack,state) = run(code, createEmptyStack, createEmptyState)
```

All the provided tests ran successfully.

(We also made our own extra tests to make sure we covered as many cases as possible)

```
testAssembler [Tru, Fals, And] == ("False", "")
testAssembler [Tru, Tru, And] == ("True", "")
testAssembler [Push 5, Push 3, Add, Push 2, Mult, Tru, Tru, And] == ("True,16", "")
testAssembler [Push 1, Store "n", Push 5, Loop [Fetch "n", Push 5, Le] [Fetch "n", Push 1, Add, Store "n"]] == ("","n=6")
testAssembler [Tru, Store "flag", Fetch "flag", Neg, Store "flag"] == ("","flag=False")
testAssembler [Push 1, Store "i", Push 0, Store "sum", Loop [Push 5, Fetch "i", Le] [Fetch "sum", Fetch "i", Add, Store "su
testAssembler [Tru, Store "flag", Push 1, Store "counter", Loop [Push 2, Fetch "counter", Le] [Fetch "flag", Neg, Store "fl

-- Examples (developed by us):
testAssembler [Tru, Fals, And] == ("False", "")
testAssembler [Tru, Tru, And] == ("True", "")
```

Part 2

Data and Types

Three data types were defined to represent the abstract syntax of a simple programming language: arithmetic expressions (Aexp), boolean expressions (Bexp), and statements (Stm). These data types are foundational for constructing an abstract syntax tree (AST), which is a data structure widely used in compilers and interpreters to represent the structure of source code.

Aexp (Arithmetic Expression) is defined with the following constructors:

- Var String: Represents a variable where String is the name of the variable.
- Num Integer: Represents a numeric constant.
- AAdd Aexp Aexp: Represents the addition of two arithmetic expressions.
- ASub Aexp Aexp: Represents the subtraction between two arithmetic expressions.
- AMult Aexp Aexp: Represents the multiplication of two arithmetic expressions.
- Each of these constructors allows the construction of an arithmetic expression tree where leaves are variables or numeric constants, and internal nodes are arithmetic operations.

Bexp (Boolean Expression) is defined with these constructors:

- AEq Aexp Aexp: Represents the equality comparison between two arithmetic expressions.
- BTrue: Represents the boolean constant true.
- BFals: Represents the boolean constant false.
- BLe Aexp Aexp: Represents a 'less than or equal to' comparison between two arithmetic expressions.
- BAnd Bexp Bexp: Represents the logical conjunction (AND) of two boolean expressions.
- BEqBool Bexp Bexp: Represents the equality comparison between two boolean expressions.
- BNot Bexp: Represents the logical negation (NOT) of a boolean expression.
- The Bexp type models the boolean expressions that can be formed using arithmetic comparisons, logical operations, and boolean constants.

Stm (Statement) has the following constructors:

- AssignStm String Aexp: Represents an assignment statement, assigning an arithmetic expression to a variable.
- IfStm Bexp [Stm] [Stm]: Represents an 'if' statement with a boolean expression as the condition, a list of statements for the 'then' branch, and a list of statements for the 'else' branch.
- WhileStm Bexp [Stm]: Represents a 'while' loop with a boolean condition and a list of statements as the loop body.
- The Stm type is used to construct statements in this simple language, including control flow constructs like 'if' conditions and 'while' loops.

The type Program is defined as a list of Stm, representing a sequence of statements, which constitutes a complete program in this language. This list structure allows for the sequential execution of statements, which is typical in imperative programming languages.

Overall, these data types are designed to be simple yet expressive enough to capture key elements of programming language syntax. The abstract nature of these types allows for manipulation and interpretation of the code without concern for the intricacies of actual syntactic notation, making it easier to reason about program structure and behavior during analysis and execution.

```
data Aexp = Var String             -- Variable
          | Num Integer            -- Constant
          | AAdd Aexp Aexp         -- Addition
          | ASub Aexp Aexp         -- Subtraction
          | AMult Aexp Aexp        -- Multiplication
          deriving (Show, Eq)

data Bexp = AEq Aexp Aexp
          | BTrue                  -- True
          | BFals                  -- False
          | BLe Aexp Aexp          -- Less than or equal to
          | BAnd Bexp Bexp         -- Logical AND
          | BEqBool Bexp Bexp      -- Boolean equality
          | BNot Bexp              -- Logical NOT
          deriving (Show, Eq)

data Stm = AssignStm String Aexp   -- Assignment
         | IfStm Bexp [Stm] [Stm]  -- 'If' condition, 'then' branch, 'else' branch
         | WhileStm Bexp [Stm]     -- 'While' condition and loop body (list of statements)
         deriving (Show, Eq)

type Program = [Stm]
```

Compiler Functions

The compiler translates high-level constructs into a list of instructions (Code) that could be interpreted by a stack-based "machine". This machine executes the code sequentially, manipulating a stack to evaluate expressions and control the flow of execution.

compA Function:

The compA function compiles arithmetic expressions into stack machine code. It operates recursively on the structure of the arithmetic expression. For example:

Var x compiles to a Fetch x instruction, which retrieves the value of the variable x from the environment and pushes it onto the stack. Num n compiles to a Push n instruction, placing the numeric constant n onto the stack. For compound expressions like AAdd, ASub, and AMult, it compiles the subexpressions first, ensuring that their results are on the stack before applying the corresponding arithmetic operation (Add, Sub, Mult).

compB Function:

The compB function compiles boolean expressions into stack machine code. Similar to compA, it works recursively:

AEq and BLe compile into code that pushes the results of two arithmetic expressions onto the stack and then compares them using Equ or Le. BNot inverts the truth value of its expression with a Neg instruction. BAnd and BEqBool ensure both boolean expressions are evaluated and placed on the stack before performing a logical And operation or equality check. BTrue and BFals compile to Tru and Fals, pushing the corresponding boolean constants onto the stack.

compile Function:

The compile function is the entry point for compiling a complete program. It processes each statement in the program and produces a list of instructions:

AssignStm compiles an arithmetic expression and then stores the result in a variable with the Store instruction. IfStm compiles the boolean condition and then branches to the code for the 'then' or 'else' part depending on the condition's value. WhileStm compiles into a loop that evaluates the boolean condition and executes the loop body repeatedly as long as the condition is true. This compiler abstracts the underlying details of machine code generation, providing a higher-level interface for translating structured programming constructs into executable machine instructions.

```
{-
compA:
Type: Aexp -> Code

Purpose: compA takes an arithmetic expression (Aexp) and returns a list of instructions (Code).
The order of operations ensures the operands are correctly placed on the stack before the operation is performed.
Input: An arithmetic expression (Aexp).
Output: A list of instructions (Code).
-}
compA :: Aexp -> Code

compA (Var x) = [Fetch x]
compA (Num n) = [Push n]
compA (AMult a1 a2) = compA a2 ++ compA a1 ++ [Mult]
compA (AAdd a1 a2) = compA a2 ++ compA a1 ++ [Add]
compA (ASub a1 a2) = compA a2 ++ compA a1 ++ [Sub]


{-
compB:
Type: Bexp -> Code

Purpose: compB takes a boolean expression (Bexp) and returns a list of instructions (Code).
The order of operations ensures the operands are correctly placed on the stack before the operation is performed.
Input: A boolean expression (Bexp).
Output: A list of instructions (Code).
-}
compB :: Bexp -> Code

compB (AEq b1 b2) = compA b2 ++ compA b1 ++ [Equ]
compB (BLe b1 b2) = compA b2 ++ compA b1 ++ [Le]
compB (BNot b1) = compB b1 ++ [Neg]
compB (BAnd b1 b2) = compB b2 ++ compB b1 ++ [And]
compB (BEqBool b1 b2) = compB b2 ++ compB b1 ++ [Equ]
compB BTrue = [Tru]
compB BFals = [Fals]


{-
compile:
Type: Program -> Code

Purpose: compile takes a program (list of statements) and returns a list of instructions (Code).
Input: A program (list of statements).
Output: A list of instructions (Code).
-}
compile :: Program -> Code
compile program = foldr (\stm acc -> compileSingle stm ++ acc) [] program
  where
    compileSingle (AssignStm var aexp) = compA aexp ++ [Store var]
    compileSingle (IfStm bexp stm1 stm2) = compB bexp ++ [Branch (compile stm1) (compile stm2)]
    compileSingle (WhileStm bexp stm) = [Loop (compB bexp) (compile stm)]
```

Lexer

This is a lexical analyzer. Its purpose is to read the input program, a string of characters, and convert it into a list of tokens that represent the syntactic components necessary for further processing by a parser.

Here's a breakdown of its functionality:

1. Whitespace Handling
    - The lexer skips over whitespace characters, effectively ignoring them. This simplifies the token stream and focuses on the meaningful characters that define the syntax of the program.
2. Number Handling
    - When the lexer encounters a digit, it uses the span function to consume all contiguous digits, forming a numeric literal token. This token is then added to the list of tokens, and the lexer continues processing the rest of the input string.

3. Word Handling
    - If the first character is alphabetical, the lexer again uses span but this time captures a sequence of alphanumeric characters. This token can be an identifier like a variable name or a keyword. The lexer then:
4. Converts the token to lowercase to ensure case-insensitivity.
    - Checks if the token matches any of the predefined keywords of the language such as "if", "then", "else", "not", "true", "false", "while", and "do". If it matches, the keyword is recognized as a distinct token.
    - If the token is not a keyword, it is considered an identifier and is added to the token list.
5. Special Characters and Operators
    - The lexer recognizes specific single and double-character tokens. Single-character tokens include semicolons, parentheses, and simple operators like +, -, and *. Double-character tokens include compound operators like := for assignment, == for equality comparison, and <= for the 'less than or equal to' comparison. These are detected by looking ahead in the input string for specific character combinations.
6. Error Handling The lexer has a default case at the end that catches any unrecognized characters. If such a character is encountered, the lexer stops with an error, indicating that it found an unrecognized token. This ensures that only valid tokens are allowed, maintaining the syntactic integrity of the input program.

In summary, the lexer serves as the first phase of the process, transforming raw text into a structured sequence of tokens. This tokenized output is essential for the next phase, parsing, where the structure of the program is analyzed and an abstract syntax tree is built.

```
lexer
Input: A String (inputString), which is the program text to be tokenized.
Output: A list of String tokens representing the lexical elements of the input program.

Purpose:
The function processes the input string using pattern matching and recursion to build the list of tokens.

Tokenization Process:
Whitespace Handling: If the current character is a space, it's ignored, and lexer continues with the rest of the characters
Number Handling: If the current character is a digit, lexer captures the entire number (sequence of digits) as a token and
Word Handling: If the current character is a letter, lexer captures the entire word. It then:
Converts the word to lowercase for case-insensitive keyword matching.
Checks if the word is a language keyword like "if", "then", "else", etc., and treats it as a separate token.
If it's not a keyword, the word is treated as an identifier (like a variable name).
Special Characters and Operators: The lexer recognizes specific single and double characters as tokens, such as:
Semicolons (;), parentheses, and operators like +, -, *, :=, ==, <=, and =.

Error Handling:
The function includes a catch-all case at the end. If it encounters an unrecognized character, it generates an error messag
-}
type Lexer = String -> [String]

lexer :: String -> [String]
lexer [] = []
lexer (c:cs)
    | isSpace c = lexer cs
    | isDigit c = let (number, rest) = span isDigit (c:cs) in number : lexer rest
    | isAlpha c =
        let (word, rest) = span isAlphaNum (c:cs)
            lowerWord = map toLower word
            in case lowerWord of
                 -- Match and separate keywords
                 "if" -> "if" : lexer rest
                 "then" -> "then" : lexer rest
                 "else" -> "else" : lexer rest
                 "not" -> "not" : lexer rest
                 "true" -> "True" : lexer rest
                 "false" -> "False" : lexer rest
                 "while" -> "while" : lexer rest
                 "do" -> "do" : lexer rest
                 "and" -> "and" : lexer rest
                 -- Handle non-keyword identifiers
                 _ -> word : lexer rest
    | c == ';' = ";" : lexer cs
    | c == ':' && not (null cs) && head cs == '=' = ":=" : lexer (tail cs)  -- Recognize ':=' as an assignment operator
    | c == '+' = "+" : lexer cs
    | c == '-' = "-" : lexer cs
    | c == '*' = "*" : lexer cs
    | c == '=' && (not (null cs) && head cs == '=') = "==" : lexer (tail cs) -- Recognize '==' as a comparison operator
    | c == '=' = "=" : lexer cs
    | c == '(' = "(" : lexer cs
    | c == ')' = ")" : lexer cs
    | c == '{' = "{" : lexer cs
    | c == '}' = "}" : lexer cs
    | c == '<' && (not (null cs) && head cs == '=') = "<=" : lexer (tail cs) -- Recognize '<=' as a comparison operator
    | c == '<' = "<" : lexer cs
    | otherwise = error $ "lexer: Unrecognized character '" ++ [c] ++ "'"
```

Arithmetic Expressions Parsing

We followed the example given in the theoretical classes and adapted it to our implementation. Our strategy was the following:

1. Syntax Tree Construction:

- As the parser processes the input string, it constructs a syntax tree that represents the hierarchical structure of the arithmetic expression.
- The syntax tree is built incrementally as the parser encounters different components of the expression, such as integers, variables, operators, and parentheses.
- Each node of the tree corresponds to an arithmetic operation or operand, allowing for a clear representation of the expression's structure.

2. Recursive Descent Parsing:

- Recursive descent parsing is a top-down parsing technique where the parser recursively descends through the input, matching the input tokens to specific grammar rules.
- In this implementation, recursive descent parsing is used extensively, especially for handling operator precedence and nested expressions.
- For example, when parsing expressions like `2 * (3 + 4)`, the parser initially recognizes the multiplication operator (`*`) and then recursively parses the subexpression `(3 + 4)` within the parentheses.
- This recursive approach ensures that the correct precedence and associativity of operators are maintained.

3. Operator Precedence Handling:

- To handle operator precedence, the parser employs a series of functions with different levels of precedence. For example, `parseProdOrInt` handles multiplication, and `parseSumSubOrProdOrInt` handles addition and subtraction or reverts to `parseProdOrInt`.
- Higher-precedence operations are evaluated first, and lower-precedence ones are evaluated later. This is achieved by calling the appropriate parsing functions in the correct order.

4. Nested Expressions and Parentheses:

- When the parser encounters an opening parenthesis, it makes recursive calls to parse the expression inside the parentheses. This allows for the correct handling of nested expressions.
- The syntax tree reflects the nesting of expressions, with each level of nesting represented by a subtree.

5. Building the Syntax Tree:

- As the parser successfully processes tokens and constructs subexpressions, it combines these subexpressions into larger expressions, ultimately building the complete syntax tree.
- For example, when parsing `2 * (3 + 4)`, the parser constructs a syntax tree with a multiplication node having `2` as its left child and a subexpression node for `(3 + 4)` as its right child.

6. Return Values:

- Each parsing function returns a tuple containing an Aexp (representing an arithmetic expression) and the remaining tokens to be parsed.
- This return value is used to build the syntax tree incrementally, combining parsed expressions into larger expressions.

```
--Aexp Parser

{-
parseInt:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Parses an integer token from the input list.
Input: A list of string tokens.
Output: A Maybe tuple containing an Aexp (arithmetic expression) representing the parsed integer (or variable if not an int
-}
parseInt :: [String] -> Maybe (Aexp, [String])
parseInt [] = Nothing
parseInt (token:remainingTokens) =
    Just $ maybe (Var token, remainingTokens) (\numValue -> (Num numValue, remainingTokens)) (readMaybe token)


{-
parseProdOrInt:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Parses expressions involving multiplication or just an integer.
Input: A list of string tokens.
Output: A Maybe tuple containing an Aexp representing the product or integer and the remaining tokens.
-}
parseProdOrInt :: [String] -> Maybe (Aexp, [String])
parseProdOrInt tokens = do
  (leftExpr, remainingTokens) <- parseInt tokens
  parseMultExpr leftExpr remainingTokens


{-
parseMultExpr:
Type: Aexp -> [String] -> Maybe (Aexp, [String])

Purpose: Helper function to parse multiplication expressions.
Input: An Aexp representing the left expression and a list of remaining tokens.
Output: A Maybe tuple with an Aexp representing the multiplication and the remaining tokens.
-}
parseMultExpr :: Aexp -> [String] -> Maybe (Aexp, [String])
parseMultExpr leftExpr tokens = case tokens of
    "*":rest -> do
      (rightExpr, remainingTokens) <- parseProdOrInt rest
      Just (AMult leftExpr rightExpr, remainingTokens)
    _ -> Just (leftExpr, tokens)


{-
```

```
parseSumSubOrProdOrInt:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Parses expressions involving addition, subtraction, or multiplication.
Input: A list of string tokens.
Output: A Maybe tuple with an Aexp and the remaining tokens.
-}
parseSumSubOrProdOrInt :: [String] -> Maybe (Aexp, [String])
parseSumSubOrProdOrInt tokens = do
  (leftExpr, remainingTokens) <- parseProdOrInt tokens
  parseAddSubOrReturn leftExpr remainingTokens


{-
parseIntOrParentExpr:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Parses an integer or a parenthesized expression.
Input: A list of string tokens.
Output: A Maybe tuple with an Aexp for the parsed integer/parenthesized expression and the remaining tokens.
-}
parseIntOrParentExpr :: [String] -> Maybe (Aexp, [String])
parseIntOrParentExpr ("(":tokens) = parseParenExpr tokens
parseIntOrParentExpr (token:tokens) = parseNumOrVar token tokens
parseIntOrParentExpr [] = Nothing


{-
parseParenExpr:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Parses a parenthesized expression.
Input: A list of string tokens.
Output: A Maybe tuple with an Aexp for the parenthesized expression and the remaining tokens.
-}
parseParenExpr :: [String] -> Maybe (Aexp, [String])
parseParenExpr tokens =
  case parseSumSubOrProdOrIntOrParen tokens of
    Just (expr, ")":remainingTokens) -> Just (expr, remainingTokens)
    _ -> Nothing


{-
parseNumOrVar:
Type: String -> [String] -> Maybe (Aexp, [String])

Purpose: Parses a numeric literal or a variable.
Input: A string token and a list of remaining tokens.
Output: A Maybe tuple with an Aexp for the number/variable and the remaining tokens.
-}
parseNumOrVar :: String -> [String] -> Maybe (Aexp, [String])
parseNumOrVar token tokens =
  case readMaybe token :: Maybe Integer of
    Just num -> Just (Num num, tokens)
    Nothing -> Just (Var token, tokens)


{-
parseProdOrIntOrParen:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Parses products, integers, or parenthesized expressions.
Input: A list of string tokens.
Output: A Maybe tuple with an Aexp and the remaining tokens.
-}
parseProdOrIntOrParen :: [String] -> Maybe (Aexp, [String])
parseProdOrIntOrParen tokens = do
  (leftExpr, remainingAfterFirstParse) <- parseIntOrParentExpr tokens
  case remainingAfterFirstParse of
    "*":remainingAfterMult -> do
      (rightExpr, remainingAfterProd) <- parseProdOrIntOrParen remainingAfterMult
      return (AMult leftExpr rightExpr, remainingAfterProd)
    _ -> Just (leftExpr, remainingAfterFirstParse)


{-
parseSumSubOrProdOrIntOrParen:
Type: [String] -> Maybe (Aexp, [String])

Purpose: Defines a parser for expressions involving addition, subtraction, multiplication, or parentheses.
Input: A list of string tokens.
Output: A Maybe tuple with an Aexp and the remaining tokens.
-}
parseSumSubOrProdOrIntOrParen :: [String] -> Maybe (Aexp, [String])
parseSumSubOrProdOrIntOrParen tokens = do
  (leftExpr, remainingTokens) <- parseProdOrIntOrParen tokens
  parseAddSubOrReturn leftExpr remainingTokens


{-
parseAddSubOrReturn:
Type: Aexp -> [String] -> Maybe (Aexp, [String])

Purpose: Handles addition or subtraction, or returns the expression if neither is found.
Input: An Aexp representing the left expression and a list of tokens.
Output: A Maybe tuple with an Aexp and the remaining tokens.
-}
```

```
parseAddSubOrReturn :: Aexp -> [String] -> Maybe (Aexp, [String])
parseAddSubOrReturn leftExpr tokens =
  case tokens of
    "+":rest -> parseSumSubExpr AAdd leftExpr rest
    "-":rest -> parseSumSubExpr ASub leftExpr rest
    _ -> Just (leftExpr, tokens)


{-
parseSumSubExpr:
Type: (Aexp -> Aexp -> Aexp) -> Aexp -> [String] -> Maybe (Aexp, [String])

Purpose: Parses an operation (addition or subtraction) and continues parsing the rest of the expression.
Input: A function representing the operation, an Aexp for the left expression, and a list of tokens.
Output: A Maybe tuple with an Aexp and the remaining tokens.
-}
parseSumSubExpr :: (Aexp -> Aexp -> Aexp) -> Aexp -> [String] -> Maybe (Aexp, [String])
parseSumSubExpr op leftExpr tokens = do
  (rightExpr, remainingTokens) <- parseSumSubOrProdOrIntOrParen tokens
  return (op leftExpr rightExpr, remainingTokens)
```

Boolean Expressions Parsing

The boolean expression parser follows a similar design strategy as the arithmetic expression parser. It utilizes top-down parsing to construct a syntax tree that accurately represents the structure of boolean expressions, ensuring correct precedence and associativity of operators. The parser's modular structure and error-handling mechanisms make it a robust and extensible tool for parsing and interpreting boolean expressions.

1. Modular Parsing Approach:
    - The parser is organized into separate functions with well-defined responsibilities, following a modular approach similar to the arithmetic expression parser.
    - Each function handles a specific aspect of parsing, such as parentheses, comparisons, "not" expressions, and boolean operators.
2. Tokenization:
    - The input string is tokenized by the lexer into a list of string tokens before parsing, simplifying the parsing process by breaking it down into manageable components.
3. Error Handling:
    - The parser uses the Maybe type for error handling, returning Nothing when it encounters unexpected tokens or parsing failures. This ensures that invalid expressions are not processed further.
4. Syntax Tree Construction:
    - As the parser processes the input, it constructs a syntax tree representing the hierarchical structure of the boolean expression.
    - The syntax tree is built incrementally, with each parsing function contributing to its construction.
5. Recursive Descent Type Parsing:
    - Recursive descent type parsing is used extensively to match the input tokens to specific grammar rules and construct the syntax tree.
    - The parser recursively descends through the input, ensuring that the correct precedence and structure of boolean expressions are maintained.
6. Parentheses Handling:
    - The parser can handle expressions enclosed in parentheses, similar to the arithmetic expression parser.
    - The parseParenthesizedExpr function ensures that expressions within parentheses are correctly parsed and included in the syntax tree.
7. Boolean Literals and Comparison Expressions:
    - The parseBasicBoolExpr function handles parsing boolean literals ("True" and "False") as well as basic comparison expressions.
    - It constructs the syntax tree nodes for these basic boolean expressions.
8. "Not" Expressions:
    - The parser recognizes and parses "not" expressions using the parseNotExpr function.
    - It ensures that negated expressions are correctly represented in the syntax tree.
9. Comparison Expressions:
    - The parseEqExpr function handles parsing comparison expressions (e.g., =, <=) and combines them with other boolean expressions.
10. Boolean Operators:
    - The parser extends its capabilities to handle boolean operators, such as "and" and "not".
    - It correctly constructs the syntax tree nodes for boolean conjunctions and negations.
11. Extensibility:
    - The modular design of the parser allows for easy extensibility to support additional boolean and comparison operators or more complex boolean expressions.

```
--BExp Parser

{-
parseParenthesizedExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Parses parenthesized boolean expressions.
Input: A list of string tokens.
Output: A Maybe tuple containing a Bexp for the parsed expression within parentheses and the remaining tokens.
-}
parseParenthesizedExpr :: [String] -> Maybe (Bexp, [String])
parseParenthesizedExpr tokens =
  case parseAndOrBoolEqOrNotOrBasicBoolExpr tokens of
    Just (expr, ")":remainingTokens) -> Just (expr, remainingTokens)
    _ -> Nothing


{-
parseComparisonExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Parses comparison expressions involving == for equality and <= for less than or equal.
Input: A list of string tokens.
Output: A Maybe tuple with a Bexp representing the comparison and the remaining tokens.
-}
parseComparisonExpr :: [String] -> Maybe (Bexp, [String])
parseComparisonExpr tokens = do
  (leftExpr, remainingTokens) <- parseSumSubOrProdOrIntOrParen tokens
  case remainingTokens of
    "==":rest -> parseComparison AEq leftExpr rest
    "<=":rest -> parseComparison BLe leftExpr rest
    _ -> Nothing


{-
parseComparison:
Type: (Aexp -> Aexp -> Bexp) -> Aexp -> [String] -> Maybe (Bexp, [String])

Purpose: Handles the logic for parsing a comparison operation.
Input: A function representing the comparison operation, an Aexp for the left expression, and a list of tokens.
Output: A Maybe tuple with a Bexp for the comparison operation and the remaining tokens.
-}
parseComparison :: (Aexp -> Aexp -> Bexp) -> Aexp -> [String] -> Maybe (Bexp, [String])
parseComparison compOp leftExpr tokens = do
  (rightExpr, tokensAfterRightExpr) <- parseSumSubOrProdOrIntOrParen tokens
  return (compOp leftExpr rightExpr, tokensAfterRightExpr)


{-
parseBasicBoolExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Defines a basic boolean expression parser for literals ("True", "False"), parenthesized expressions, and compariso
Input: A list of string tokens.
Output: A Maybe tuple with a Bexp for the basic boolean expression and the remaining tokens.
-}
parseBasicBoolExpr :: [String] -> Maybe (Bexp, [String])
parseBasicBoolExpr ("True":remainingTokens)  = Just (BTrue, remainingTokens)
parseBasicBoolExpr ("False":remainingTokens) = Just (BFals, remainingTokens)
parseBasicBoolExpr ("(":tokens)              = parseParenthesizedExpr tokens
parseBasicBoolExpr tokens                    = parseComparisonExpr tokens


{-
parseNotOrBasicBoolExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Parses expressions involving the "not" operator or falls back to parsing basic boolean expressions.
Input: A list of string tokens.
Output: A Maybe tuple with a Bexp and the remaining tokens.
-}
parseNotOrBasicBoolExpr :: [String] -> Maybe (Bexp, [String])
parseNotOrBasicBoolExpr tokens = case tokens of
    "not":rest -> parseNotExpr rest
    _ -> parseBasicBoolExpr tokens


{-
parseNotExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Helper function to parse the "not" expression.
Input: A list of string tokens.
Output: A Maybe tuple with a Bexp for the negated expression and the remaining tokens.
-}
parseNotExpr :: [String] -> Maybe (Bexp, [String])
parseNotExpr tokens = do
  (parsedExpr, remainingTokens) <- parseBasicBoolExpr tokens
  Just (BNot parsedExpr, remainingTokens)


{-
parseBoolEqOrNotOrBasicBoolExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Parses expressions involving "equal", "not", or more basic boolean expressions.
Input: A list of string tokens.
```

```
Output: A Maybe tuple with a Bexp and the remaining tokens.
-}
parseBoolEqOrNotOrBasicBoolExpr :: [String] -> Maybe (Bexp, [String])
parseBoolEqOrNotOrBasicBoolExpr tokens = do
  (parsedLeftExpr, remainingTokens) <- parseNotOrBasicBoolExpr tokens
  parseEqExpr parsedLeftExpr remainingTokens


{-
parseEqExpr:
Type: Bexp -> [String] -> Maybe (Bexp, [String])

Purpose: Helper function to parse the "equal" boolean expression.
Input: A Bexp for the left expression and a list of tokens.
Output: A Maybe tuple with a Bexp for the equality expression and the remaining tokens.
-}
parseEqExpr :: Bexp -> [String] -> Maybe (Bexp, [String])
parseEqExpr leftExpr tokens = case tokens of
    "=":rest -> do
      (parsedRightExpr, remainingTokens) <- parseBoolEqOrNotOrBasicBoolExpr rest
      Just (BEqBool leftExpr parsedRightExpr, remainingTokens)
    _ -> Just (leftExpr, tokens)


{-
parseAndOrBoolEqOrNotOrBasicBoolExpr:
Type: [String] -> Maybe (Bexp, [String])

Purpose: Extends the parser to handle expressions involving the "and" operator, building upon the previous parsers.
Input: A list of string tokens.
Output: A Maybe tuple with a Bexp for the combined expression and the remaining tokens.
-}
parseAndOrBoolEqOrNotOrBasicBoolExpr :: [String] -> Maybe (Bexp, [String])
parseAndOrBoolEqOrNotOrBasicBoolExpr tokens = do
  (parsedLeftExpr, remainingTokens) <- parseBoolEqOrNotOrBasicBoolExpr tokens
  parseAndExpr parsedLeftExpr remainingTokens


{-
parseAndExpr:
Type: Bexp -> [String] -> Maybe (Bexp, [String])

Purpose: Helper function to parse the "and" expression.
Input: A Bexp for the left expression and a list of tokens.
Output: A Maybe tuple with a Bexp for the conjunction and the remaining tokens.
-}
parseAndExpr :: Bexp -> [String] -> Maybe (Bexp, [String])
parseAndExpr leftExpr tokens =
  case tokens of
    "and":rest -> do
      (parsedRightExpr, remainingTokens) <- parseAndOrBoolEqOrNotOrBasicBoolExpr rest
      Just (BAnd leftExpr parsedRightExpr, remainingTokens)
    _ -> Just (leftExpr, tokens)
```

Statements Parsing

The Statements Parser is a critical component of the project. The functions provided below are a part of this parser, each serving to interpret a different aspect of the language's syntax.

1. parseParenthesizedStm:
    - This function is designed to handle statements enclosed within parentheses. It's essential for understanding the precedence of operations and properly grouping expressions. By finding the matching closing parenthesis, it ensures that the inner statements are correctly parsed and then continues with the rest of the tokens.
2. findClosingParen:
    - This helper function is crucial for the operation of parseParenthesizedStm. It recursively scans the list of tokens to find the closing parenthesis that matches an opening one, keeping track of nesting to handle complex expressions correctly. It returns the inner tokens (those within the parentheses) and the remaining tokens, allowing further parsing.
3. parseAssignment:
    - Assignment statements are fundamental in any programming language, and this function specifically parses them. It looks for the pattern where a variable is followed by ':=' and an expression, ensuring that the expression is valid and ends appropriately with a semicolon.
4. parseExpression:
    - It's designed to handle the right-hand side of an assignment.
5. splitWhileDo & parseWhileLoop:
    - These functions work together to parse while loop constructs, a common control flow mechanism in programming languages. They identify the condition for the loop and its body, ensuring that the syntax follows the expected pattern.
6. splitIfThenElse & parseIfThenElse:
    - These functions are designed to handle if-then-else constructs, another crucial control flow feature. They split the tokens into the relevant parts of the construct (condition, then branch, else branch) and ensure that each part is parsed correctly.

7. parseElse:
    - This function specifically deals with parsing the 'else' part of an if-then-else construct. It's designed to handle cases where the else block might be empty, a single statement, or a series of statements.
8. parseStm:
    - This function is the heart of the Statements Parser. It dispatches the tokens to the appropriate specific parser based on the current context. It's a recursive function that keeps parsing tokens until there are none left, accumulating the parsed statements as it goes.

Each of these functions contributes to the overall capability of the Statements Parser to understand and translate high-level constructs into a structured format suitable for further processing or execution. The design reflects a recursive descent parsing approach, common in compilers and interpreters for its simplicity and effectiveness in handling a wide variety of programming

```
--Statements Parser

{-
parseParenthesizedStm:
Type: [String] -> [Stm] -> ([Stm], [String])

Purpose: Parses statements within parentheses, managing the extraction and parsing of inner tokens.
Input: A list of string tokens and the current list of accumulated statements.
Output: A tuple with updated accumulated statements and remaining tokens.
-}
parseParenthesizedStm :: [String] -> [Stm] -> ([Stm], [String])
parseParenthesizedStm ("(":tokens) currentStm =
    case findClosingParen tokens 0 of
        Just (innerTokens, remainingTokens) ->
            let parsedInnerStms = parseStm innerTokens []
            in (currentStm ++ parsedInnerStms, tail remainingTokens)  -- tail to skip the closing ")"
        Nothing ->
            error "ParseStm Error: Expected closing ')' but none was found."
parseParenthesizedStm _ _ =
    error "ParseStm Error: Expected opening '('."


{-
findClosingParen:
Type: [String] -> Int -> Maybe ([String], [String])

Purpose: Helper function to find the closing parenthesis for a parenthesized expression.
Input: A list of string tokens and the current depth of nested parentheses.
Output: A tuple with the inner tokens and remaining tokens after the closing parenthesis.
-}
findClosingParen :: [String] -> Int -> Maybe ([String], [String])
findClosingParen tokens depth =
    case tokens of
        [] -> Nothing
        (")":_) | depth == 1 -> Just ([], tokens)
        (")":xs) -> Just (splitAt (length tokens - length xs - 1) tokens)
        ("(":xs) -> findClosingParen xs (depth + 1)
        (x:xs) ->
            let (innerTokens, restTokens) = fromJust (findClosingParen xs depth)
            in Just (x : innerTokens, restTokens)



{-
parseAssignment:
Type: [String] -> [Stm] -> ([Stm], [String])

Purpose: Parses assignment statements, extracting the variable name and the expression to be assigned.
Input: A list of string tokens and the current list of accumulated statements.
Output: A tuple with updated accumulated statements and remaining tokens after processing the assignment.
-}
parseAssignment :: [String] -> [Stm] -> ([Stm], [String])
parseAssignment tokens stmAccumulator =
    case tokens of
        (varName : ":=" : rest) ->
            let (expressionTokens, ";":remainingTokens) = break (== ";") rest
            in case parseSumSubOrProdOrIntOrParen expressionTokens of
                Just (expr, []) ->
                    let newAssignStm = AssignStm varName expr
                    in (stmAccumulator ++ [newAssignStm], remainingTokens)
                Just _ -> error "ParseStm Error: Extra tokens after expression."
                Nothing -> error "ParseStm Error: Invalid expression after ':='."
        _ -> error "ParseStm Error: Expected a variable name followed by ':='."


{-
parseExpression:
Type: String -> [String] -> [Stm] -> ([Stm], [String])

Purpose: Parses expressions, extracting the variable name and the expression to be assigned.
Input: A variable name, a list of string tokens, and the current list of accumulated statements.
Output: A tuple with updated accumulated statements and remaining tokens after processing the expression.
-}
parseExpression :: String -> [String] -> [Stm] -> ([Stm], [String])
parseExpression varName tokens stmAccumulator =
    let (expressionTokens, remainingTokens) = break (== ";") tokens
    in case parseSumSubOrProdOrIntOrParen expressionTokens of
        Just (expr, []) ->
            let newAssignStm = AssignStm varName expr
                (_, rest) = splitAt 1 remainingTokens -- safely discard the ";" which is the first element
```

```
            in (stmAccumulator ++ [newAssignStm], rest)
        Just _ -> error "ParseStm Error: Extra tokens after expression."
        Nothing -> error "ParseStm Error: Invalid expression after ':='."



{-
splitWhileDo:
Type: [String] -> ([String], [String], [String])

Purpose: Helper function to split the tokens for a while loop into the condition and body.
Input: A list of string tokens.
Output: A tuple with the tokens before "while", the condition tokens, and the tokens after "do".
-}
splitWhileDo :: [String] -> ([String], [String], [String])
splitWhileDo tokens =
    let (beforeWhile, rest) = break (== "while") tokens
        (_, afterWhile) = splitAt 1 rest
        (condition, afterDoWithRest) = break (== "do") afterWhile
        (_, afterDo) = splitAt 1 afterDoWithRest
    in (beforeWhile, condition, afterDo)

{-
parseWhileLoop:
Type: [String] -> [Stm] -> ([Stm], [String])

Purpose: Parses while loop constructs by identifying the condition and body of the loop.
Input: A list of string tokens and the current list of accumulated statements.
Output: A tuple with updated accumulated statements and remaining tokens after parsing the while loop.

-}
parseWhileLoop :: [String] -> [Stm] -> ([Stm], [String])
parseWhileLoop tokens currentStm =
    let (beforeWhile, conditionTokensRaw, afterDo) = splitWhileDo tokens
        cleanedConditionTokens = if not (null conditionTokensRaw) && head conditionTokensRaw == "(" then tail (init conditi
        parsedBooleanExpr = parseAndOrBoolEqOrNotOrBasicBoolExpr cleanedConditionTokens
    in case afterDo of
          ("(":_) ->
              case parsedBooleanExpr of
                  Just (boolExpr, [")"]) ->
                      let (whileBodyTokens, restTokens) = span (/= ")") afterDo
                          (_, bodyTokens) = splitAt 1 whileBodyTokens
                          whileBody = parseStm bodyTokens []
                      in (currentStm ++ [WhileStm boolExpr whileBody], tail restTokens)
                  Just (boolExpr, []) ->
                      let (whileBodyTokens, restTokens) = span (/= ")") afterDo
                          (_, bodyTokens) = splitAt 1 whileBodyTokens
                          whileBody = parseStm bodyTokens []
                      in (currentStm ++ [WhileStm boolExpr whileBody], tail restTokens)
                  Nothing -> error "ParseStm Error: Failed to parse boolean expression in 'while' statement."
                  _ -> error "ParseStm Error: Unexpected tokens after parsing boolean expression in 'while' statement."

          (x:_) ->
              case parsedBooleanExpr of
                  Just (boolExpr, [")"]) ->
                      let (whileBodyTokens, restTokens) = span (/= ";") afterDo
                          whileBody = parseStm whileBodyTokens []
                      in (currentStm ++ [WhileStm boolExpr whileBody], tail restTokens)
                  Just (boolExpr, []) ->
                      let (whileBodyTokens, restTokens) = span (/= ";") afterDo
                          whileBody = parseStm whileBodyTokens []
                      in (currentStm ++ [WhileStm boolExpr whileBody], tail restTokens)
                  Nothing -> error "ParseStm Error: Failed to parse boolean expression in 'while' statement."
                  _ -> error "ParseStm Error: Unexpected tokens after parsing boolean expression in 'while' statement."

          [] -> error "ParseStm While Loop Error: No statements after 'do'."


{-
splitIfThenElse:
Type: [String] -> ([String], [String], [String], [String], [String])

Purpose: Helper function to split the tokens for an if-then-else construct into tokens before if, the condition, then-branc
Input: A list of string tokens.
Output: A tuple with the tokens before "if", the condition tokens, the then-branch tokens, the else-branch tokens, and the
-}
splitIfThenElse :: [String] -> ([String], [String], [String], [String], [String])
splitIfThenElse tokens =
    let (beforeIf, rest) = break (== "if") tokens
        (_, afterIf) = splitAt 1 rest
        (condition, afterThenWithRest) = break (== "then") afterIf
        (_, afterThen) = splitAt 1 afterThenWithRest
        (thenBranch, afterElseWithRest) = break (== "else") afterThen
        (_, afterElse) = splitAt 1 afterElseWithRest
        (elseBranchWithoutSemi, outOfConditionStatementsRaw) = breakAtOutCondition afterElse 0
        elseBranch = if not (null elseBranchWithoutSemi) && last elseBranchWithoutSemi /= ";" then elseBranchWithoutSemi ++
        outOfConditionStatements = if not (null outOfConditionStatementsRaw) && head outOfConditionStatementsRaw == ";" the
    in (beforeIf, condition, thenBranch, elseBranch, outOfConditionStatements)


{-
breakAtOutCondition:
Type: [String] -> Int -> ([String], [String])

Purpose: Helper function to break at the end of the 'else' block or at the first ';' outside of parentheses.
Input: A list of string tokens and the current level of nested parentheses.
Output: A tuple with the tokens before the end of the 'else' block and the remaining tokens.
```

```
-}
breakAtOutCondition :: [String] -> Int -> ([String], [String])
breakAtOutCondition [] _ = ([], [])
breakAtOutCondition (t:ts) level =
    case t of
        ";" | level == 0 -> ([], ts)
        "(" -> let (body, rest) = breakAtOutCondition ts (level + 1)
               in (t : body, rest)
        ")" -> let (body, rest) = breakAtOutCondition ts (level - 1)
               in (t : body, rest)
        _ -> let (body, rest) = breakAtOutCondition ts level
             in (t : body, rest)


{-
parseIfThenElse:
Type: [String] -> [Stm] -> ([Stm], [String])

Purpose: Parses if-then-else constructs, identifying and parsing the condition, then-branch, and else-branch.
Input: A list of string tokens and the current list of accumulated statements.
Output: A tuple with updated accumulated statements and remaining tokens after parsing the if-then-else construct.
-}
parseIfThenElse :: [String] -> [Stm] -> ([Stm], [String])
parseIfThenElse tokens currentStm =
    let (beforeIf, conditionTokensRaw, thenBranchTokensRaw, elseBranchTokensRaw, outOfConditionTokens) = splitIfThenElse to
        cleanedConditionTokens = if not (null conditionTokensRaw) && head conditionTokensRaw == "(" then tail (init conditi
        cleanedThenBranchTokens = if not (null thenBranchTokensRaw) && head thenBranchTokensRaw == "(" then tail (init then
        parsedBooleanExpr = parseAndOrBoolEqOrNotOrBasicBoolExpr cleanedConditionTokens
    in case parsedBooleanExpr of
        Just (boolExpr, []) ->
            let thenBody = parseStm cleanedThenBranchTokens []
                (elseBody, _) = parseElse elseBranchTokensRaw  -- No need to use remainingTokens here as it's captured in o
            in (currentStm ++ [IfStm boolExpr thenBody elseBody], outOfConditionTokens)
        Just (boolExpr, _) -> error "ParseStm Error: Unexpected tokens after parsing boolean expression in 'if' statement."
        Nothing -> error "ParseStm Error: Failed to parse boolean expression in 'if' statement."


{-
parseElse:
Type: [String] -> ([Stm], [String])

Purpose: Parses the 'else' block of an if-then-else construct.
Input: A list of string tokens.
Output: A tuple with the parsed statements in the 'else' block and the remaining tokens.
-}
parseElse :: [String] -> ([Stm], [String])
parseElse tokens =
    if null tokens then
        -- Handle case where there are no 'else' statements.
        ([], [])
    else
        case head tokens of
            "(" ->
                -- Handle case where 'else' block is enclosed in parentheses.
                let (bodyTokens, remainingTokens) = span (/= ")") tokens
                    (_, bodyTokensTail) = splitAt 1 bodyTokens
                    body = parseStm bodyTokensTail []
                in (body, tail remainingTokens)
            _ ->
                -- Directly handle 'else' block tokens.
                -- No need to check for semicolons as the tokens are already part of the 'else' block.
                let body = parseStm tokens []
                in (body, [])


{-
parseStm:
Type: [String] -> [Stm] -> [Stm]

Purpose: The main parsing function that dispatches to specific parsers based on the current token, managing the overall par
Input: A list of string tokens and the current list of accumulated statements.
Output: The final list of parsed statements after processing all tokens.
-}
parseStm :: [String] -> [Stm] -> [Stm]
parseStm [] stm = stm

parseStm (";":remainingTokens) accumulatedStatements = parseStm remainingTokens accumulatedStatements

parseStm (varName:":=":tokens) stmAccumulator =
    let (newStms, remainingTokens) = parseAssignment (varName:":=":tokens) stmAccumulator
    in parseStm remainingTokens newStms

parseStm ("(":remainingTokens) currentStm =
    let (newStms, remainingTokensAfterParen) = parseParenthesizedStm ("(":remainingTokens) currentStm
    in parseStm remainingTokensAfterParen newStms

parseStm ("if":remainingTokens) currentStm =
    let (newIfStm, remainingTokensAfterIf) = parseIfThenElse ("if":remainingTokens) currentStm
    in parseStm remainingTokensAfterIf (currentStm ++ newIfStm)


parseStm ("while":tokens) currentStm =
    let (newWhileStm, remainingTokensAfterWhile) = parseWhileLoop ("while":tokens) currentStm
    in parseStm remainingTokensAfterWhile (currentStm ++ newWhileStm)
```

Main Parse Function

1. lexer inputString: The lexer function is called with the inputString as its argument. The lexer's role is to tokenize the input string into a list of strings, where each string represents a token (such as keywords, identifiers, operators, and literals) found in the input program.

2. parseStm (lexer inputString) []: The result of the lexer, which is a list of tokens, is passed to the parseStm function along with an empty list as the initial context. parseStm is the starting point of the parsing process, responsible for parsing statements within the program.

3. The parsed result is returned as a structured representation of the input program, which is of type Program.

```
{-
parse:
Type: String -> Program

Purpose: To parse a program written in the language defined in the assignment.
Input: A String (inputString), which is the text of the program to be parsed.
Output: A Program, which is a structured representation of the input program.
-}
parse :: String -> Program
parse inputString = parseStm (lexer inputString) []
```

Tests

We tested our program with the provided **testParser** function:

```
-- To help you test your parser
testParser :: String -> (String, String)
testParser programCode = (stack2Str stack, state2Str state)
  where (_,stack,state) = run(compile (parse programCode), createEmptyStack, createEmptyState)
```

All the provided tests ran successfully.

```
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")
```