Compiler Theory and Practice

Course Assignment 2017/2018

cps2000

Dylan Galea |84296M | 21/04/2018

# Table of Contents

# 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 constitute 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 notice 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 7 main sections, where for the first 6 sections, 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. The final section illustrates some improvements that could have been made and any limitations in the software. 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 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).

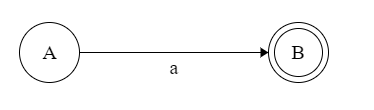


Figure 1-Automaton example

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):

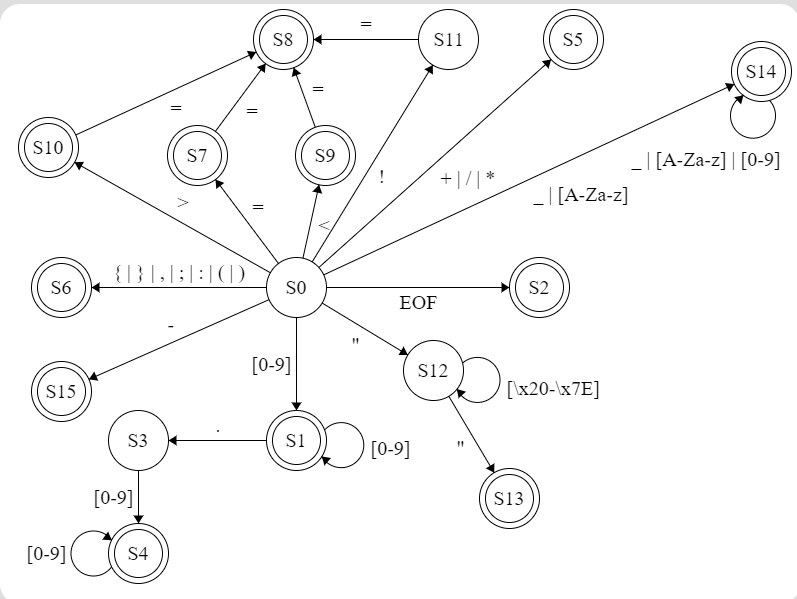


Figure 2- Finite State automaton for language

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:

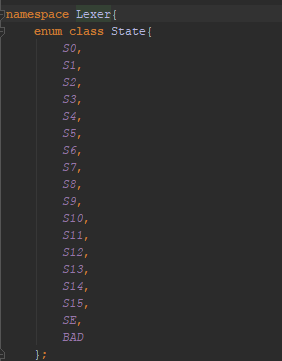


Figure 3-Enum class state

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 a state SE is returned if the lexer scans the input #, 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, were 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 types 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, were 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):

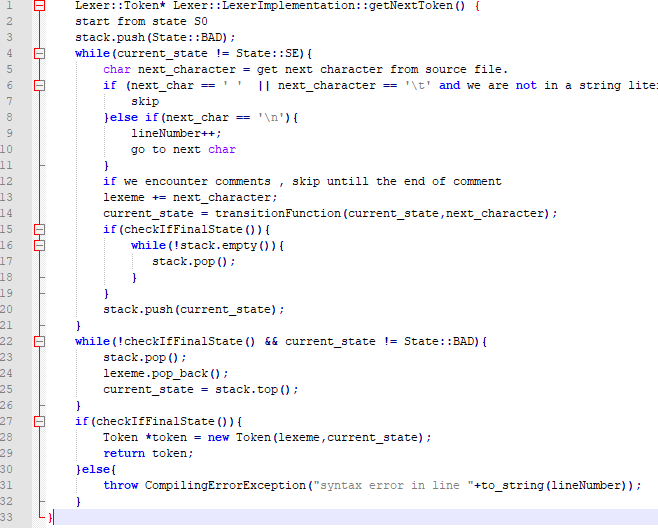


Figure 5-Lexer Pseudocode

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.:

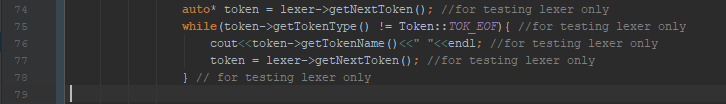


Figure 6-invocation in main class

**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 depend 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 had to be designed for the AST. 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 were in this class only 1 method is declared. This method is the virtual accept method that will be used for the 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.

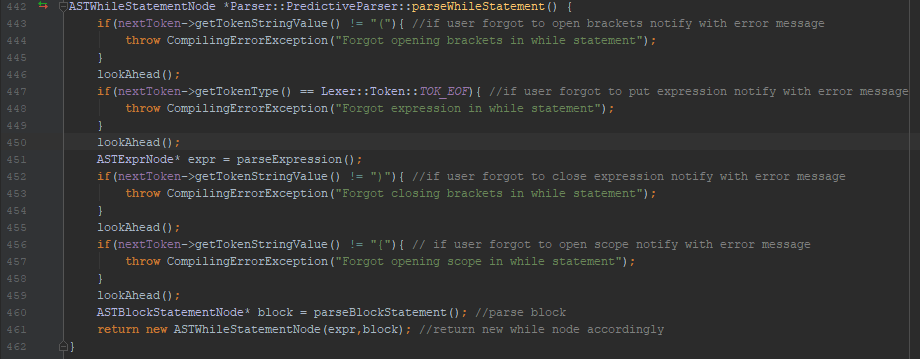
The next class that needed to be designed was the PredictiveParser class. The PredictiveParser class contains a lexer, currentToken, nextToken and a vector that stores pointers of type ASTStatementNode as fields. The lexer is used so that the parser always gets the next token that is scanned from the 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.

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, and 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 node 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. Thus t

### 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 parse tree 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 is 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.

****

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:**

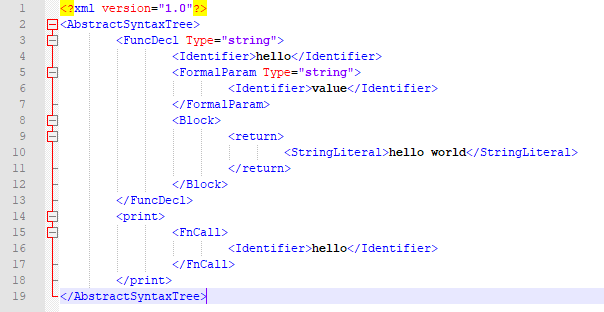
****

Figure 11-AST for test case 1

**Actual Result:**

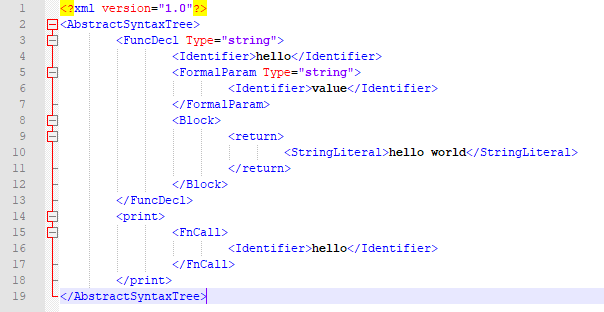
****

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:**

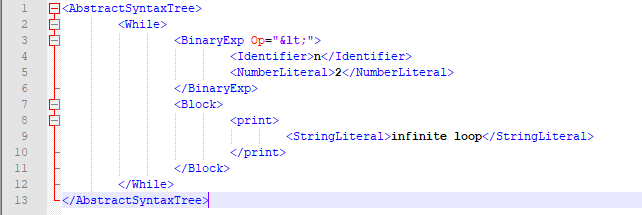


Figure 13-AST for test case 2

**Actual Result:**

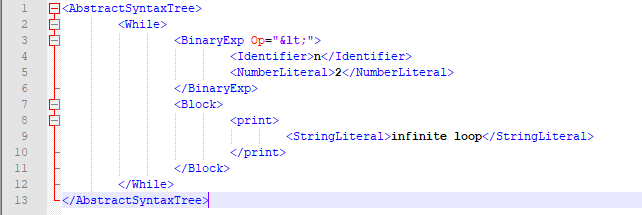


Figure 14-Result for test case 2

**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:**

****

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:**

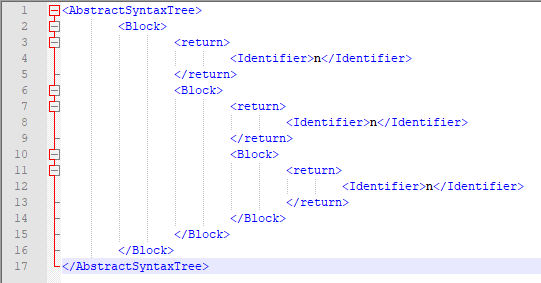


Figure 13- Expected Result for test case 5

**Actual Result:**

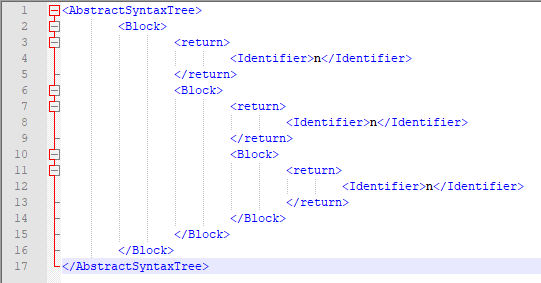


Figure 14-Actual Result For test case 4

**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

# 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>.

[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>.