



L-Università ta' Malta
Faculty of Information &
Communication Technology

Compiler Theory & Practice Project

Part 1

Karl Attard 203501(L)
B.Sc. (Hons) Artificial Intelligence

Study-unit: **Compiler Theory and Practice**
Unit Code: **CPS2000**
Lecturer: **Dr Sandro Spina**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

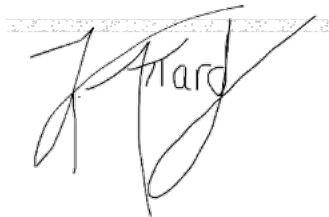
Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines” (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

We, the undersigned, declare that the assignment submitted is our work, except where acknowledged and referenced.

We understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).



Karl Attard

Student Name

Signature

ICS2207

Compiler Theory & Practice Part 1

Course Code

Title of work submitted

07/07/2021

Date

Table of Contents

INTRODUCTION.....	4
TASK 1.....	5
LEXER	5
<i>Implementation</i>	6
TASK 2.....	10
PARSER	10
<i>Implementation</i>	12
TASK 3.....	15
XML GENERATION	15
IMPLEMENTATION	15
TASK 4.....	16
SEMANTIC ANALYSIS	16
IMPLEMENTATION	17
<i>VariableDeclarartionNode</i>	18
<i>IfStatementNode</i>	18
<i>BlockStatNode</i>	18
<i>WhileStatementNode</i>	19
TASK 5.....	20
INTERPRETER OUTPUT	20
IMPLEMENTATION	20
<i>VariableDeclarationNode</i>	20
<i>IfStatementNode</i>	21
<i>ForStatementNode</i>	21
<i>PrintStatNode</i>	21
TESTING	22
LEXER	22
PARSER	22
<i>Case 1</i>	22
<i>Case 2</i>	23
<i>Case 3</i>	23
XML	23
<i>Case 1</i>	23
<i>Case 2</i>	24
SEMANTIC ANALYSIS	25
<i>Case 1</i>	25
<i>Case 2</i>	25
<i>Case 3</i>	25
INTERPRETER	26
<i>Case 1</i>	26
<i>Case 2</i>	26
<i>Case 3</i>	27

Introduction

In the first part of this project, it was required to build a TeaLang programming language from the EBNF rules provided. This process involved the implementation of the Lexer, Parser, XML generation, Semantic analysis, and Interpreter output. In this documentation, the implementation for the tasks mentioned above is discussed.

All source scripts were programmed using Java on the IntelliJ IDE.

Task 1

Lexer

The first task required for the process of compiler TeaLang programming language is Lexical Analysis and a table-driven approach was used. Lexical analysis is the process tokenisation, meaning converting the string program into a list of tokens. Each token consists of two properties, its enum value which holds the type of token and the string lexeme holding the string value of the token.

In order to build a lexer, it was first required to design and build a DFSA. A DFSA includes a start state, transition from one state to another and final state/s. Each final state returns the respective token for the EBNF rule. The DFSA for this project is illustrated in figure 1 below.

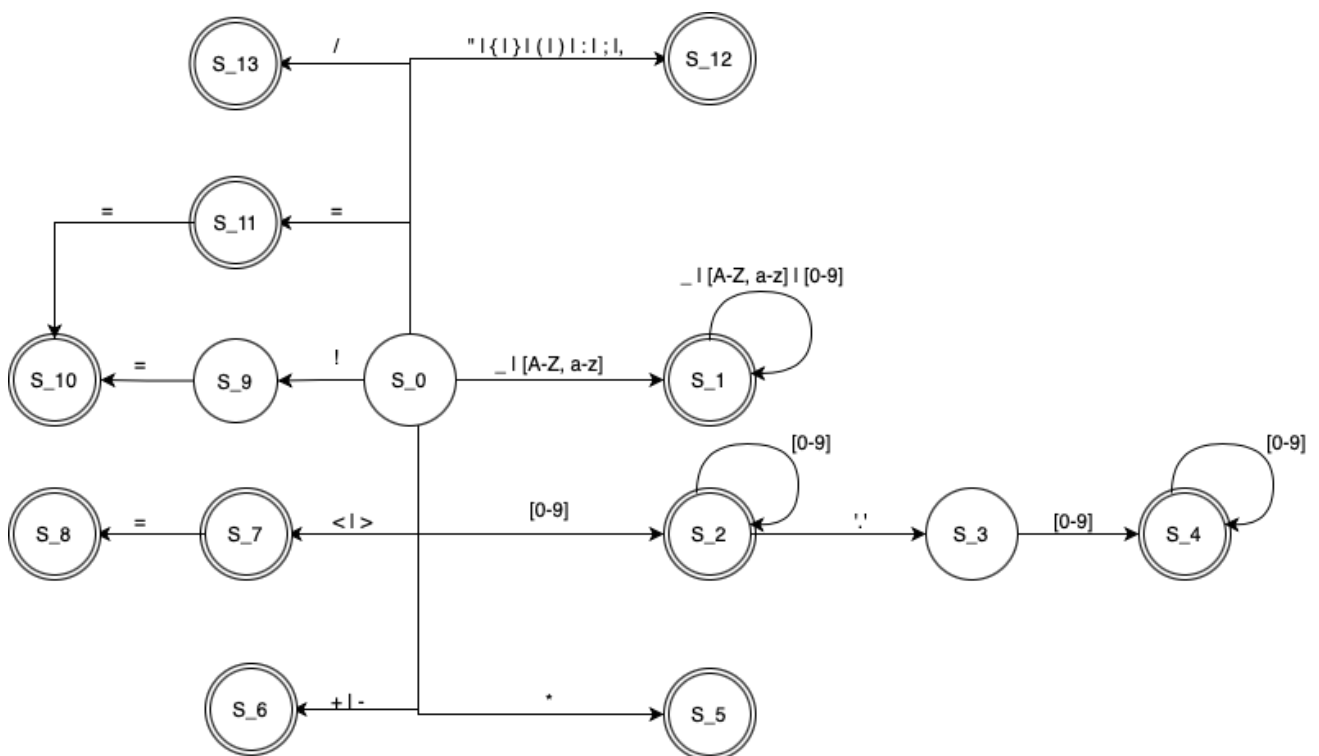


Figure 1: DFSA for Task 1 including all states required for the EBNF.

For example, the following transition diagram is the EBNF rule for `<identifier>`.

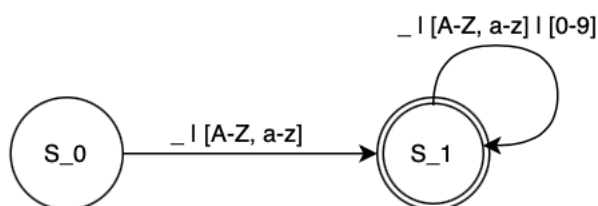


Figure 2: Transition diagram for `<identifier>` EBNF rule.

It is important to note that the DFSA does not handle any in/multi-line comments. These are handled by reading the text file and removing all comments from the file read and then a new text file is written having no comments. This text file will then be passed to the function that perform lexical analysis.

Implementation

After having successfully designed a DFSA, it was then required to implement task 1. In this section, this implementation will be explained, starting from the token class.

Token Class

This is the first class which was created which stores the needed data for each token, its type (stored as an enum) and string lexeme. The following is a list of enums representing each token type created:

- TOKEN_FLOAT_TYPE
- TOKEN_INT_TYPE
- TOKEN_BOOL_TYPE
- TOKEN_STRING_TYPE
- TOKEN_BOOLEAN
- TOKEN_INTEGER
- TOKEN_FLOAT
- TOKEN_IDENTIFIER
- TOKEN_MULTIPLICATIVE
- TOKEN_ADDITIVE
- TOKEN_RELATIONAL
- TOKEN_NOT
- TOKEN_ASSIGNMENT
- TOKEN_LET
- TOKEN_PRINT
- TOKEN_RETURN
- TOKEN_IF
- TOKEN_ELSE
- TOKEN_FOR
- TOKEN_WHILE
- TOKEN_SEMICOLON
- TOKEN_COLON
- TOKEN_LEFT_BRACKET
- TOKEN_RIGHT_BRACKET
- TOKEN_LEFT_BRACE
- TOKEN_RIGHT_BRACE
- TOKEN_PARENTHESIS
- TOKEN_COMMA
- TOKEN_ERROR
- TOKEN_EOF

The above tokens were created for each keyword, one for an identifier, all operators and for special symbols/characters.

ClassifierTable Class

This class was created in order to get the column index of the DFSA table. A static function called 'getColumnType()' was created which returns the mentioned index and takes one parameter, the character read by the lexer for which the column index of this character need to be returned. Therefore, a switch statement on this character is implemented which returns the DFSA transition table column index for this character passed as parameter. If the character passed is not defined as a column in the DFSA, then a negative number is return indicating an error. This function is called within the lexer class which will be described below.

Lexer Class

The first thing which was implemented in this class are the variables which will be used throughout this whole class. The following are the data structures:

- Array List of type Character which is used to store every character read from the text file having no comments.
- A static integer index used to store the index of the last character which was read from the above array list.
- A 2D static integer array which represents the DFSA table in figure 1 above.
- A set of Strings which stores all the keywords in the TeaLang, such as "for", "while" and "let".
- A set of integers which stores the final states (end states in figure 1 above).

Then several functions were created which will read from text file and which contribute to lexical analysis. The following are the list of functions created.

- 'ReadTxtFile()' that takes one parameter – the file to be read.
 - This function will read the text file character by character and adds each character to the array list mentioned above. This function returns exception if file was not found or if empty.
- 'tokenization()' is a function which performs the process of tokenisation, i.e. generates a list of tokens from the text file read.
 - This function will iterate until the 'TOKEN_EOF' is reached – the end of file. During each iteration, it calls the 'scanner()' method which returns the token and then adds it to an array list of type 'Token'.
 - This function returns the array list of type 'Token' which contains all generated tokens.
- 'scanner()' is a function that scans that characters from the array list of type Character and produces a token.
 - This function implemented works in the following way:
 - First, push an error state to the stack of visited states.
 - It is important to note that all preceding white spaces and/or tabs are ignored before entering the below while loop.
 - Now, a while loop iterates until the current state would be an error state.
 - During each iteration, the next character is read. If this character read is a whitespace, then break from this loop. On

the other hand, if the character read is a new line, then continue iterating.

- Then, the algorithm will clear the stack if and only if the current state is a final state.
- Now, the current state is pushed in the stack.
- Finally, get the column index of the current character read by calling the 'getColumnType()' function. Then, pass the returned index to the function 'getNewState()' which transitions to the respective state based on the current state and the column index returned. This process is then repeated until all the characters are read.
- After doing the above, a rollback loop is implemented. This consist of another while loop where the condition will be until the current state is a final state or until the visited stack is empty.
 - In each iteration, a state is popped out of the stack, the string lexeme is truncated (removing last character from string and updates index used to read from array list of type Character) and updated.
- Once this rollback loop has been terminated, if the current state is a final state, then 'setTokenType()' function is called. Otherwise, a lexical error will occur.
- Therefore, in a nutshell, this algorithm will first ignore any trailing whitespaces and/or tabs and the tokenisation will then take place. This will read character by character from the respective array list. The character is concatenated to the lexeme string if and only if it is not a whitespace (exits loop) or new line (begins iteration again). The current state is updated based on the previous state and new character read. Now, if the current state is also a final state, then the stack of visited states is cleared. The current state is then pushed into this stack, and this is repeated until the next character leads either to an error state or until all characters in the array list are read. Then, the rollback loop is performed which then allows us to create the token for that state once finished.
 - This function will return the token created.
- The 'setTokenType()' function creates the token for the current state. This function has 2 parameters: the current state and the string lexeme.
 - Initially, the string lexeme of the token created is set to the latter parameter.
 - Then, a switch statement is implemented on the former parameter. Each case in this switch statement is a final state and a token is created having its type with respect to the final state. However, it is important to note that if the state is either 1 or 12, i.e., states for identifier or symbols, then another switch statement must be implemented in each of the above states. This is because in state 1, the identifier can be a predefined keyword or a predefined character symbol, hence, different token types must be assigned to each of these cases. For example, suppose the state is 1 and lexeme it "let", then the token type must be 'TOKEN_LET' and not 'TOKEN_IDENTIFIER' since this is a keyword.

Therefore, the above functionalities contribute to the process of lexical analysis which will return an array list of type Token. This array list will be used by the parser in the next task.

Task 2

Parser

After having successfully completed the lexical analysis stage, it is now time to ensure that the TeaLang program is syntactically correct. This is achieved by using the so-called parser, which iterates over the list of tokens returned by the lexer. If a successful parse is done, then an Abstract Syntax Tree (AST) is generated. Hence, prior to building the parser, it was imperative to design all the nodes which will be used to build the AST. A node was created for every statement and factor defined in the EBNF rules for the TeaLang.

The following is the Class Diagram representation for the AST:

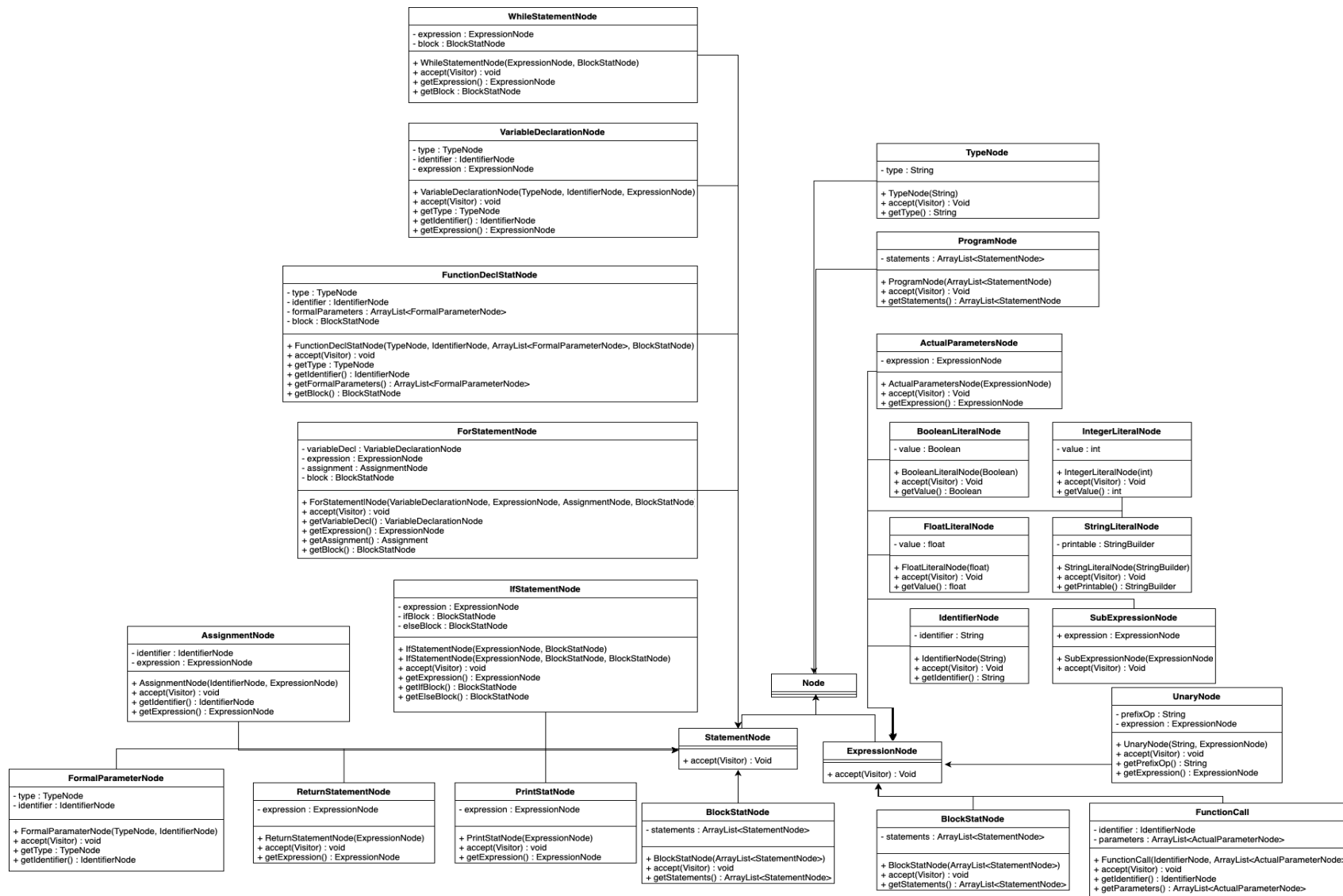


Figure 3: Class Diagram for AST

As illustrated in figure 3 above, there a class for every node was created. Moreover, object-oriented techniques were used, ranging from a mix of inheritance, composition and encapsulation to describe nodes. The 'Node' class is the parent of all nodes and has 4 distinct children: 'StatementNode', 'ExpressionNode', 'TypeNode' and 'ProgramNode'. Hence, all statements described in EBNF inherit from the 'StatementNode', all expression inherit from 'ExpressionNode' whilst also using composition to create objects having other nodes within this node.

For instance, the 'PrintStatNode' extends from the 'StatementNode' and has only one property, the 'ExpressionNode' which is used to describe the expression. In a print statement, this expression describes what will be outputted to the user.

Moreover, it is well worth noting that all nodes have an exact 'accept(Visitor)' method, which will be explained later on as it is required for tasks 3-5.

Implementation

After having built the AST nodes, it is now time to implement the parser. Therefore, a Parser class was created which handles this process and returns a valid AST if TeaLang program is syntactically correct. This class has 2 properties, the current token and next token and these will be explained later on.

The parser process works in the following way:

- The first method which is called is 'parseProgram()' which takes in one parameter, the array list of type 'Token', i.e., the list of tokens returned by the lexer.
- Moreover, another function created which is used throughout the parsing process is 'getNextToken()' which updates the current token to the next one in line and also updates the next token to the following one.
- Now, the 'parseProgram()' function works in the following manner:
 - It creates an array list of type 'StatementNode'.
 - Then, it iterates until the 'TOKEN_EOF' is reached. During each iteration, it will parse the <statement> rule and subsequently, adds it to the above-mentioned array list.
 - Finally, it returns the created 'ProgramNode' with the list of statements parsed in the iteration/s.
- Parsing then continues in the 'parseStatement()' function which parses the <statement> rule as defined in the EBNF. This function will essentially select the appropriate parse method according to the current token and returns a 'StatementNode'. This is implemented through a switch statement on the current token type. For example, suppose the current token type is 'TOKEN_LET', then this means that this is a variable declaration, hence the function responsible to parse a variable declaration is called. On the other hand, if an invalid token is received, then a parser error is outputted. It is important to note that after the appropriate parse method is called, the parser check if the statement end with a semicolon, if not, an error is displayed.
 - For example, suppose the 'parseVariableDeclaration()' method is called. This means that the current token found was 'TOKEN_LET'. Now, the tokens are updated to get the next token. The next token must be an identifier, and hence, the method responsible to parse identifiers is called which returns an

IdentifierNode. Then, tokens are updated again, and the next token must be a `':'`. If not, a parser error is outputted. Tokens are then updated and then `<type>` must be parsed. Hence, the `'parseType()'` function will determine the type of the node based on the current token. This will return a `TypeNode`. Tokens are once again updated, and the next token must be a `'='` operator. If not, a syntax error is displayed, otherwise tokens are updated again. The algorithm will then parse the `<expression>` by calling the `'parseExpression()'` method which returns an `ExpressionNode`. Finally, the `'parseVariableDeclaration()'` terminates by returning a `VariableDeclarationNode` which stores the `IdentifierNode`, `TypeNode` and `ExpressionNode`.

- The above process is done for the other EBNF rules described by TeaLang.
- The `'parseExpression()'` function is responsible to parse `<expression>`. This method contains 3 distinct methods in order to parse expression – as defined by the EBNF. Therefore, when `'parseExpression()'` is called, it in turn calls the `'parseSimpleExpression()'` method which then calls the `'parseTerm()'` method and finally calls the `'parseFactor()'` method. In the `'parseExpression()'` method, after the first `<Expression>` has been parsed, the next token is checked to see if it is a relational operator token. If this is the case, then according to the EBNF rule, another expression is needed. Hence, `'parseExpression()'` is recalled to parse the next expression, which in turn recalls the above mentioned algorithm again. The same reasoning is done within the `'parseFactor()'` function where this function is recalled if and only if the next token is a multiplication operator. (also same for `'parseSimpleExpression()'` if next token is an additive operator). Figure 4 illustrated below show the `'parseFactor()'` method implementation.

```
// <Factor> ::= <Literal> | <Identifier> | <FunctionCall> | <SubExpression> | <Unary>
private ExpressionNode parseFactor(ArrayList<Token> tokens) {

    getNextToken(tokens);
    switch (currentToken.tokenType) {

        case TOKEN_FLOAT:
        case TOKEN_INTEGER:
        case TOKEN_BOOLEAN:
        case TOKEN_PARENTHESIS:
            return parseLiteral(tokens, currentToken.tokenType);

        case TOKEN_IDENTIFIER:

            // If next token is left bracket token, then this is a <FunctionCall>. Otherwise, it is just an <Identifier>.
            if (nextToken.tokenType == tokenTypes.TOKEN_LEFT_BRACKET) {
                return parseFunctionCall(tokens);
            }
            return new IdentifierNode(currentToken.getLexeme());

        case TOKEN_LEFT_BRACKET:
            // This is a <Subexpression>

            return parseSubExpression(tokens);

        case TOKEN_ADDITIVE:
        case TOKEN_NOT:
            // This is a <Unary>
            return parseUnary(tokens);

        default:
            System.out.println("Incorrect token type!");
            System.exit( status: 1);
    }
    return null;
}
```

Figure 4: `parseFactor()` method implementation

- As can be seen in figure 4, the 'parseFactor()' method will return the appropriate AST node based on the current token. For instance, if token is a literal, then the appropriate literal Node is returned, whilst if token is an identifier, first it is required to check if the next token is a left bracket or not. If it is, then this is a function call and not an identifier. Moreover, if current token is a left bracket, then this means that a <subexpression> need to be parsed whilst if a '-' or 'not' is found, then parse <unary>.

It is important to note that not all parse function created are explained, however, parsing is essentially a series of function calls to parse the EBNF rules having switch and if-else to check what is the token type and to check for non-terminal symbols respectively. The parser will finally return the AST containing the appropriate nodes.

Task 3

XML Generation

In this part of the assignment, it was required to generate XML on the AST returned by the parser. In order to do this task, it was first required to build a Visitor interface class which is able to build the AST produced by the parser in XML format. This interface consists of 'visit()' functions for each of the nodes in the AST and also an 'accept()' method was implemented within each class of the node (as aforementioned). After creating this interface, another class was created called XMLVisitor which implements the Visitor class. Within the XMLVisitor, all methods are overridden and implemented in order to output XML format of the AST. It is important to note that in this task, no actual XML file was created but XML was outputted to the user only.

Implementation

As previously mentioned, the XMLVisitor has a visit method for each type of node because it implements the Visitor class, hence, all function must be overridden. Each visit method in this XMLVisitor class prints the appropriate XML tags to the console according to the node's type.

For example, figure 5 below is the visit method for a variable declaration statement node.

```
@Override
public void visit(VariableDeclarationNode variableDeclarationNode)
{
    System.out.println("<VarDeclaration>");
    incrementIndent();
    System.out.println("<Var Type=\"\" + variableDeclarationNode.getType().getType() + \"\">");
    incrementIndent();
    variableDeclarationNode.getIdentifier().accept( v: this);
    decrementIndent();
    System.out.println("</Var>");
    printIndents();
    variableDeclarationNode.getExpression().accept( v: this);
    decrementIndent();
    System.out.println("</VarDeclaration>");
}
```

Figure 5: Visit method for VariableDeclarationNode.

Figure 5 illustrated above shows the implementation of the visit method for variable declarations. This prints the appropriate XML tags for variable declaration that outline the type of variable, the variable identifier, and its value. It is important to note that 'incrementIndent()', 'decrementIndent()' and 'printIndents()' are functions responsible to keep track of the indentation level when printing the AST's XML tags to console.

All the other overridden methods follow the same reasoning and logic where the correct XML tags are printed out based on the AST node.

Task 4

Semantic Analysis

After having performed lexical analysis and parsing, now it is time to do semantic analysis. Semantic Analysis is the process of ensuring that the declarations and statements of a program are semantically correct. This means that in semantic analysis, meaning to the program is made. Hence, the aim of this task is to perform type and declaration checking. This is achieved by creating another class called SemanticAnalysisVisitor which implements the Visitor class. This class includes overridden visit method for all the nodes and a symbol table to keep track of the variable and function declared together with their respective types. Moreover, every scope has a separate symbol table to differentiate which variables have been declared in different scopes.

It is important to note that in this task, additional rules were added, these being:

1. Function overloading
 - This is polymorphism where several methods having the same identifier, but different parameters are allowed.
 - *NB: This is a task which was required particularly in Part 2 of this assignment; however, it was also decided to include it in Part 1 too.*
 2. Float & int literals
 - The semantic analyser built does not cope with expression having float and int types mixed. This means that a any expression having int and float types together will yield to a semantic error. Figure 6 below is an example of this rule which is added.
- ```
let example : int = 2+1.0;
```
- Figure 6: Semantic Analyser will output error for this variable declaration due to this rule.
- Therefore, to fix the above issue, '1.0' must be change to int, thus, '1'.
3. Functions and their returns
    - When performing semantic analysis on function declarations, a return statement matching the type of the function must need to be present. This means that, if a function does not have return statement, then an error is outputted. Moreover, if the function has a return statement but its type differs to the function's type, then an error is also outputted.

Figure 7 below shows an example of what is required in function declarations for this rule to be obeyed:

```
bool test(x:int)
{
 print "Hello world";
 if(x == 2)
 {
 return true;
 }
 return false;
}
```

Figure 7: Valid Example of this rule



- If the last return statement is omitted from Figure 7's function declaration block, then the semantic analysis outputs an error.

#### 4. Variable shadowing.

- This rule means that variables with the same identifier having same or different types can be redeclared multiple times in different scopes. This means that variables declared within scopes shadow variables declared within their parent scope. However, once the scopes have been processed, the variable declaration within the parent scope will then still take precedence.

For example, figure 8 below outlines an example with outputs in comments:

```
let i : int = 3;
print i; //3

{
 print i; //3
 let i : string = "hello world";
 print i; //hello world
}
print i; //3
```

Figure 8: Example for variable shadowing with outputs in comments.

## Implementation

To build this implementation, it was first required to create a class to define scopes in order to handle variables and functions in a symbol table for different scopes. Moreover, this class also provides methods such as insert and look up. The former allows us to add variables and function bindings to the symbol table whilst the latter allows us to search for a variable or a function in the symbol table of that respective scope.

Hash maps are used to store information about these variables and functions:

- A hash map is used to hold variables.
  - Key: variable identifier
  - Value: type of variable
- A hash map is used to store function bindings.
  - Key: tuple of function identifier and list of parameter types.
  - Values: return type of function.

After having implemented the Scope class, it was then time to start building the semantic analyzer. Hence, a class called SemanticAnalysisVisitor was created which implements the Visitor class. Therefore, all methods found within the Visitor class must be overridden and subsequently implemented to perform semantic analysis on different nodes. Moreover, this class includes several data structures and functions that contribute to successful semantic analysis.

The following are all the data structures created and used:

- A stack of Scopes – holds all scopes in program which are pushed into this stack
- A variable of type Scope which stores the current scope being analysed
- A variable of type String that holds the type of the expression currently being analysed.
- A Boolean flag used to know when a new scope needs to be pushed.
- A stack of type CheckFunctionType – this is a class that stores the data of a function which still needs to be verified, i.e., return type matched function type. A stack is used in this regard to easily keep track of the functions.

Moreover, several functions were also implemented for this task. These are:

- A 'push()' method – pushes a new Scope to the stack that holds all scopes.
- A 'pop()' method – pops a Scope from the stack of all scopes.
- A 'checkForReturn()' – this function checks for a return statement.
- A 'checkFunctionReturnType()' – this function is used to check if the return statement type is the same as the function type.
- A 'getIdentifierType()' – this function takes in an identifier and sets the expression type to the type of the identifier.

Since the visit methods follow almost the same reasoning as the visit methods in the XMLVisitor, it is decided to explain some of them rather than all of them.

#### VariableDeclarationNode

In the visit method of this AST node, first it is required to get the last scope which was pushed into the stack. Then, get the identifier of this AST node and check if there already exists a variable with the same identifier. If yes, then output an error since a variable with the same identifier already exists. Otherwise, get the expression of the AST node and call its accept method. The algorithm will then check if the value of the variable identifier is the same type as the current expression type being analysed by the sentiment analyzer. If incompatible, an error is outputted but if in conformity, then this identifier and its value are added to the respective hash map.

#### IfStatementNode

The visit method for this implementation first accepts the expression (condition) and then checks if this expression is Boolean or not. If not, then a semantic analysis error is outputted since the condition must be Boolean. Then, the if-block of statements are accepted and finally, checking if the if-statement also contains an else-block. If in the affirmative, the sentiment analysis class is also passed to the else block.

#### BlockStatNode

This visit method pushes a new scope to the stack of scopes if and only if the flag is set to false. This is because if the flag is true, then this means that a new scope has already been pushed and thus, does not need to be pushed. Then, it loops through all the statement in this block and passes the semantic analysis class to them. As soon as the analysis of all the

statements has been done, a scope from the stack is popped out (since it has been successfully analysed).

#### WhileStatementNode

This visit method begins by accepting the expression (calling its 'accept()' method). Then, it checks if the expression type is Boolean since conditional expressions must be Boolean. If not, an error is outputted to the user. The visit method will then accept the block since every while must have a block of statements.

## Task 5

### Interpreter Output

The last and finally task for this part of the project was to develop and implement an interpreter. The objective of this task was to build an interpreter for TeaLang so that the code can be run and output is produced to the user. Therefore, another visitor class called `InterpreterExecVisitor` also having another `InterpreterScope` class. These classes are similar to the `SemanticAnalysisVisitor` in implementation but this time, the `Scope` class needs to store the value of variables and not just their type.

### Implementation

As mentioned, in this task it is required to store not only the type of the variable but its value. Therefore, a class called `TypeValueOfVar` was created which is used to store both the type and value of a variable. Moreover, this class makes use of generic type programming for variable's type to be assigned when instantiating.

The following are the data structures used in this `InterpreterScope` class:

- A hash map to hold variables
  - Key: the identifier
  - Value: `TypeValueOfVar` class that holds the type and value.
- A hash map to hold functions
  - Key: a tuple of identifier and function type
  - Value: `FunctionDeclStatNode` that holds the respective properties of that function.

Furthermore, other functions were implemented that insert a variable, function and get variables and functions.

On the other hand, with regards to data structures for the `InterpreterExecVisitor`, they are the exact same as the semantic analysis but this time another Boolean flag was added. Other further changes and adaptations included that the variable '`expressionType`' type was changed to `TypeValueOfVar` (value is also stored apart from type). Furthermore, all visit methods are overridden and implemented together with a '`getIdentifierType()`' method which returns the value of that identifier type.

The following are the Visit methods chosen to be explained (others follow the same reasoning pattern):

### `VariableDeclarationNode`

This visitor method works by first getting the last `InterpreterScope` which was pushed into the stack and then getting the identifier string of this variable. Then, the `InterpreterExecVisitor` class is passed to the Expression node and finally, this method adds this variable into the stack including its value.

### IfStatementNode

This visitor function works by evaluating the condition expression. If evaluation is true, then the if block is processed. Otherwise, the else block is.

### ForStatementNode

This visitor function begins by pushing a new scope into the stack of all scopes. This is done because if the for loop contains a variable declaration, then this must be considered in the block of the for loop and not outside of it. The respective flag responsible for managing the pushing of block scoped is initialised to true so that when the InterpreterExecVisitor class is passed to the block, another scope is not pushed. Furthermore, the Boolean flag responsible to manage the pops from scopes is then also set to true so that if a variable inside this loop is needed, it will not be omitted/disregarded. Hence, a scope is only popped when the loop ends and the for loop works in principle by looping through a while loop until the expression remain true. This condition is evaluated at each iteration of the loop.

While loops are interpreted similarly to for loops.

### PrintStatNode

This visitor function works by first passing the InterpreterExecVisitor class to the expression to be computed, and then printing it out to the user. This function in essence is used to print out the value of expressions in a print statement.

## Testing

For this project, white box testing was used. This testing technique involves dry running the program and ensuring that the expected output is the actual output. In this testing, the tester knows how the program functionalities work and what the output should be. These are portrayed in the screenshots below.

### Lexer

The lexer was tested by writing sample programs having lexical errors and no errors.

#### Case 1

The following program sample was written for this use-case.

```
let i : bool = ! true;
```

An error was outputted by the lexer since the '!' symbol is not defined in the TeaLang EBNF rules. Thus, the expected output matches the actual output and hence, the test for this use case has passed.

```
Incorrect token type!
```

#### Case 2

The following program snippet generates another error:

```
print add();
```

A lexical analysis error was outputted since the symbol '[' is not defined in the EBNF, leading to an error. Subsequently, the expected output matches the actual output and thus, this test passed.

### Parser

Testing the parser was done the same as the lexer.

#### Case 1

The following the program sample:

```
let i ; int = 3;
```

The following is the parser error generated:

```
Syntax Error!
Parser expected a ':' but got a ';'.'
HINT: A ':' must separate a variable name and its type (ex: variableName : int).
```

The expected output for this case is the actual output. This case has successfully passed.

## Case 2

The following is the program sample for this case:

```
print 5
```

A syntactical error was outputted since every statement must end with a ';'. Therefore, the expected output in this case is also the actual output, thus, test passed.

```
Syntax Error!
Parser expected ';'.
A print statement must end with a ';'
```

## Case 3

The following is another program sample example with errors:

```
int add(x:int,y:int)

return x+y;
}
```

A syntax error was generated since an opening brace is omitted. The test has passed since the expected output matched the actual output.

```
Syntax Error!
Parser expected '{' but got 'return'
HINT: A '{' must be present before a block of statement/s
```

## XML

The same pattern as above will be used but this time, showing successfully generations of XML.

## Case 1

The first use case involves the below snippet:

```
print 5;
```

The following XML was generated:

```

----- XML Output -----
<Program>
 <Print>
 <IntegerLiteral>5</IntegerLiteral>
 </Print>
</Program>

```

As can be seen, the XML generated is good, thus, this test has passed for this use case.

## Case 2

The following is the program sample for this use case:

```

while(i < 3){
 print i;
 i = i+1;
}

```

The XML generated for the above snippet is the following:

```

----- XML Output -----
<Program>
 <While>
 <Condition>
 <ExpressionV2 Operator="<">
 <Identifier>i</Identifier>
 <IntegerLiteral>3</IntegerLiteral>
 </ExpressionV2>
 </Condition>
 <Block>
 <Print>
 <Identifier>i</Identifier>
 </Print>
 <VarAssign>
 <Identifier>i</Identifier>
 <ExpressionV2 Operator="+">
 <Identifier>i</Identifier>
 <IntegerLiteral>1</IntegerLiteral>
 </ExpressionV2>
 </VarAssign>
 </Block>
 </While>
</Program>

```

Thus use case passed since the expected output is the same as the actual output, thus successful generation of XML.



## Semantic Analysis

In this part of the testing phase, programs breaking rules were created.

### Case 1

The following program was written:

```
let i : string = 2;
```

It can be noted that although the syntax is correct, semantically is incorrect. This is because the assignment is integer, but the variable declaration type is string. The program outputs the following error:

```
Error found by Semantic Analyser.
Incompatible types found. Compiler expected 'string' but got 'int'
```

The expected output matches the actual output leading to a successful use case test.

### Case 2

The following snippet program was written:

```
int add(x:int, y:float)
{
 return x+y;
}
```

The TeaLang compiler outputs a semantic error because mismatched types cannot be performed in the same expression.

```
Error found by Semantic Analyser.
Mismatched types in expression. Cannot perform operation on 'int' and 'float'
```

Therefore, this test case has also passed since the correct output error is shown.

### Case 3

The snippet program is shown below:

```
let x : int = 1;
let y : int = 6;

print add(x, y);
```

As can be noted, the function 'add()' was never defined after being called, thus an error should be displayed. The following is the error shown to the user:

```
Error found by Semantic Analyser.
No function with an identifier 'add' and parameters with types: int, int, for function call
```

The correct error is shown to the user, and thus, the test case is successfully completed.

## Interpreter

In this part, I will be testing sound programs in order to see that the respective outputs are printed to the screen.

### Case 1

The following snippet is programmed:

```
let x : int = 1;
print x;
```

A variable declaration of type int having value of '1' is declared. The print statement is used to print the value of the variable. The following output is shown:

```
----- Interpreter Output -----
1
```

Hence, the correct output is shown, meaning successful test.

### Case 2

The following is the snippet of code:

```
for(let x:int = 1; x <=5; x=x+1){
 if(x==5){
 print "End of loop";
 }else{
 print x;
 }
}
```

In the above snippet, a for loop is declared, and through each iteration, and if-else block is executing. When the value of 'x' becomes 5, a string is outputted to the user. Otherwise, the value of 'x' is printed. The following is the output generated:

```
----- Interpreter Output -----
1
2
3
4
End of loop
```

### Case 3

The following is the program being tested:

```
let x : float = 1.5+2.5;
print x;

//x = 12.3

print x;

/*
x=2.0+1.0;
*/

print x;
```

In the above implementation, the main objective is to test inline and multi-line comments. The following is the output generated:

```
----- Interpreter Output -----
4.0
4.0
4.0

TeaLang program successfully executed!
```

It can be noted that every comment was ignored since no assignment took place, and the initial value of the variable declaration remained unchanged. Therefore, this case passed the test.

It is important to note that there is a bug when a function call takes place. This error only happens in task 5, i.e., in the interpreter but works perfectly fine in the other tasks.

In conclusion, all tasks required were implemented and work as expected. The only bug known is the function calls in the interpreter only.

To run this project, either build all packages found in the 'src' folder and then running the 'Main.java', or else, open the project folder in IntelliJ and run it there.