

# Compiler Theory and Practice

## Course Assignment 2017/2018

CPS2000

Dylan Galea | 84296M | 21/04/2018

# Table of Contents

## Contents

Table of Contents .....	1
Introduction .....	2
Task 1: A table-driven lexer in C++ .....	3
Implementation and Design of the Lexer .....	3
Test Cases and Results: .....	7
Task 2: Hand-Crafted Recursive Descent Parser. ....	11
Implementation and Design of the Parser .....	11
Test Cases and Results .....	15
Task3: Generate XML of AST .....	21
Implementation and Design .....	21
Test Cases and Results: .....	24
Task4: Semantic Analysis Pass .....	30
Implementation and Design .....	30
Test Cases and Results .....	34
Task 5: Interpreter Execution Pass .....	37
Implementation and Design .....	37
Test Cases and Results .....	39
TASK 6: The REPL .....	42
Implementation and Design .....	42
Test Cases and Results .....	45
Conclusion.....	49
References .....	50

## Introduction

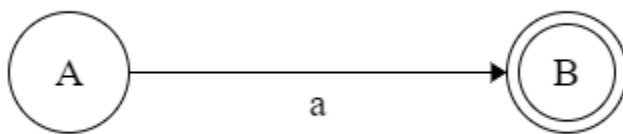
The aim of this assignment was to build the front-end of the compiler for the MINILANG language specified in the assignment specification sheet. A compiler is a program that converts an input called the source program written in some programming language, into a semantically equivalent program that the computer can understand. However, the front—end of the compiler includes only the analysis part of the compilation, and this includes lexical analysis, parsing and semantic analysis. Lexical analysis is the process of taking the source program written in a high-level language and producing a set of tokens that represent the program. These tokens are then passed one by one to the parser where the parser evaluates the stream of tokens and tries to re-structure back the program, whose structure was lost when tokenizing. If there is a syntax error in the program the parser would then detect it and report the error to the programmer. On correct syntax, the parser would then produce an Abstract Syntax Tree representing the structure of the program. This AST is then semantically analyzed in order to check that the program has a valid meaning. For example, the statement `int a = "hey"` is a syntactically valid statement, however semantically it is incorrect because the type of the right-hand side is not the same as the type of the variable on the left-hand side. Apart from type checking, semantic analysis involves also checking for scopes and array bounds.[1], [2], [3]

This report is divided into 6 main sections, where 1 section is dedicated to each independent task identified from the assignment specification sheet. Each main section is further subdivided into 2 sub-sections where one of the sub-sections describes how the task was designed, implemented and any variations from the specification sheet or the EBNF. The other sub section describes how the task was tested and its test results, these tests and any code used will be displayed in screen shots, since these had to be removed because a REPL process had to be designed later in task 6. It is also important to note that for any specific implementation details, dioxygen comments were constructed in order to explain better what is happening in the code. Also, a README file in the attached source code was created to show the user how to compile and run the executable. The next section describes the design, implementation, variations from the specification sheet and test results of the Lexer.

## Task 1: A table-driven lexer in C++

### Implementation and Design of the Lexer

For this task a table-driven lexer had to be implemented. In a table-driven lexer a finite automaton represents a set of strings that are acceptable in the language were the transition from one state to the next is encoded in a look-up table in the program, see [4]. For example, in the automaton in figure 1 below, the Lexer would start from state A and move to state B if the next character in the source input program is 'a'. The Lexer determines to move from state A to state B using the hard-coded look-up table. If during the Lexing process a character different from 'a' is read from the source program, then the lexer must produce a lexing error since there is no state transition that involves any other character and thus the compilation fails. If, however the lexer successfully reads the character 'a' and moves to state B, then since state B is a final state a token will be produced whenever the Lexer stops scanning. Also depending on which final state the lexer passed from last determines what type of token or attributes in the token are passed to the next stage (Note this is just an easy example).



*Figure 1-Automaton example*

Thus prior to start coding, a finite state automaton had to be designed in order to determine the strings and elements that constitute the language. The finite state automaton for this implementation can be seen in figure 2 below (note the starting state is state S0):

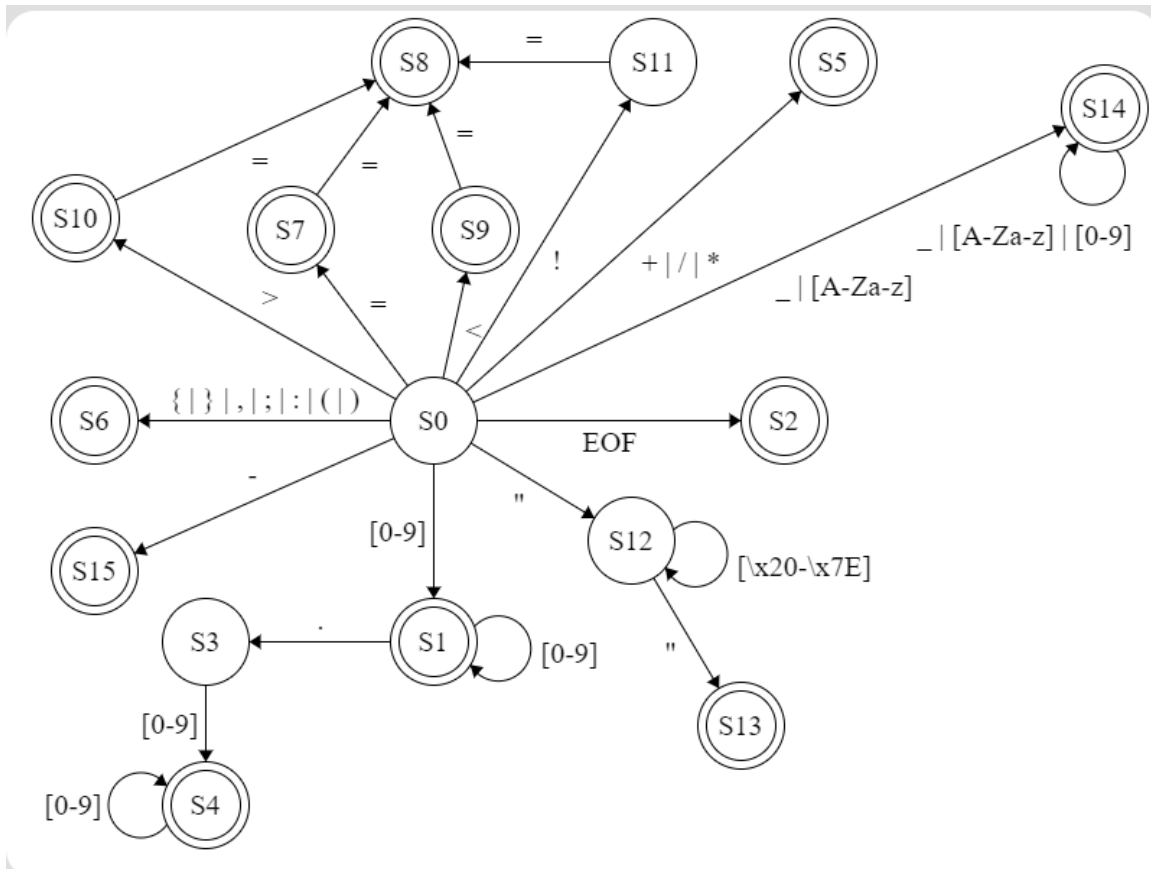


Figure 2- Finite State automaton for language

Thus, following the finite state automaton designed in figure 2 above, first an Enum class called state was created which encoded all the states in figure 2 above in the program, so that the lexer can keep track in which states he is in. The enum class can be seen in figure 3 below:

```
namespace Lexer{
    enum class State{
        S0,
        S1,
        S2,
        S3,
        S4,
        S5,
        S6,
        S7,
        S8,
        S9,
        S10,
        S11,
        S12,
        S13,
        S14,
        S15,
        SE,
        BAD
    };
}
```

Figure 3-Enum class state

Thus, as it can be seen in figure 3 above, a state was created for each state in the automaton. In addition to these states the state SE and BAD were created so that when the lexer is in some state, and a transition function that moves the lexer from the current state to another state does not exist then this means that the lexer encountered a lexical error. For example, in figure 2 above, if the lexer is in state S0 and the lexer scans the input # then state SE is returned, since there exists no transition function that takes that input to another state.

The next step was to design the Tokens that would be relevant to the programming language. In this implementation a token was designed as a class, where each token object has a number of attributes. These attributes were:

- Token Type -> A value of type enum class TokenType were 11 types were designed
- Token\_name -> The name of the token
- Token value -> The numeric representation of the lexeme that created the token
- Token character value-> the character representation of the lexeme that created the token
- Token string value -> the lexeme that created the token

These attributes were important to design so that in the parser, any information related to that token could be easily extracted. For example, if a token of type TOK\_PUNCTUATION is created, if the lexeme is not stored in the token class, the parser could not determine what type of punctuation was scanned by the lexer.

The following table shows the different token types that were designed in this implementation:

Token Type	Values Represented by this Type	State that created this token
TOK_NUMBER	Real numbers	S1, S4
TOK_ARITHMETIC_OPERATOR	+   *   /	S5
TOK_MINUS	-	S15
TOK_EOF	EOF	S2
TOK_PUNCTUATION	{   }   (   )   :   ;   ,	S6
TOK_EQUALS	=	S7
TOK_RELATIONAL_OPERATOR	<   >   >=   <=   ==   !=	S8, S9, S10, S11
TOK_STRING_LITERAL	Any strings enclosed in ""	S13
TOK_IDENTIFIER	Any string starting with _ or a character (not digit)	S14
TOK_KEYWORD	Any keyword of MINILANG	S14
TOK_INVALID	Any remaining strings not in the language	Any invalid transition during lexical analysis

Figure 4- Different token types

As seen in figure 4 above the keywords were handled differently in this implementation. Whenever the lexer is in state S0 and encounters a character, he moves to state S14 in trying to match an identifier token. However, when scanning stops if the matched lexeme is a keyword in MINILANG, this is not matched as an identifier token, but as a keyword token. The list of keywords of MINILANG were defined in a class that contains a field of type `set<string>` containing all keywords of the language. The keywords include, `real`, `int`, `bool`, `string`, `true`, `false`, `and`, `or`, `not`, `set`, `var`, `print`, `return`, `if`, `else`, `while`, `def`. From the above table it could also be noted that there is a separation between the minus token and the rest of the arithmetic operator tokens. This was done in this way because the according to the EBNF the minus operator could be used as a negation operator.

After the above designs were made, the lexer was next to be implemented. The implementation of the lexer was wrapped in the `getNextToken()` function, where whenever this method is called either a valid token is returned or an exception with an error message is returned. Note that in this implementation whenever a lexical error occurs, the line number is given. The essential parts of the `getNextToken` function can be seen in figure 5 below (note the actual code has the specific comments):

```

1  Lexer::Token* Lexer::LexerImplementation::getNextToken() {
2      start from state S0
3      stack.push(State::BAD);
4      while(current_state != State::SE){
5          char next_character = get next character from source file.
6          if (next_char == ' ' || next_character == '\t' and we are not in a string lite:
7              skip
8          }else if(next_char == '\n'){
9              lineNumber++;
10             go to next char
11         }
12         if we encounter comments , skip untill the end of comment
13         lexeme += next_character;
14         current_state = transitionFunction(current_state,next_character);
15         if(checkIfFinalState()){
16             while(!stack.empty()){
17                 stack.pop();
18             }
19         }
20         stack.push(current_state);
21     }
22     while(!checkIfFinalState() && current_state != State::BAD){
23         stack.pop();
24         lexeme.pop_back();
25         current_state = stack.top();
26     }
27     if(checkIfFinalState()){
28         Token *token = new Token(lexeme,current_state);
29         return token;
30     }else{
31         throw CompilingErrorException("syntax error in line "+to_string(lineNumber));
32     }
33 }

```

Figure 5-Lexer Pseudocode

Thus, as seen in figure 5 line 2 above, first the scanning process starts from state S0. As seen in line number 3, then the bad state is pushed so that if no final state is met and scanning stops, this means we got a lexical error. As seen in lines 5 the next character is then read. If this next character is a space or a tab and the current lexeme is empty, this means that the white space or tab is not a relevant token, thus it is skipped. However, if the lexeme is not empty this means only one thing, that the lexer is currently scanning a string literal, thus the white space is important. Also, as seen in lines 8-11, if a new line is encountered, the line number is incremented so that if an error is encountered the correct line number is displayed. If, however the next character is a '/', the scanner checks whether the following character is a '/' or a '\*' since this would mean that the lexer encountered comments. If single line comments are encountered the lexer keeps on skipping characters until meeting a new line. On the other hand, if the lexer meets a multi-line comment it keeps on skipping characters until meeting with \*/. If a \*/ is not met then a lexical error must occur since the comments were not closed. If none of the above are met, then the lexer gets the next state it must move to from the transition table as shown in line 14 figure 5 above. As shown in lines 15-20 if this new state is a final state then there is no need to store previous states because the lexer found a better token. After the loop is broken then as seen in lines 22-27 the lexer must trim the lexeme back, either to an acceptance state which would create a corresponding token of type related to the state it stopped on. Or if it did not end on a final state this means that a lexical error occurred and thus an exception with an error message highlighting the line number of the error is displayed.

In this task since the parser is table driven, the ideal way to encode the table would have been to make a 2-D array where the columns represent the state and the rows represented the character. However, this was not done in this way and instead a switch statement was wrapped around a function name transitionFunction, where given the current state the lexer is in and the read character, this function returns the next state. This was done in this way because the code becomes more readable. Another implementation specific detail is that comments, whitespace and new lines are not passed as tokens to the parser, but they are handled directly in the lexer since these are of no use to the parser.

### Test Cases and Results:

The code in the main program that was used in these test cases is commented out, since in task 6 the REPL had to be implemented. However, the code will be shown in screen shots below.

The following code in figure 6 below found commented in the MiniLangI class is used in order to invoke the getNextToken function. Note in the test cases, the displayed attribute of the token is the token name, since the token type may correspond to different tokens.:



```

74         auto* token = lexer->getNextToken(); //for testing lexer only
75         while(token->getTokenType() != Token::TOK_EOF){ //for testing lexer only
76             cout<<token->getTokenName()<<" "<<endl; //for testing lexer only
77             token = lexer->getNextToken(); //for testing lexer only
78         } // for testing lexer only
79

```

Figure 6-invocation in main class

### Test Case 1: Testing Tokens for Factorial program

#### Input Program:

```

def fact (n: int): int {
    if ((n==0) or (n==1)){
        return n;
    }else{
        return n*fact(n-1);
    }
}

```

#### Expected Result:

KEYWORD TOKEN ,IDENTIFIER TOKEN ,LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN ,COLON  
TOKEN ,KEYWORD TOKEN, RIGHT ROUND BRACKET TOKEN, COLON TOKEN ,KEYWORD TOKEN ,  
LEFT CURLY BRACKET TOKEN, KEYWORD TOKEN, LEFT ROUND BRACKET TOKEN , LEFT ROUND  
BRACKET TOKEN, IDENTIFIER TOKEN, IS EQUAL TO TOKEN , INTEGER NUMBER, RIGHT ROUND  
BRACKET TOKEN ,KEYWORD TOKEN, LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN, IS EQUAL  
TO TOKEN, INTEGER NUMBER ,RIGHT ROUND BRACKET TOKEN, RIGHT ROUND BRACKET TOKEN  
,LEFT CURLY BRACKET TOKEN, KEYWORD TOKEN, IDENTIFIER TOKEN ,SEMI COLON TOKEN  
,RIGHT CURLY BRACKET TOKEN, KEYWORD TOKEN, LEFT CURLY BRACKET TOKEN, KEYWORD  
TOKEN, IDENTIFIER TOKEN, BINARY OPERATOR TOKEN, IDENTIFIER TOKEN, LEFT ROUND  
BRACKET TOKEN, IDENTIFIER TOKEN ,MINUS TOKEN, INTEGER NUMBER, RIGHT ROUND  
BRACKET TOKEN, SEMI COLON TOKEN, RIGHT CURLY BRACKET TOKEN, RIGHT CURLY BRACKET  
TOKEN

#### Actual Result:

KEYWORD TOKEN ,IDENTIFIER TOKEN ,LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN ,COLON  
TOKEN ,KEYWORD TOKEN, RIGHT ROUND BRACKET TOKEN, COLON TOKEN ,KEYWORD TOKEN ,  
LEFT CURLY BRACKET TOKEN, KEYWORD TOKEN, LEFT ROUND BRACKET TOKEN , LEFT ROUND  
BRACKET TOKEN, IDENTIFIER TOKEN, IS EQUAL TO TOKEN , INTEGER NUMBER, RIGHT ROUND  
BRACKET TOKEN ,KEYWORD TOKEN, LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN, IS EQUAL  
TO TOKEN, INTEGER NUMBER ,RIGHT ROUND BRACKET TOKEN, RIGHT ROUND BRACKET TOKEN  
,LEFT CURLY BRACKET TOKEN, KEYWORD TOKEN, IDENTIFIER TOKEN ,SEMI COLON TOKEN  
,RIGHT CURLY BRACKET TOKEN, KEYWORD TOKEN, LEFT CURLY BRACKET TOKEN, KEYWORD  
TOKEN, IDENTIFIER TOKEN, BINARY OPERATOR TOKEN, IDENTIFIER TOKEN, LEFT ROUND  
BRACKET TOKEN, IDENTIFIER TOKEN ,MINUS TOKEN, INTEGER NUMBER, RIGHT ROUND

BRACKET TOKEN, SEMI COLON TOKEN, RIGHT CURLY BRACKET TOKEN, RIGHT CURLY BRACKET TOKEN

**Test Outcome: Success**

### Test Case 2: Test error in factorial program

#### Input Program:

```
def fact (n: int): int {  
    if ((#==0) or (n==1)) {  
        return n;  
    } else {  
        return n*fact(n-1);  
    }  
}
```

#### Expected Result:

KEYWORD TOKEN, IDENTIFIER TOKEN, LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN, COLON TOKEN, KEYWORD TOKEN, RIGHT ROUND BRACKET TOKEN, COLON TOKEN, KEYWORD TOKEN, LEFT CURLY BRACKET TOKEN, KEYWORD TOKEN, LEFT ROUND BRACKET TOKEN, LEFT ROUND BRACKET TOKEN, syntax error in line 2

#### Actual Result:

KEYWORD TOKEN, IDENTIFIER TOKEN, LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN, COLON TOKEN, KEYWORD TOKEN, RIGHT ROUND BRACKET TOKEN, COLON TOKEN, KEYWORD TOKEN, LEFT CURLY BRACKET TOKEN, KEYWORD TOKEN, LEFT ROUND BRACKET TOKEN, LEFT ROUND BRACKET TOKEN, syntax error in line 2

**Test Outcome: Success**

### Test Case 3: Test skipping of comments and error in last line.

#### Input Program:

```
def fact (n : int) : int{  
    /** hey */  
    // bye  
}  
/**  
*/  
?
```

#### Expected Result:

KEYWORD TOKEN, IDENTIFIER TOKEN, LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN, COLON TOKEN, KEYWORD TOKEN, RIGHT ROUND BRACKET TOKEN, COLON TOKEN, KEYWORD TOKEN, LEFT CURLY BRACKET TOKEN, RIGHT CURLY BRACKET TOKEN, syntax error in line 7

**Actual Result:**

KEYWORD TOKEN, IDENTIFIER TOKEN, LEFT ROUND BRACKET TOKEN, IDENTIFIER TOKEN, COLON  
TOKEN, KEYWORD TOKEN, RIGHT ROUND BRACKET TOKEN, COLON TOKEN, KEYWORD TOKEN,  
LEFT CURLY BRACKET TOKEN, RIGHT CURLY BRACKET TOKEN, syntax error in line 7

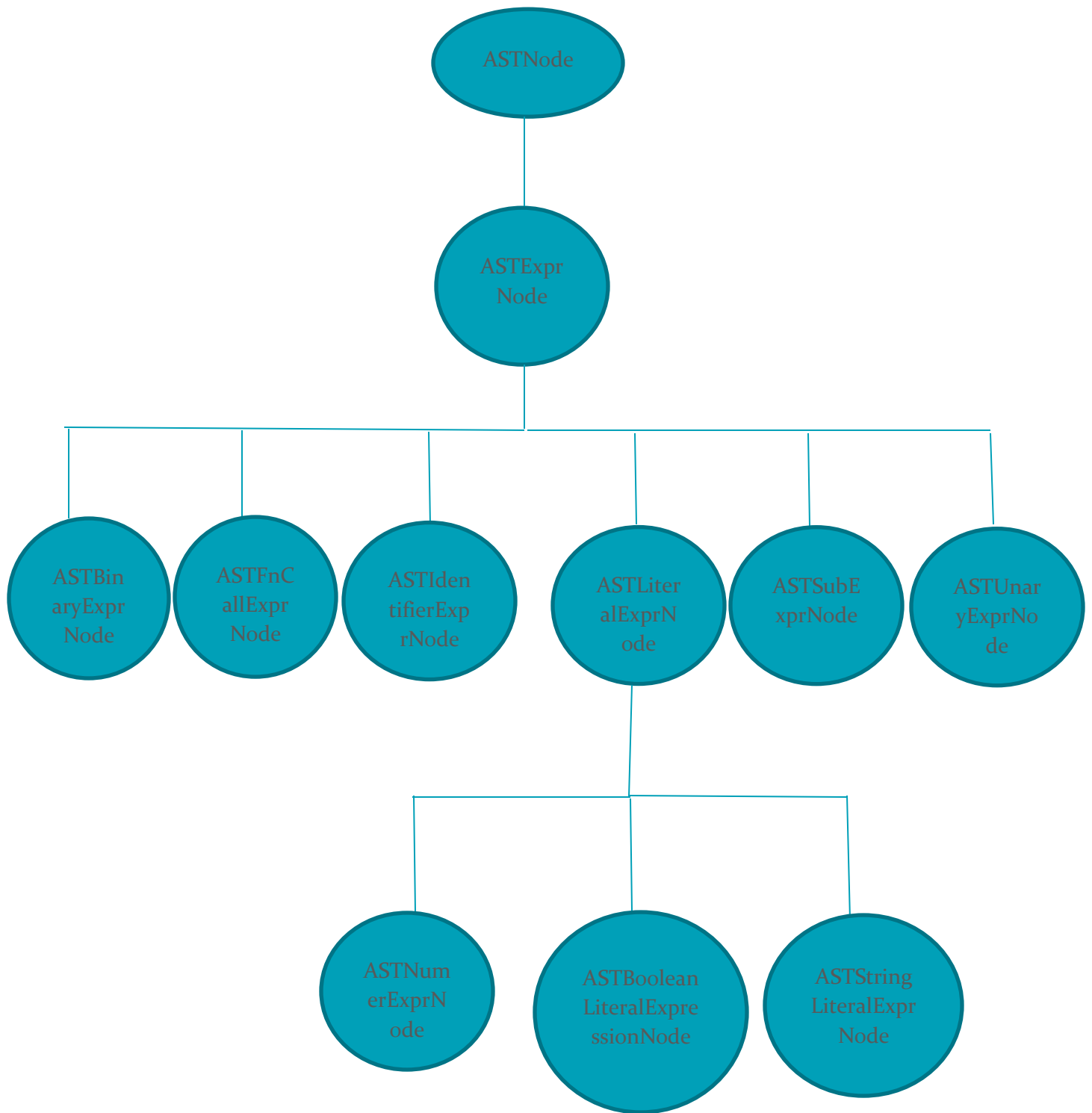
**Test Outcome: Success**

## Task 2: Hand-Crafted Recursive Descent Parser.

### Implementation and Design of the Parser

For this task, a recursive descent predictive parser had to be created from scratch. A recursive descent parser constructs a parse tree where the tree may require back tracking or it may not depending on which type of parser is implemented. A predictive parser is a recursive descent parser that does not require back tracking. The predictive parser uses a pointer called the lookahead which determines which production rules are to be used in order to produce the parse tree for the source program and then eventually the AST. However, one drawback of predictive parsing is that it accepts only one class of grammars called LL(K) grammars. [5]

Since a successful parse of the source program is determined by a correct AST representation of the source program, first the nodes of the AST had to be designed. The following diagram (figure 7) represents the class hierarchy that was created for the AST (The hierarchy is split in 2 due to the large diagram):



*Figure 7.1-Branch 1 of  
AST Hierarchy*

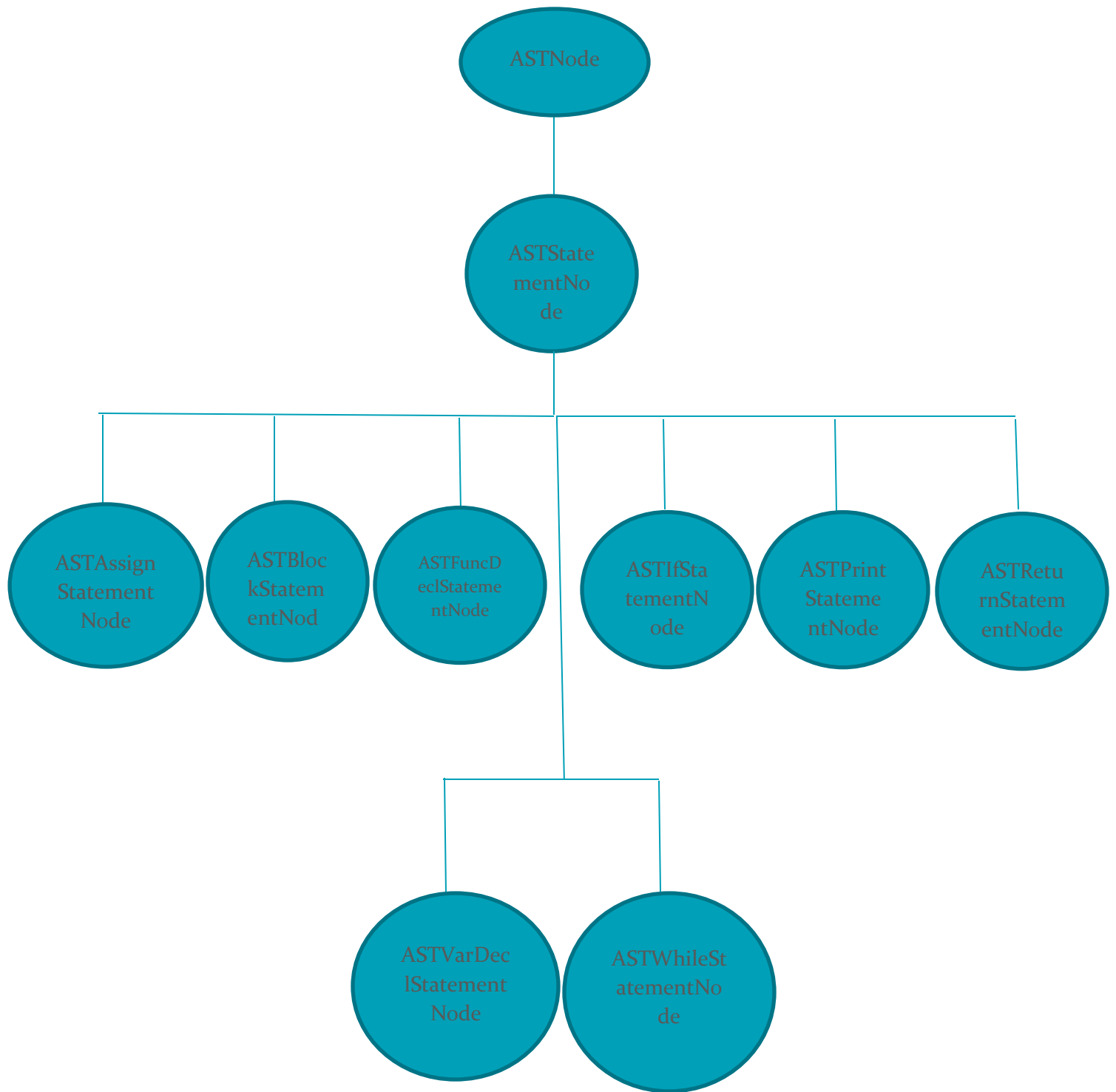


Figure 8.2-Branch 2 of  
AST Hierarchy

As seen in figures 7.1 and 7.2, the class hierarchy is divided into 2 branches. The first branch is dedicated to expressions, whilst the second branch is dedicated to the statements. In the figure above, the parent root node is the virtual class ASTNode where in this class only 1 method is declared. This method is the virtual accept method that will be used for the visitor design pattern. Thus, since all the other concrete classes nodes are descendants of ASTNode, then they must all implement the accept method. The classes ASTExprNode and ASTStatementNode are also both virtual classes since an expression or statement can be of various types and thus the accept method cannot be implemented. In addition to the accept method, the ASTExprNode and ASTStatementNode contain a field that stores the type of the expression. For example, if an expression is a function call, then the enum value FNCALL is stored in the field expressionType in the class ASTExprNode, this way needed because in the visitor classes there were scenarios that the type of the expressions or statements had to be inferred directly. Another virtual class is the ASTLiteralExprNode, this class had to be created so that when parsing numbers, Booleans or strings these could be parsed together in the parse Literal () method. The remaining class nodes are all concrete classes and for further implementation details, see the source code attached in the deliverables. It is important to note that a node may have a number of other nodes attached to it as children. For example, the ASTWhileStatementNode has 2 children, the ASTExprNode that stores the predicate to be evaluated and the ASTBlockStatementNode that stores the block to be executed if the predicate evaluates to true.

The next class that needed to be designed was the PredictiveParser class. The PredictiveParser class contains a lexer, currentToken, nextToken and a vector that stores pointers of type ASTStatementNode as fields. The lexer is used so that the parser always gets the next token that is scanned from the source code. The current token stores the token that is currently being parsed, the nextToken field is the lookahead that stores the next token that is going to be parsed, and the vector stores the AST that will be returned to the semantic analyzer. In addition to these fields, methods had to be designed to parse the different expressions and statements that could occur in the program. For example, consider the scenario when a while statement token is returned by the lexer. The code in figure 9 below shows the implementation for parsing a while statement token.

```

442  ASTWhileStatementNode *Parser::PredictiveParser::parseWhileStatement() {
443      if(nextToken->getTokenStringValue() != "{"){ //if user forgot to open brackets notify with error message
444          throw CompilingErrorException("Forgot opening brackets in while statement");
445      }
446      lookahead();
447      if(nextToken->getTokenType() == Lexer::Token::TOK_EOF){ //if user forgot to put expression notify with error message
448          throw CompilingErrorException("Forgot expression in while statement");
449      }
450      lookahead();
451      ASTExprNode* expr = parseExpression();
452      if(nextToken->getTokenStringValue() != "){ //if user forgot to close expression notify with error message
453          throw CompilingErrorException("Forgot closing brackets in while statement");
454      }
455      lookahead();
456      if(nextToken->getTokenStringValue() != "{"){ // if user forgot to open scope notify with error message
457          throw CompilingErrorException("Forgot opening scope in while statement");
458      }
459      lookahead();
460      ASTBlockStatementNode* block = parseBlockStatement(); //parse block
461      return new ASTWhileStatementNode(expr,block); //return new while node accordingly
462  }

```

Figure 9-parseWhileStatement () function

As seen in figure 9 lines 443-445 above, if a while statement token is received, first the predictive parser checks whether the next token is related to the lexeme “(”. If this is true then the parser continues parsing, otherwise there must be a syntax error in the program, and thus the programmer must be notified. Then as seen in line 46, the token stored in nextToken is moved to currentToken in order to be parsed, and nextToken gets the next token from the lexer. Then the predictive parser checks in the same way whether the predicate expression, and the tokens related to the lexemes “)”, “{”, “}” are returned by the lexer. If not, there must be a syntax error in the program and thus the programmer must be notified. If not, that statement must be correctly parsed and, in this case, an ASTWhileStatementNode must be created having 2 children, the ASTExprNode storing the predicate and the ASTBlockStatementNode storing the block that needs to be executed if the predicate evaluates to true. All the other expressions and statements are parsed in the same way, where the nextToken variable is used in order to check whether the correct syntax behavior is present. If yes, a node is created, if not an error is displayed.

It is important to note that in this task not all non-terminals were declared as nodes. For example, the operators were declared as an enum class. This was done in this way so that only the important non-terminals are declared as nodes and thus reducing the number of nodes that make up the hierarchy. In this way the enum classes could be used as a type. For example, in the class ASTBinaryExprNode the operator variable of type class Operators is not a child attached to the ASTBinaryExprNode but only as a value holder in the node. Also, the AST is returned as a vector of statements, thus the root of the AST is the pointer to the vector. This was done in this way since according to the EBNF in the specification, a program is 0 or more statements.

## Test Cases and Results

As described in section 1, the code that will be used in the main program to invoke only the parser for testing is commented out since the REPL is implemented in the same class. However, there are comments that identify that part that was used for parser testing. For this test, since the AST must be checked pictorially if it is correct or not, the XMLGenerator developed in task 3 is going to be used to check if the test is correct or not. Note that testing for the actual xml is done in section 3, here only the structure of the AST is viewed.

The following code in figure 10 below found commented in the MiniLangI class is used in order to invoke the parser.

```
83  /* This was used to test the parser only , to view test , just remove the comments */  
84  xml->visitTree(parser->parse());
```

*Figure 10-Parser Test invocator*



### Test Case 1: Testing AST for hello world program

#### Input Program:

```
def hello (value: string): string {  
    return "hello world";  
}
```

```
print hello ();
```

#### Expected Result:

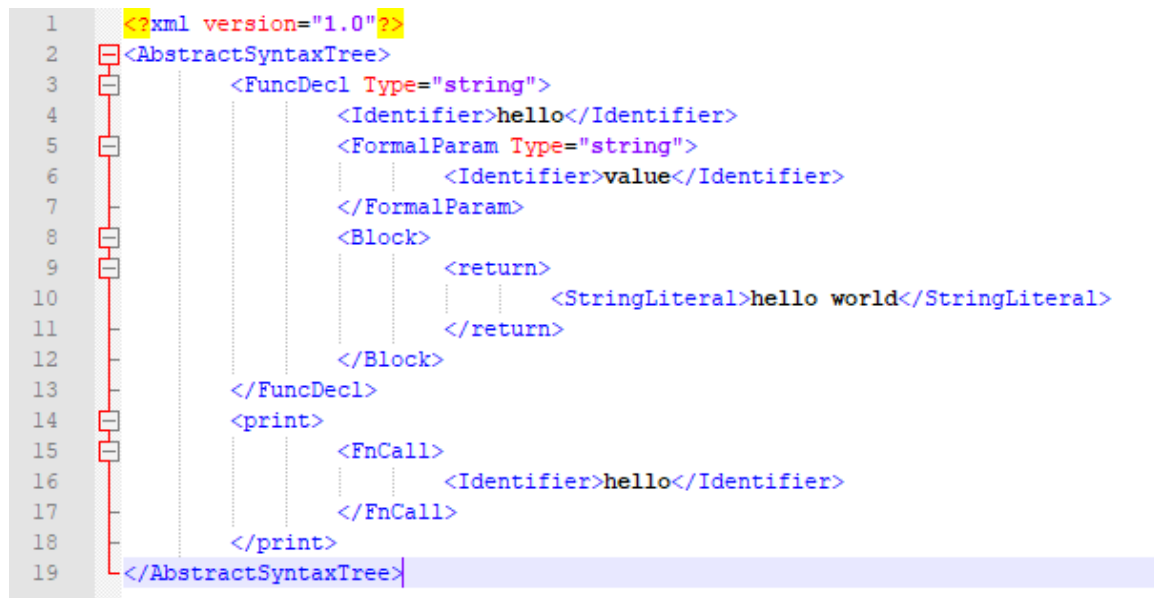


Figure 11-AST for test case 1

#### Actual Result:

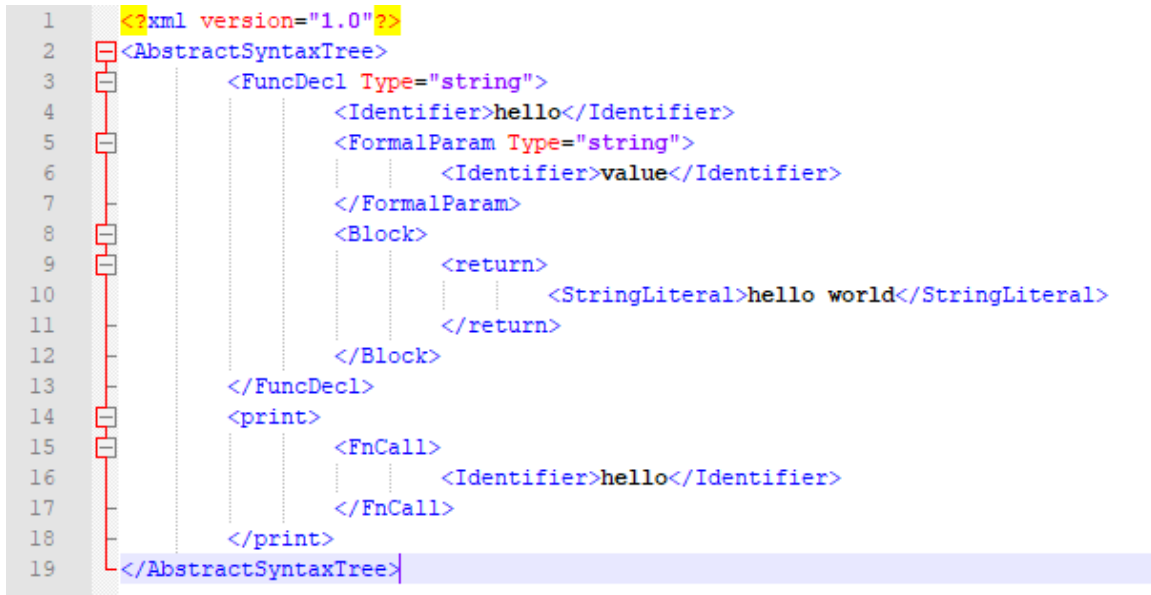


Figure 12-Result for test case 1

**Test Outcome: Success**

## Test Case 2: Testing AST for infinite loop statement

### Input Program:

```

while (n<2) {
    print "infinite loop";
}

```

### Expected Result:

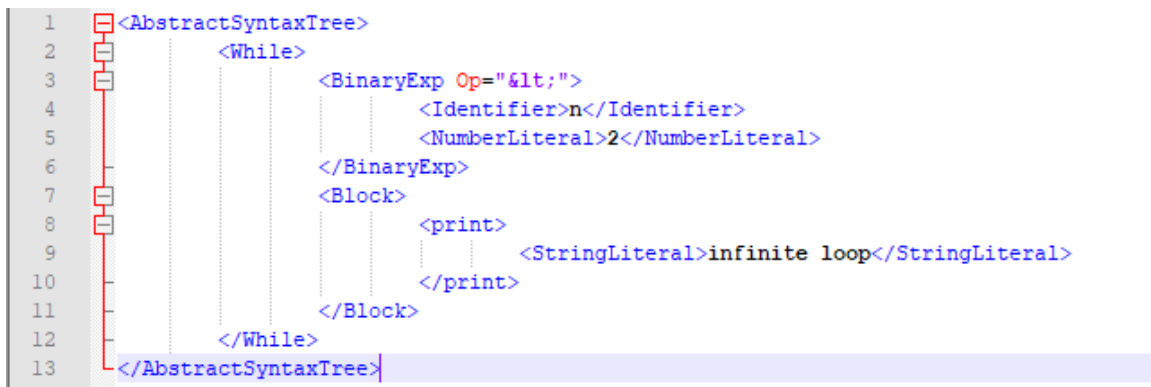


Figure 13-AST for test case 2

#### Actual Result:

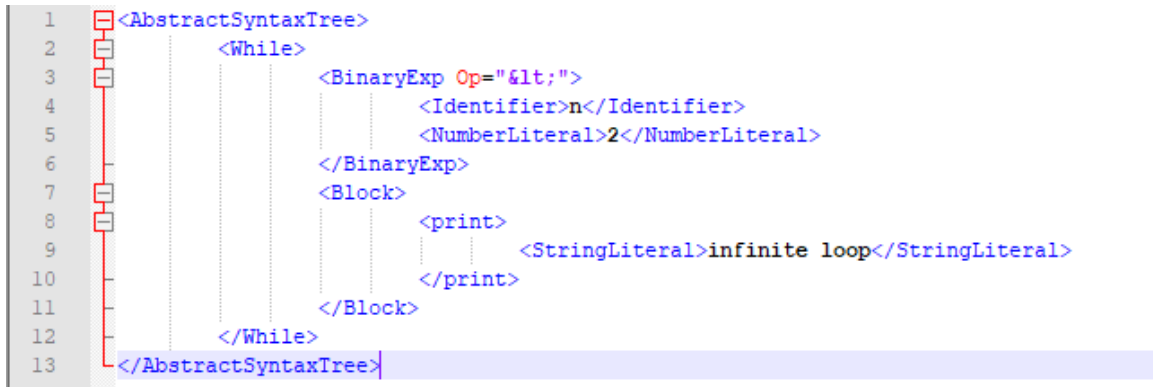


Figure 14-Result for test case 2

**Test Outcome: Success**

#### Test Case 3: Testing Error output for infinite loop statement

##### Input Program:

```
while (n<2)
    print "infinite loop";
}
```

**Expected Result:** Forgot opening scope in while statement

##### Actual Result:

```
Forgot opening scope in while statement
```

Figure 15- Actual Result for test case 3

**Test Outcome: Success**

### Test Case 5: Testing AST for different block statements

#### Input Program:

```
{  
    return n;  
    {  
        return n;  
        {  
            return n;  
        }  
    }  
}
```

#### Expected Result:

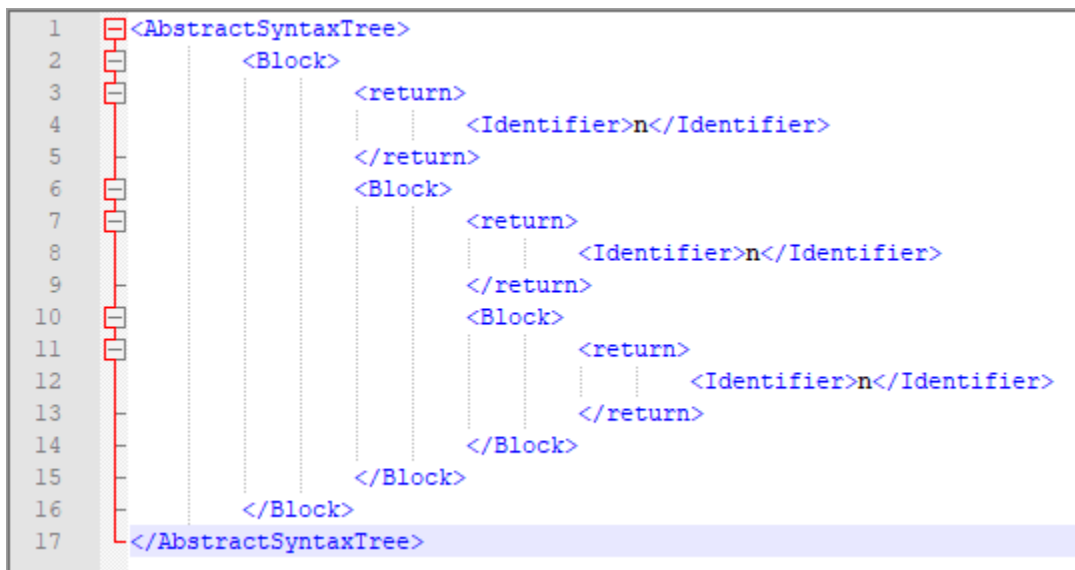


Figure 13- Expected Result for test case 5

**Actual Result:**

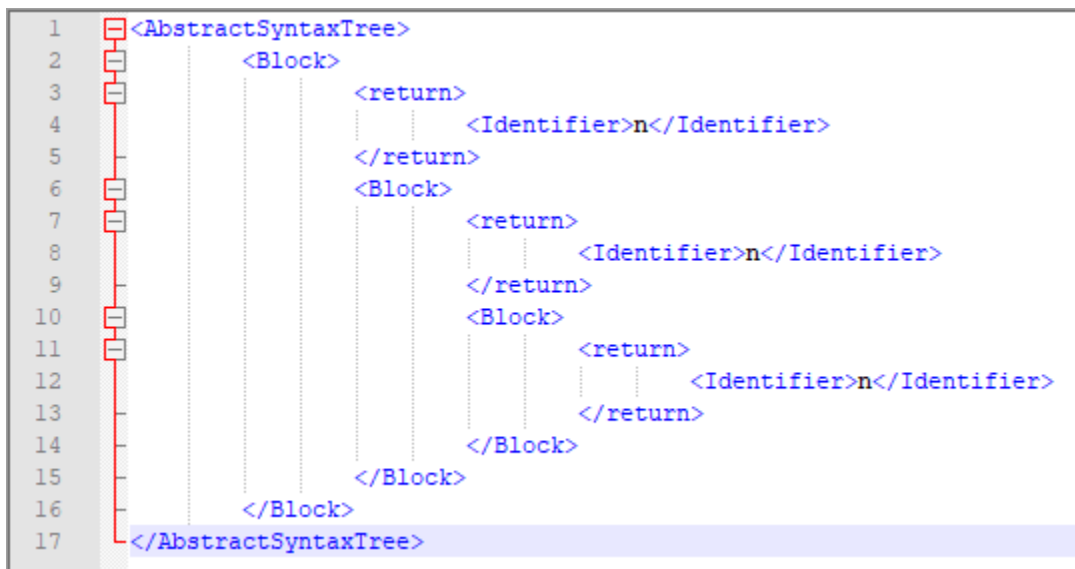


Figure 14-Actual Result For test case 4

**Test Outcome: Success**

Note that more testing was done during the development of the program, however, here only some of the important scenarios are showed.

## Task3: Generate XML of AST

### Implementation and Design

For this task the Visitor design pattern had to be used in order to implement a class that visits each node of the produced AST in task 2 to manipulate and produce the XML representation of the AST. The purpose of the Visitor design pattern is to define a new operation without changing the object structure the operation is to be performed on. Thus, a Visitor class hierarchy must be created where a pure virtual method called visit () is created in an abstract class. This visit method takes a pointer to some element in the object to be operated on. Then each derived class from this abstract class must implement a version of the visit method for each element in the object to be operated on. In addition to the visit () method, the accept () method taking some pointer to a visitor class, must then be implemented in each element of the object structure so that the visitor's visit method on that element can be performed on that element. [6]

Thus, the first thing that was implemented was the accept method in all of the AST nodes that were described in section 2. The accept () method in all of the nodes is implemented as seen in figure 15 below, since all it needs for the node is to call the visitor's visit method so that he can apply his method on it. Note that in the virtual classes ASTNode, ASTExprNode, ASTStatementExprNode and ASTLiteralExprNode the accept method is still kept as pure virtual, since these are abstract classes.

```
21 void AST::ASTBooleanLiteralExpressionNode::accept(Visitor *v) {  
22     v->visit(this);  
23 }
```

*Figure 15-Implementation of accept method*

The second thing that was created was the abstract class Visitor. This abstract class provides the interface for all derived classes where a visit method is created for each node in the AST. A sample of this interface class can be seen in figure 16 below.

```

35      * This method is used by the visitor whenever he visits an ASTAssignStatementNode
36      * @param node
37      * Stores the address of the ASTAssignStatementNode to be visited
38      */
39      virtual void visit(ASTAssignStatementNode *node)=0;
40      /**
41      * This method is used by the visitor whenever he visits an ASTBlockStatementNode
42      * @param node
43      * Stores the address of the ASTBlockStatementNode to be visited
44      */
45      virtual void visit(ASTBlockStatementNode *node)=0;
46      /**
47      * This method is used by the visitor whenever he visits an ASTIfStatementNode
48      * @param node
49      * Stores the address of the ASTIfStatementNode to be visited
50      */
51      virtual void visit(ASTIfStatementNode *node)=0;

```

Figure 16-Sample of Visitors.h class

Thirdly, the derived class XMLGenerator was created. This class implements all the visit methods for each node and outputs the XML representation of the tree according to which node it is visiting. In addition to the implementation of the pure virtual methods in the parent class, the XMLGenerator has an additional 2 private methods and 1 private field. The private field indentationLevel stores the number of tabs currently in the XML representation, and the methods TabIn () and TabOut () go in or out an indentation level depending on which part of the tree the visitor class is in.

For example, if we have the AST in figure 17 below the resulting XML is also below:

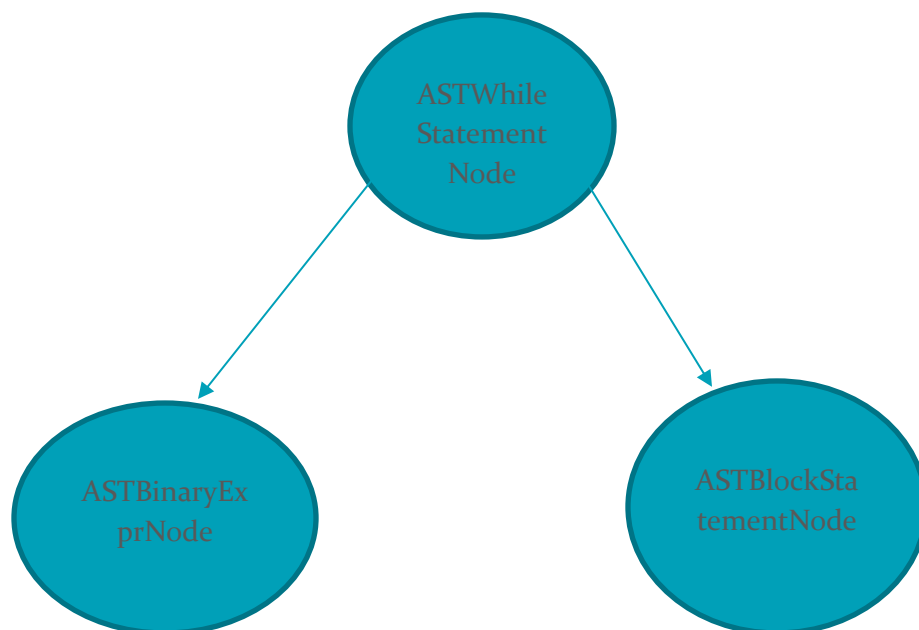


Figure 17- Sub-AST example

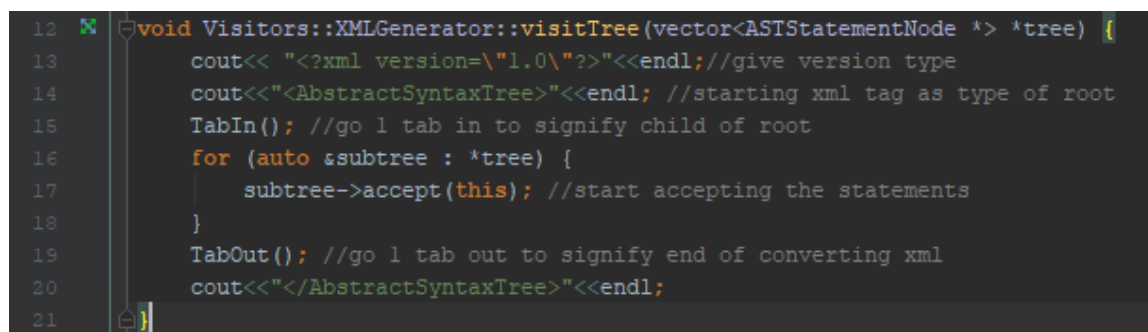
```

<AbstractSyntaxTree>
    <WhileStatement>
        <Block>
            ...
        </Block>
    <ASTBinaryExprNode>
        ...
    </ASTBinaryExprNode>
</WhileStatement>
</AbstractSyntaxTree>

```

Note in the above code the ... means there is more information related to the block stored either in the block or as children attached to that node.

The XMLGenerator class is invoked by starting to visit the AST as shown in figure 18 below:



```

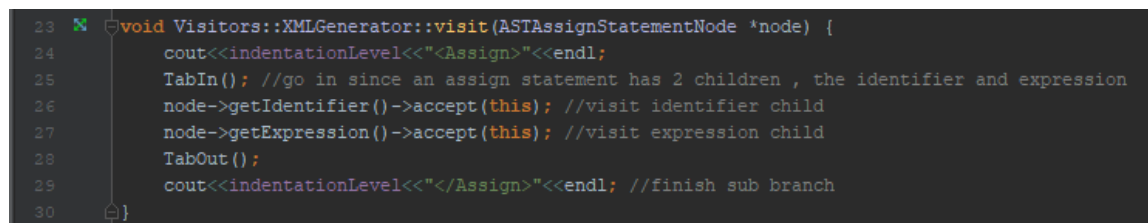
12 void Visitors::XMLGenerator::visitTree(vector<ASTStatementNode *> *tree) {
13     cout<< "<?xml version='1.0'?'><<endl; //give version type
14     cout<<"<AbstractSyntaxTree><<endl; //starting xml tag as type of root
15     TabIn(); //go 1 tab in to signify child of root
16     for (auto &subtree : *tree) {
17         subtree->accept(this); //start accepting the statements
18     }
19     TabOut(); //go 1 tab out to signify end of converting xml
20     cout<<"</AbstractSyntaxTree><<endl;
21 }

```

Figure 18-The method used to start visiting the tree

As shown in figure 18 line 14 above, first the tag <AbstractSyntaxTree> is outputted, this is an indicator that the AST is started to be visited. Thus, since each statement is a child of the root node, as shown in line 15 we need to go in by 1 indentation level. As shown in lines 16-18 each statement of the program is then visited by calling the accept method of the node that is being visited using polymorphism and passing 'this' class as the class that is going to visit the node. After all subtrees have been visited, the indentation level must again be tabbed out in order to output the closing tag related to the tag <AbstractSyntaxTree>

As an example, how, the visitor methods were created, figure 19 shows the case scenario were the ASTAssignStatementNode is visited.



```

23 void Visitors::XMLGenerator::visit(ASTAssignStatementNode *node) {
24     cout<<indentationLevel<<"<Assign><<endl;
25     TabIn(); //go in since an assign statement has 2 children , the identifier and expression
26     node->getIdentifier()->accept(this); //visit identifier child
27     node->getExpression()->accept(this); //visit expression child
28     TabOut();
29     cout<<indentationLevel<<"</Assign><<endl; //finish sub branch
30 }

```

Figure 19-The visit method for the ASTAssignStatementNode



As shown in figure 19 line 24 above, the tag <Assign> is outputted as we are visiting an assign statement node, where the indentationLevel determines how deep we are in the AST. Now as seen in line 24, since the ASTAssignStatementNode has 2 children, then we need to tab in since we are going down one level in the AST. Thus, as shown in lines 26,27 the children ASTIdentifierExprNode and ASTExprNode are visited, where then their XML representation is outputted in a similar way. As seen in line 28, then we need to tab in again to finish the XML representation of that sub branch. The remaining nodes are visited in a similar way. These were not described in this report because they were all similar to the case scenario of figure 19 above and dioxygen comments were created for each visit method in the source code listing.

Note that for this task there are no deviations from the original EBNF and assumptions taken from the specification.

### Test Cases and Results:

As mentioned in previous sections, since the main class is used to implement the REPL in task 6, this cannot be used to call the XMLGenerator separately. Thus, the testing code shown in figure 20 below to invoke the XMLGenerator will be commented out in the main program.

```
/* This was used to test the parser and xml generator only , to view test , just remove the comments*/  
xml->visitTree(parser->parse());
```

*Figure 20-Code used to invoke the XMLGenerator in order to test the program*

#### Test Case 1: Testing XML for Square function

##### Input Program:

```
def funcSquare (x: real): real {  
    return x*x;  
}  
  
var y: real = funcSquare (2.5);  
print y;
```

**Expected Result:**

```
1  <?xml version="1.0"?>
2  <AbstractSyntaxTree>
3      <FuncDecl Type="real">
4          <Identifier>funcSquare</Identifier>
5          <FormalParam Type="real">
6              <Identifier>x</Identifier>
7          </FormalParam>
8          <Block>
9              <return>
10                 <BinaryExp Op="*">
11                     <Identifier>x</Identifier>
12                     <Identifier>x</Identifier>
13                 </BinaryExp>
14             </return>
15         </Block>
16     </FuncDecl>
17     <VarDecl Type="real">
18         <Identifier>y</Identifier>
19         <FnCall>
20             <Identifier>funcSquare</Identifier>
21             <NumberLiteral>2.5</NumberLiteral>
22         </FnCall>
23     </VarDecl>
24     <print>
25         <Identifier>y</Identifier>
26     </print>
27 </AbstractSyntaxTree>
```

Figure 21-Expected Result for test case 1

## Actual Result:



Figure 22-Actual Result for test case 1

**Test Outcome: Success**

## Test Case 2: Testing XML for program having a function declaration inside a function declaration

### Input Program:

```
def hello() : real{
  def hello(n:int) : real{
    return 2.0;
  }
  return hello(2);
}
```

### Expected Result:

```
1 <AbstractSyntaxTree>
2   <FuncDecl Type="real">
3     <Identifier>hello</Identifier>
4     <Block>
5       <FuncDecl Type="real">
6         <Identifier>hello</Identifier>
7         <FormalParam Type="int">
8           <Identifier>n</Identifier>
9         </FormalParam>
10        <Block>
11          <return>
12            <NumberLiteral>2</NumberLiteral>
13          </return>
14        </Block>
15      </FuncDecl>
16      <return>
17        <FnCall>
18          <Identifier>hello</Identifier>
19          <NumberLiteral>2</NumberLiteral>
20        </FnCall>
21      </return>
22    </Block>
23  </FuncDecl>
24 </AbstractSyntaxTree>
```

Figure 23-Expected result for test case 2

### Actual Result:

```
1 <AbstractSyntaxTree>
2   <FuncDecl Type="real">
3     <Identifier>hello</Identifier>
4     <Block>
5       <FuncDecl Type="real">
6         <Identifier>hello</Identifier>
7         <FormalParam Type="int">
8           <Identifier>n</Identifier>
9         </FormalParam>
10        <Block>
11          <return>
12            <NumberLiteral>2</NumberLiteral>
13          </return>
14        </Block>
15      </FuncDecl>
16      <return>
17        <FnCall>
18          <Identifier>hello</Identifier>
19          <NumberLiteral>2</NumberLiteral>
20        </FnCall>
21      </return>
22    </Block>
23  </FuncDecl>
24 </AbstractSyntaxTree>
```

Figure 24-Actual Result for test case 3 **Test Outcome: Success**

### Test Case 3: Testing XML for program having if's and While's Together

#### Input Program:

```
while(true) {  
    if(true) {  
        return 1;  
    } else {  
        return 2;  
    }  
}
```

#### Expected Result:



Figure 25-Expected Result for test case 3

#### Actual Result:



Figure 26-Actual Result for test case 3

**Test Outcome: Success**

## Task4: Semantic Analysis Pass

### Implementation and Design

For this task, another visitor class had to be implemented in order to check that the program is semantically valid. The purpose of Semantic checking is to check that the program makes sense, for example that a variable is assigned to an expression of the same type of the variable. For example, the statement, 'var a: int = "hello"' is not a semantically correct program because a string literal cannot be assigned to a variable of type integer. On the other hand, the following statement is a semantically correct statement, 'var a: int =5'. Semantics depend entirely on the programming language, however common semantic checks include, checking that the type return of the function is the same as function definition and that when scopes are created variables and functions with the same name can be created, since these hide the outside scope.

In order to keep track of the function and variable names that were created in a scope, a map had to be created where the key element of the map is the identifier name and the mapped value is an instance of the class TypeBinder. The TypeBinder class includes information about the identifier in the current scope. Such information is whether the identifier is a function or a variable, the value stored in the variable or function in case of recursion, the primitive type of the function or variable and the parameters and block node if the identifier is a function. It is important to note that in this implementation a multi-map was implemented instead of a map. A multi-map works in the same way as a map, however there could be key values mapped to more than 1 value. This was done in this way because in the same scope a variable and a function can have the same name.

The next thing that was designed and implemented was the SymbolTable class. The SymbolTable class encodes the symbol table for one particular scope. In fact, this class encodes the multi-map described in the paragraph above and any methods to be performed on the symbol table such as adding a new value to the symbol table, getting the type binder mapped to an identifier and checking whether a particular identifier with a specific type binder is in the symbol table or not.

After all the utility classes were designed, the next step was to implement the actual Semantic Analysis visitor class. The semantic analysis visitor class was designed to be a concrete class of the Visitor interface class, thus it had to implement all the visit methods related to visiting a particular node in the AST. The semantic analysis visitor class had a private field called ScopedTable, this is a vector of Symbol Tables, i.e. representing the symbol table for different scopes. The first element in the ScopedTable vector is the global scope. In addition to this field there are also other fields that are used for type checking, see dioxygen comments for more details. The next paragraph describes how the Semantic Analysis visitor class works.

The first important check in this visitor class is checking that when a variable is assigned a value, it is assigned a value of the same type of the variable. This was implemented by creating a private field called 'typeToBeChecked' which stores the type of the expression that has been checked recently. In fact, as seen in figure 27 below, whenever a literal is visited, its type is returned. The same is done when visiting an identifier, the type of the identifier is stored in the variable typeToBeChecked.

```

243 void SemanticAnalysis::visit(ASTNumberExprNode *node) {
244     if(node->getNumberType() == ASTNumberExprNode::REAL) {
245         typeToBeChecked = Type::REAL;
246     } else if (node->getNumberType() == ASTNumberExprNode::INT) {
247         typeToBeChecked = Type::INT;
248     }
249 }
250
251 void SemanticAnalysis::visit(ASTBooleanLiteralExpressionNode*) {
252     typeToBeChecked = Type::BOOL;
253 }
254
255 void SemanticAnalysis::visit(ASTStringLiteralExprNode*) {
256     typeToBeChecked = Type::STRING;
257 }

```

Figure 27-Visit implementation for literals

The design decision described in the previous paragraph makes it easier to check that the type of the expression and the variable is the same. In fact, as shown in figure 28 below, first the identifier is visited where its type is stored in typeToBeChecked, similarly the expression is visited and the type is stored in type to be checked. If type of the expression and the type of the variable are not equal, a compiling error exception is thrown, otherwise this part of the program is semantically correct.

```

17 void SemanticAnalysis::visit(ASTAssignStatementNode *node) {
18     node->getIdentifier()->accept(this); //visit identifier to check if it has been declared, and get it's type
19     Type identifierType = typeToBeChecked; // store the type
20     node->getExpression()->accept(this); // visit the expression to check it's type
21     Type expressionType = typeToBeChecked; //store type
22     if(identifierType != expressionType){ // if variable type and it's assigned expression are not the same we have a se
23         // note int and reals are different types and are not automatically type caste
24         throw CompilingErrorException("The identifier "+node->getIdentifier()->getIdentifierName()+" does not have the "
25             " same type as expression");
26     }
27 }

```

Figure 28-Visit implementation for Assign Node

As shown in figure 29 below, in addition to storing the type in typeToBeChecked, visiting the ASTIdentifierExprNode makes sure that if the variable is not declared in the program then the program is semantically incorrect. In fact, in figure 29 below if the variable name is not found the method won't return and thus an exception is thrown.



```

259 void SemanticAnalysis::visit(ASTIdentifierExprNode *node) {
260     for(int i=0;i<ScopedTable.size();i++){ //Check if variable is declared in some scope
261         if(ScopedTable.at(ScopedTable.size()-i-1)->checkIfInSymbolTable(node->getIdentifierName(),TypeBinder::VARIABLE)){
262             typeToBeChecked = ScopedTable.at(ScopedTable.size()-i-1)->getTypeBinder(node->getIdentifierName())
263                 ,TypeBinder::VARIABLE).getPrimitiveType();
264             //if found store it's type.
265             return;
266         }
267     }
268     throw CompilingErrorException("Variable "+node->getIdentifierName()+" was not declared");
269 }

```

Figure 29-Visit of ASTIdentifierExprNode

Regarding variables in addition to type checking the visitor class is made to check that whenever a variable declaration happens, the method checks whether a variable with the same name is already in the same scope. If this happens a compiling error exception is thrown. This is done as shown in lines 99-101 below, where if a variable already exists in the current scope with the same name, an exception is thrown. Thus, it is important to note that in this implementation there could be variables that have the same name in different scopes per assignment specification. Also, since as shown in line 99 the type of the identifier is specified (variable or function), this enables the programmer to create a function and a variable with the same name as per assignment specification.

```

96 void SemanticAnalysis::visit(ASTVarDeclStatementNode *node) {
97     //if in the same scope the variable is being declared we have a variable that has already been declared with the same
98     // name , we have a semantic error , (notelast scope is last element in vector).
99     if(ScopedTable.at(ScopedTable.size()-1)->checkIfInSymbolTable(node->getIdentifier()->getIdentifierName(),TypeBinder::VARIABLE)){
100         throw CompilingErrorException("A variable with name "+node->getIdentifier()->getIdentifierName()+" has already been declared
101     }

```

Figure 30-Variable name checking

With regards to functions the semantic analyzer was implemented to check some important function properties. First as shown in figure 31 below, the semantic analyzer checks whether there is a function with the same name in the current scope. If this happens, then a compiler error occurs. Due to this design decision, there could be functions with the same name in different scopes as required. Also, there could be a function and variable with the same name in the same scope. However, this implementation does not include function overloading in the same scope (i.e. functions with the same name in the same scope having different parameters). This was not done because it was not specified in the assignment specification sheet.

```

126 //if a function with the name stored in the ASTFunctionStatementNode has already been declared then we get a semantic error
127 if(ScopedTable.at(ScopedTable.size()-1)->checkIfInSymbolTable(node->getIdentifier()->getIdentifierName(),TypeBinder::FUNCTION)){
128     throw CompilingErrorException("A function with name "+node->getIdentifier()->getIdentifierName()+" has already been declared i
129 }

```

Figure 31-Name checking in functions

More checks that were done in this implementation related to functions is checking that the type return of the function in the function definition is the same as the return statement in the function and checking that a return statement exists. It is also important to note that it was checked that if there is an if statement inside a function definition that all branches have a return statement for correct semantics. To see implementation details, check dioxygen comments.

Another important step was that whenever a block is met by the visitor class, a new scope is created and destroyed when the visitor exists the block. This can be seen in figure 31 below:

```

29 void SemanticAnalysis::visit(ASTBlockStatementNode *node) {
30     ScopedTable.push_back(new SymbolTable()); //create new scope since we met a new block

```

Figure 32-Part of the ASTBlockStatementNode visit implementation

In this implementation it was assumed that functions with the same name in the same scope are not allowed. This was done in this way because it was not specified in the assignment specification sheet. However, all other requirements were implemented.

A summary of the semantic rules implemented were the following:

- No variables and functions with the same name in the same scope
- Can declare variables and functions with the same name in different scope
- A new block statement creates a new scope which is terminated when the visitor class exits the block's scope.
- Each variable's type must match with the expression it is assigned to's type
- The function declaration's return type must equal the function's return statement type
- Each Function must have a return statement
- Arithmetic Operators can be done on operands of the same type (this was done in the ASTBinaryExprNode by checking that the value stored in the variable typeToBeChecked when visiting the LHS is equal to the value stored in the variable typeToBeChecked when visiting the RHS). Also, for unary expressions '-' can be applied to numbers whilst 'not' can be applied to Booleans.

Since the semantic analyzer visitor class contains a lot of implementation details, only the design decisions were discussed here, for specific implementation details see the source code with very detailed dioxygen comments.

## Test Cases and Results

### Test Case 1: Checking that no variables can have the same name in the same scope

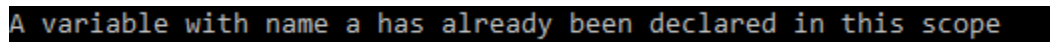
#### Input Program:

```
var a: int = 2;  
var a: bool = true;
```

#### Expected Result:

A variable with name a has already been declared in this scope

#### Actual Result:



A variable with name a has already been declared in this scope

*Figure 33-Actual Result of Test case 1*

**Test Outcome: Success**

### Test Case 2: Checking that variables can have the same name in different scopes

#### Input Program:

```
var a: int = 2;  
{  
    var a: string = "hello";  
}
```

**Expected Result:** Semantically Correct

**Actual Result:** Semantically Correct

**Test Outcome: Success**

### Test Case 3: Checking that functions can have the same name in different scopes

#### Input Program:

```
def fact (): int {  
    def fact ():int {  
        return 2;  
    }  
    return fact ();  
}
```

**Expected Result:** Semantically Correct

**Actual Result:** Semantically Correct

**Test Outcome: Success**

#### Test Case 4: Checking that functions can't have the same name in the same scope

##### Input Program:

```
def fact (): int {  
    return 2;  
}  
def fact ():int {  
    return 2;  
}
```

##### Expected Result:

A function with name fact has already been declared in this scope.

##### Actual Result:

```
A function with name fact has already been declared in this scope
```

**Test Outcome: Success**

#### Test Case 5: Checking that variables cannot be assigned to expressions of different types

##### Input Program:

```
var a: int = "hello";
```

##### Expected Result:

Type of variable a does not match with expression

##### Actual Result:

```
Type of variable a does not match with expression
```

**Test Outcome: Success**

#### Test Case 6: Checking that variables can be assigned to expressions of same type , and that each function must have a return statement

##### Input Program:

```
var a: real = 2.0;  
  
def fact ():int {  
  
}
```

##### Expected Result:

You forgot return in fact

**Actual Result:**

You forgot return in fact

**Test Outcome: Success**

**Test Case 7: Checking that functions having if statements must have all branches with a return statement if the actual function does not have a return statement**

**Input Program:**

```
def fact (): bool {  
    if(true) {  
        return true;  
    } else {  
        return false;  
    }  
}  
  
def fact1(): bool {  
    if(true) {  
        return true;  
    }  
}
```

**Expected Result:**

You forgot return in fact1

**Actual Result:**

You forgot return in fact1

**Test Outcome: Success**

**Test Case 8: Checking that binary expressions can have an operator applied only to operands of the same type.**

**Input Program:**

```
var a : int = 5.0+4;
```

**Expected Result:**

Binary operators can only be applied to expressions of the same type

**Actual Result:**

Binary operators can only be applied to expressions of the same type

**Test Outcome: Success**

## Task 5: Interpreter Execution Pass

### Implementation and Design

For this task a visitor class had to again be implemented where the class visits each node in the produced AST and executes the program supplied by the user. The only class that was designed and implemented for this task was the InterpreterExecutionPass class which was designed to have a number of private fields. The first field that it was designed to have was a field called validator. This field is of type Semantic Analysis; thus, this indicates that prior to executing the program, the class fires the semantic analyzer first because if the program is not semantically valid it cannot be executed. Like the SemanticAnalysis class the InterpreterExecutionPass class also contains a scoped table that stores the identifiers stored in each scope along with the type binder. In addition to these identifiers, 4 vectors were designed, each storing value for int, string, bool and double. The utility of these vectors can be seen in figure 33 below:

```
229 void InterpreterExecutionPass::visit(ASTBinaryExprNode *node) {
230     node->getLhs()->accept(this); //visit left hand side of the e
231     node->getRhs()->accept(this); // visit right hand side of the
232     bool operand1; // stores one of the operands in case of boole
233     bool operand2; // stores one of the operands in case of boole
234     switch (node->getOperator()){
235         case Operators::PLUS:
236             if(lastEvaluatedType == Type::INT){ // according to t
237                 // and do the ope
238                 int opl = integerVals.at(integerVals.size()-2);
239                 int op2 = integerVals.at(integerVals.size()-1);
240                 integerVals.pop_back();
241                 integerVals.pop_back();
242                 integerVals.push_back(op1+op2);
243                 lastEvaluatedType = Type::INT; // the result of a
```

Figure 34-ASTBinaryExprNode visit implementation

Suppose the ASTBinaryExprNode corresponding to the expression 3+4 is visited. Then using figure 34 above, the left-hand side is visited. Since the LHS is a number expression node, the code in figure 35 below is executed. Thus, as shown in figure 35 (lines 450-451) below, the variable lastEvaluatedType is assigned to type int and the vector relating to the int is appended the number literal 3. As shown in figure 34 line 231 above, then the RHS is visited and as the case for the LHS, the value is also appended on the vector of type int. As shown in the if statement in lines 236-243(fig 34), if the lastEvaluatedType is an integer then both the operand is removed from the vector and 7 is pushed on the vector of type integer so that further manipulation of the result can occur. All the other expressions and literals are evaluated in a similar way, thus to see the rest of the implementation see the dioxygen comments.

```

447 void InterpreterExecutionPass::visit(ASTNumberExprNode *node) {
448     if (node->getNumberType() == ASTNumberExprNode::REAL) { // if n
449                                     // and
450         lastEvaluatedType = Type::REAL;
451         realVals.push_back(node->getValue());
452     } else if (node->getNumberType() == ASTNumberExprNode::INT) { //
453                                     //
454         lastEvaluatedType = Type::INT;
455         integerVals.push_back((int)node->getValue());
456     }
457 }

```

Figure 35-ASTNumberExpressionNode visit implementation

Another important part of the implementation can be seen in figure 36 below. As shown after line 20 in figure 36, after visiting a statement in the block, that statement must be checked whether it is a return statement. If it is a return statement, then the remaining statements in the block do not need to be executed further, thus as seen in line 23 the loop is broken and the scope related to that block is popped.

```

118     vector<ASTStatementNode*> statements = *node->getStatements();
119     for (auto &statement : statements) {
120         statement->accept(this); //visit each statement in the block
121         if (isReturnPresent) { // if a return is identified , then we need to bre
122                                     // statements are unreachable
123             break;
124         }
125     }
126     ScopedTable.pop_back(); // remove scope since we are exiting the block
127 }

```

Figure 36-Part of the visit for ASTBlockStatementNode

The variable isReturnPresent used in figure 36 is updated as shown in figure 37 by the visitor for the ASTReturnStatementNode. This will indicate that a return statement has occurred in a particular scope.

```

208 void InterpreterExecutionPass::visit(ASTReturnStatementNode *node) {
209     node->getExpression()->accept(this); // visit expression to be returned
210     isReturnPresent = true; // indicate that we found a return statement to act accordingly.
211 }

```

Figure 37-Visit for the ASTReturnStatementNode

However, in this implementation whenever the visitor exits from a function declaration the isReturnPresent is set to false again, the reason being is that if we have a function definition inside a function definition, where in the inside function a return is present, then isReturnPresent would still remain true inside the outward function and thus the remaining statements of the outward function would not execute.

The remaining nodes are all executed in a similar way to the mentioned nodes, however for further specific implementation details see the detailed dioxygen comments in the source code listing.

In this implementation the visitor class was implemented as required, however the implementation could have been more efficient by having some type named valueType that can take an int, string, bool or double so that one stack can be implemented of type valueType instead of implemented 4 vectors. However, this was done using 4 vectors in order to be kept as simple as possible. Another feature that could have been added was the feature for function overloading, thus allowing functions to have the same name but different parameters in the same scope. However, this was not done since it was not mentioned in the assignment specification sheet.

## Test Cases and Results

### Test Case 1: Testing recursion result using factorial function

#### Input Program:

```
def fact (n: int): int {  
    if ((n==0) or (n==1)) {  
        return n;  
    } else {  
        return n*fact(n-1);  
    }  
}  
  
print fact (5);
```

#### Expected Result:

120

#### Actual Result:

120

**Test Outcome: Success**

### Test Case 2: Testing Power function giving in assignment specification sheet

#### Input Program:

```
def funcPow ( x : real , n : int ) : real{  
    var y : real = 1.0 ;  
    if (n>0){  
        while ( n>0){  
            set y = y * x ;  
            set n = n - 1 ;  
        }  
    }
```



```

    }
    else{
        while (n<0){
            set y = y / x ;
            set n = n + 1 ;
        }
    }
    return y ;
}

```

```
print funcPow(2.0,2);
```

**Expected Result: 4**

**Actual Result:** 

**Test Outcome: Success**

**Test Case 3: Testing result of program in assignment specification sheet**

**Input Program:**

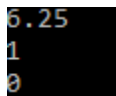
```

def funcSquare (x:real) : real {
    return x*x ;
}
def funcGreaterThen(x:real , y:real ) : bool {
    var ans : bool = true ;
    if ( y > x ) { set ans = false ; }
    return ans ;
}
var x : real = 2.4;
var y : real = funcSquare(2.5);
print y ;
print funcGreaterThen(x,2.3);
print funcGreaterThen(x,funcSquare(1.555));

```

**Expected Result: 6.25,1,0**

**Actual Result:**



**Test Outcome: Success**

#### Test Case 4: Testing result of program having function with the same name in different scopes

##### Input Program:

```
def test(n:int):int{
  def test (n:int):bool{
    if(n == 2){
      return true;
    }else{
      return false;
    }
  }
  if(test(n)){
    return n;
  }else{
    return -n;
  }
}

print test(5);

print test(2);
```

**Expected Result:** -5,2

##### Actual Result:



**Test Outcome: Success**

#### Test Case 5: Testing result of program having same variable name in different scopes which result into an infinite loop

##### Input Program:

```
var a: int =2;
while(a>0) {
  var a: int = 2;
  set a = a-1;
}
```

**Expected Result:** Infinite loop, since the set statement will update the variable 'a' inside the while block

**Actual Result:** Infinite loop

**Test Outcome: Success**

## TASK 6: The REPL

### Implementation and Design

For this task a REPL had to be implemented, where the main class turns itself into an interactive console for the interpreter. Thus, the main class MiniLangI was created where this contains all the logic related to the REPL. The REPL specific commands that this console could take were designed to be the following:

- #st -> Display Symbol Table
- #exit -> exit from REPL
- #load -> load some source program from /CompilersAssignment/TestPrograms
- #help -> Displays a help screen to the user

These commands were designed to be very case sensitive, thus for example if the user types “#load” or “#Load” the REPL would display a syntax error.

As shown in figure 38 lines 45-48 below, the REPL contains an instance of the lexer, parser, xml generator (only used for testing) and the interpreter. An instance of the lexer was needed in order to initialize the lexer pointer in the parser. Note that no semantic analyzer was initialized in this class because the interpreter has its own semantic analyzer.

```
45     auto * lexer = new LexerImplementation(); //instance of the lexer
46     auto* parser = new PredictiveParser(lexer); // instance of the parser
47     auto* xml = new XMLGenerator();
48     auto* interpreterExecutor = new InterpreterExecutionPass(); // instance of the
49     cout<<"Minilang Interpreter running ,enter #help for command breakdown"<<endl;
50     cout<<"MLI> ";
51     string input;
52     SymbolTable tempValue ; // used in case when an exception is thrown whilst in a
53     getline(cin,input); //get input
```

Figure 38-Design of MiniLangI

```

55     while(true) { // keep iterating
56         //Store previous state so that if an exception happens whilst the interpreter is in a function block, the s
57         // of the program prior to loading the program is kept, this had to be implemented because in the function
58         //declaration in order to implement factorial, the function had to be inserted immediately in the symbol tab
59         //for semantics
60         for(unsigned int i=0;i<interpreterExecutor->getValidator()->getScopedTable().size();i++){
61             scopedTemp.push_back(*interpreterExecutor->getValidator()->getScopedTable().at(i));
62         }
63         try {
64             if (input == "#help") { // if help is entered display help screen menu
65                 help();
66             } else if (input == "#st") { // if st is entered
67                 vector<SymbolTable*> st = interpreterExecutor->getScopedTable();// get symbol table containingg values
68                 for (auto &i : st) {
69                     auto contents = i->getMultimap(); // get the multimap corresponding to the current scope
70                     int j = 0; // sort of index in the multimap
71                     for (auto &content : contents) { // iterate untill the end of multimap
72                         j++;
73                         //display the name of the identifier and it's type(function or variable)
74                         cout <<j<<". "<<content.first<<" "<<content.second.getStringRepresentationOfIdentifierType()<<
75                         " "<<content.second.getStringRepresentationOfPrimitiveType();
76                         if(content.second.getStringRepresentationOfIdentifierType()=="variable"){ //display value if v
77                             if(content.second.getStringRepresentationOfPrimitiveType()=="int"){

```

Figure 39-Implementation of MiniLangI

As seen in figure 39 above, the REPL is made to keep iterating in order to get line by line commands, it is important to note that in this implementation the REPL can take only single line commands, whilst whole programs could only be inputted using the #load command. This was implemented in this way since the load command would be sufficient enough. As seen in line 60 in figure 39 above, the current state of the Scope table is stored prior to modifying it further. This was done so that if an exception happens in a middle of a function block, the erroneous function would not have its name still binded in the symbol table. Thus, the REPL can retrieve its previous state. As seen in lines 64-76 figure 39 above if the user enters “#help” the help screen is outputted, otherwise if he enters “#st” the symbol table is outputted. As seen in figure 40 lines 76-87, if an element in the scope table is a variable, then the variable’s name, type, identifier type, primitive type and value are display. On the other-hand if the element in the scope table is a function, the function identifier, identifier type and primitive are displayed.

```

70     int j = 0; // sort of index in the multimap
71     for (auto &content : contents) { // iterate untill the end of multimap
72         j++;
73         //display the name of the identifier and it's type(function or variable)
74         cout <<j<<". "<<content.first<<" "<<content.second.getStringRepresentationOfIdentifierType()<<
75         " "<<content.second.getStringRepresentationOfPrimitiveType();
76         if(content.second.getStringRepresentationOfIdentifierType()=="variable"){ //display value if var
77             if(content.second.getStringRepresentationOfPrimitiveType()=="int"){
78                 cout<<" "<<content.second.getValueInIdentifier()->intValue<<endl;
79             }else if (content.second.getStringRepresentationOfPrimitiveType()=="string"){
80                 cout<<" "<<content.second.getValueInIdentifier()->stringValue<<endl;
81             }else if (content.second.getStringRepresentationOfPrimitiveType()=="real"){
82                 cout<<" "<<content.second.getValueInIdentifier()->realValue<<endl;
83             }else if (content.second.getStringRepresentationOfPrimitiveType()=="bool"){
84                 cout<<" "<<content.second.getValueInIdentifier()->boolValue<<endl;
85             }
86         }else{
87             cout<<endl;
88         }
89     }

```

Figure 40-#st if branch

```

91         } else if (input == "#load") { //if load is entered
92             cout<<"Please enter the file name , make sure it is in the folder CompilersAssignment/TestPrograms"<<endl;
93             cout<<"MLI> ";
94             getline(cin,input); // get file name
95             lexer->initialize_input_characters("../TestPrograms/"+input); //initialize characters with file name
96
97             /* This was used to test the lexer only , to view test , just remove the comments
98             auto* token = lexer->getNextToken();
99             while(token->getTokenName() != Token::TOK_EOF){
100                 cout<<token->getTokenName()<<" ";
101                 token = lexer->getNextToken();
102             } */
103
104             /* This was used to test the parser and xml generator only , to view test , just remove the comments*/
105             //xml->visitTree(parser->parse());
106
107             interpreterExecutor->visitTree(parser->parse()); // create parse tree and call interpreter on it
108             lexer->clearCharactersContainer(); // clear characters to start from the beginning next time round
109             lexer->restartCurrentInputIndex(); // restart pointer from beginning
110             parser->resetAST(); // empty AST to start from the beginning next time round

```

Figure 41- #load branch

As shown in figure 41 above, if a #load is entered, then the user is asked to enter the file name to be used as source program to the compiler. It is important to note that files in the folder CompilersAssignment/TestPrograms are accepted. Thus, as seen in line 95 then the lexer's input character stream is initialized. Afterwards, as shown in line 107 the interpreter is executed on the parser's generated parse tree. An important step is as shown in lines 108-110 where the lexer and parser's state are reset. This is important so that programs could be analyzed separately, otherwise there would be analysis of the already analyzed programs since the parse tree would contain programs that have already been analyzed.

```

111         } else if (input == "#exit") {
112             break; //if exit is entered break from the loop
113         } else { //if another input is entered
114             lexer->initializeCharactersWithString(input); //initialize
115             interpreterExecutor->visitTree(parser->parse()); // create
116             lexer->clearCharactersContainer(); //clear characters to st
117             lexer->restartCurrentInputIndex(); // restart pointer from
118             parser->resetAST(); // empty AST to start from the beginn
119         }

```

Figure 42-#exit and any statement input

As shown in figure 42 above, if the user enters "#exit" then the REPL stops execution. On the other hand, if he enters a valid statement then that statement is treated as a valid program on its own and thus the lexer, parser and interpreter are executed again on that statement as shown in lines 114-115. As described in the previous paragraph, the state of the parser and lexer needed to be reset.

```

120         } catch (CompilingErrorException &e) { //if any compiling exception is met
121             cout << e.getErrorMessage() << endl; // display error message and restart from initial state ,
122             lexer->clearCharactersContainer();
123             lexer->restartCurrentInputIndex();
124             parser->resetAST();
125             auto * previousScope = new vector<SymbolTable*>();
126             for(unsigned int i=0;i<scopedTemp.size();i++){ //get previous state , this works since in the i
127                 // we are always in the global scope.
128                 if(interpreterExecutor->getValidator()->getScopedTable().at(i)->getMultimap().size() ==
129                     scopedTemp.at(i).getMultimap().size()){
130                     previousScope->push_back(interpreterExecutor->getValidator()->getScopedTable().at(i));
131                 }else{
132                     tempValue=scopedTemp.at(i);
133                     previousScope->push_back(&tempValue);
134                 }
135             }
136             interpreterExecutor->getValidator()->setScopedTable(*previousScope);
137         }
138         scopedTemp.clear();

```

Figure 43-Handling Exception

If a particular statement is not according to the EBNF in the assignment specification, the program throws a `CompilingErrorException`. In this case the lexer and parser state are reset as shown in figure 43 lines 122-124 so that after the REPL handles the exception the REPL can take more statements are programs as input. Then as shown in lines 126-136, the symbol table is checked to see if the loaded program had invalidated the symbol table by adding a function that caused an exception. If this is the case, the previous state is retrieved by the REPL.

In this implementation the REPL was designed to either take large programs using the load function, or single line statements using the interactive console. This was done in this way in order to keep the REPL as simple as possible. It is also important to note that in this implementation expressions cannot be taken as statements. For example, “3+4” is an invalid statement. In order for this example to be valid the user must enter “print 3+4”. This was implemented in this way because the ebnf does not specify expressions as statements. It is important to note that the “it” variable was not implemented in this REPL because the design of the `TypeBinder` would make it difficult to include it.

## Test Cases and Results

### Test Case 1: Testing the #help function

**Input Program: #help in console**

**Expected Result: Help menu**

**Actual Result:**

```
Minilang Interpreter running ,enter #help for command breakdown
MLI> #help
Any of the following commands makes the Interpreter act :
1.Minilang commands to be executed
2.Enter #st to display symbol table
3.Enter #help for help
4.Enter #exit to exit interpreter
5.Enter #load to load a text file. NOTE TEXT FILE MUST BE IN CompilersAssignment/TestPrograms
6.Any command that does not fit the above will be declared as invalid
7.Note white spaces and empty statements are ignored , so you have to enter something
```

*Figure 44-Actual Result for test case 1*

**Test Outcome: Success**

**Test Case 2: Testing the #exit feature**

**Input Program: #exit in console**

**Expected Result: Goodbye message and exit**

**Actual Result:**

```
MLI> #exit
Bye :)
```

*Figure 45-Actual Result for test case 3*

**Test Outcome: Success**

**Test Case 3: Testing #st with no variables and functions entered**

**Input Program: #st in console**

**Expected Result: no display**

**Actual Result: no display**

**Test Outcome: Success**

#### Test Case 4: Testing the #load feature

Input Program: #load + Factorial.txt in console

Expected Result: 120

Actual Result:

```
Minilang Interpreter running ,enter #help for command breakdown
MLI> #st
MLI> #load
Please enter the file name , make sure it is in the folder CompilersAssignment/TestPrograms
MLI> Factorial
120
```

*Figure 46-Actual Result for test case 4*

**Test Outcome: Success**

#### Test Case 5: Testing #st with Factorial function declared and variable declared

Input Program: #st in console

Expected Result:

```
1.a variable int 5
2.fact function int
```

Actual Result:

```
Minilang Interpreter running ,enter #help for command breakdown
MLI> #st
MLI> #load
Please enter the file name , make sure it is in the folder CompilersAssignment/TestPrograms
MLI> Factorial
120
MLI> var a : int =5;
MLI> #st
1.a variable int 5
2.fact function int
```

*Figure 47-Actual Result for test case 5*

**Test Outcome: Success**



### Test Case 6: Testing #st after program results into exception

**Input Program:** #st in console after bad program semantics, along with this erroneous program:

```
var a : int =2;
while(a>0){
    while(a>0){
        var a : int = 2;
        set a = a-1;
        return 4;
    }
    var a : int = 2;
    set a = a-1;
}

def func1():int{
    return 2+2;
}

var func : int =2;

def func():int{
    return 2+2.0;
}
```

**Expected Result:**

```
1.b variable int 4
2.fact function int
```

**Actual Result:**

```
Minilang Interpreter running ,enter #help for command breakdown
MLI> #load
Please enter the file name , make sure it is in the folder CompilersAssignment/TestPrograms
MLI> Factorial
120
MLI> var b : int =4;
MLI> #load
Please enter the file name , make sure it is in the folder CompilersAssignment/TestPrograms
MLI> Test2
Binary operators can only be applied to expressions of the same type
MLI> #st
1.b variable int 4
2.fact function int
```

*Figure 48-Actual Result for test case 8*

**Test Outcome: Success**

## Conclusion

Thus, it was concluded that the compiler for MINILANG was implemented successfully as per requirements. However as described in each of the individual sections, there were a bit of assumptions taken. Firstly, in the Lexer implementation, it was assumed that the lexer does not produce comments and whitespace tokens since these were deemed irrelevant for the parser. Secondly the parser was assumed to have no nodes for the operators and parameters since these were treated as type classes. Thirdly, in the semantic analyzer it was assumed that the visitor class does not allow function overloading. This was done in this way because it was not specified in the assignment specification sheet. Fourthly in the REPL it was also assumed that the REPL only takes long programs as input using the #load function. This was done in this way because it was not specified in the assignment specification sheet.

Further improvements could have been done to the artifact by improving the REPL. The REPL could have an “it” variable that stores the last answer computed by the interpreter executor pass. This was not done since the Type Binder design would make it difficult to include such feature. Another improvement that could have been made was to accept expressions as input in the REPL, however this was not done since the EBNF specification states otherwise.

## References

- [1] "What is a Compiler? - Definition from Techopedia", Techopedia.com. [Online]. Available: <https://www.techopedia.com/definition/3912/compiler>.
- [2] "Operating Systems Notes", Personal.kent.edu. [Online]. Available <http://www.personal.kent.edu/~rmuhamma/Compilers/MyCompiler/phase.htm>.
- [3] "Compiler Design - Semantic Analysis", www.tutorialspoint.com. [Online]. Available: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_semantic\\_analysis.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm).
- [4] A. Ahmed, "direct-coded vs table-driven lexer?", Stackoverflow.com, 2015. [Online]. Available: <https://stackoverflow.com/questions/27763544/direct-coded-vs-table-driven-lexer>.
- [5] "Compiler Design - Top-Down Parser", www.tutorialspoint.com. [Online]. Available: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_top\\_down\\_parser.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm).
- [6] "Design Patterns and Refactoring", Sourcemaking.com. [Online]. Available: [https://sourcemaking.com/design\\_patterns/visitor](https://sourcemaking.com/design_patterns/visitor).