LL(1) Parser

A Mini Project Report Submitted by

Rahul D Shetty 4NM16CS111
Saurabh D Rao 4NM16CS132

UNDER THE GUIDANCE OF

Mrs. ANISHA P RODRIGUES

Assistant Professor Gd. II

Department of Computer Science and Engineering

in partial fulfilment of the requirements for the award of the Degree of

Bachelor of Engineering in Computer Science & Engineering

from

Visvesvaraya Technological University, Belgaum

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

N.M.A.M. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU, Belgaum) (AICTE approved, NBA Accredited, ISO 9001:2008 Certified) NITTE -574 110, Udupi District, KARNATAKA.

April 2019

NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India
(ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC
☎: 08258 - 281039 – 281263, Fax: 08258 – 281265
Department of Computer Science and Engineering
B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

LL(1) Parser

is bona fide work carried out by

Rahul D Shetty 4NM16CS111
Saurabh D Rao 4NM16CS132

in partial fulfilment of the requirements for the award of
Bachelor of Engineering Degree in Computer Science and Engineering
prescribed by Visvesvaraya Technological University,
Belgaum during the year 2018-2019.

It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report.

The Mini project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree.

Signature of Guide Signature of HOD

ACKNOWLEDGEMENT

We believe that our project will be complete only after we thank the people who have contributed to make this project successful.

First and foremost, our sincere thanks to our beloved principal, **Dr. Niranjan N. Chiplunkar** for giving us an opportunity to carry out our project work at our college and providing us with all the needed facilities.

We express our deep sense of gratitude and indebtedness to our guide **Mrs. Anisha P Rodrigues**, Assistant Professor Gd II, Department of Computer Science and Engineering, for his inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.

We sincerely thank **Dr. K.R. Udaya Kumar Reddy**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyantaya Memorial Institute of Technology, Nitte.

We also thank all those who have supported us throughout the entire duration of our project.

Finally, we thank the staff members of the Department of Computer Science and Engineering and all our friends for their honest opinions and suggestions throughout the course of our project.

Rahul D Shetty 4NM16CS111
Saurabh D Rao 4NM16CS132

ABSTRACT

The purpose of this project is to design lexical analyser and syntax analyser for a LL(1) Grammar. The two stages are the integral part of Analysis phase of a compilation process which involves identifying the tokens of the given program and using these tokens to identify if each of them are syntactically proper based on given production rules. The main program takes in two inputs namely the source program which we need to process and the grammar rules to parse the program. The objective of the project is to generate the parsed sequence which can be further given for the later stages of the compiler.

The grammar that is defined for parsing, should be LL(1) that is to say it should not contain any left recursion and it should be left factored. By using the LL(1) productions, we generate the parse table which has entries for each terminals and non-terminals identified in them. Before the generation of parse table, we identified the FIRST and FOLLOW's of each terminals using a recursive method. The final stage is the parsing which is done by using the standard LL(1) parsing steps. If the given source code contains some syntax errors, the appropriate line number would be shown. The error handling part of the parser is implemented using Panic Mode recovery.

The outcome of the project is to identify the parsing actions taken by the grammar for proper and invalid source code. Along with this we are trying to present the Symbol Table for variable declarations in the entire code.

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 Compiler

We know that computer is a logical assembly of both hardware and software. The hardware consists of all the physical components interconnected to function as needed and the software is used to control and manage the software. But when we look into the actual implementation we have the basic blocks which work by using Low and High Voltages and all the basic blocks connected in a particular manner to do different operations. As by using the software we can assign a low and high voltages using 0's and 1's. In a nutshell the computer can only understand 0's and 1's that are given to it. The programs written in this format is known as a machine code. Therefore, we have each instruction in terms of an equivalent binary code representation.

For any novice to expert programmer it is difficult to remember all the code equivalents and programming them in a computer would get quite complex. In order to overcome this problem, we have software programs called "Compiler" whose task is to convert a high lever language code that is easy to understand by humans to a machine code which can be executed. There are many reasons to use a high level language specification when implementing a code, few of them are:

1. High level languages are easier for a human being to understand.
2. Modifying or updating the code becomes easier providing flexibility.
3. Debugging the faulty code is easier in compilers as we have a general rule to define each and every instruction. Some compilers provide with error handling techniques and also provide a detailed description about errors.
4. The programs written in these languages are shorter when compared to those written in machine code.

There are few drawbacks in using the high level language for programming, one of them is that the compilation process could take up a lot of time and it will increase with complexity of the programs.
In some of the programming languages like C, we can access the primary memory by making use of pointers. These types of access are generally not safe and the programmer should be careful while using them.

1.2 Phases in Compiler

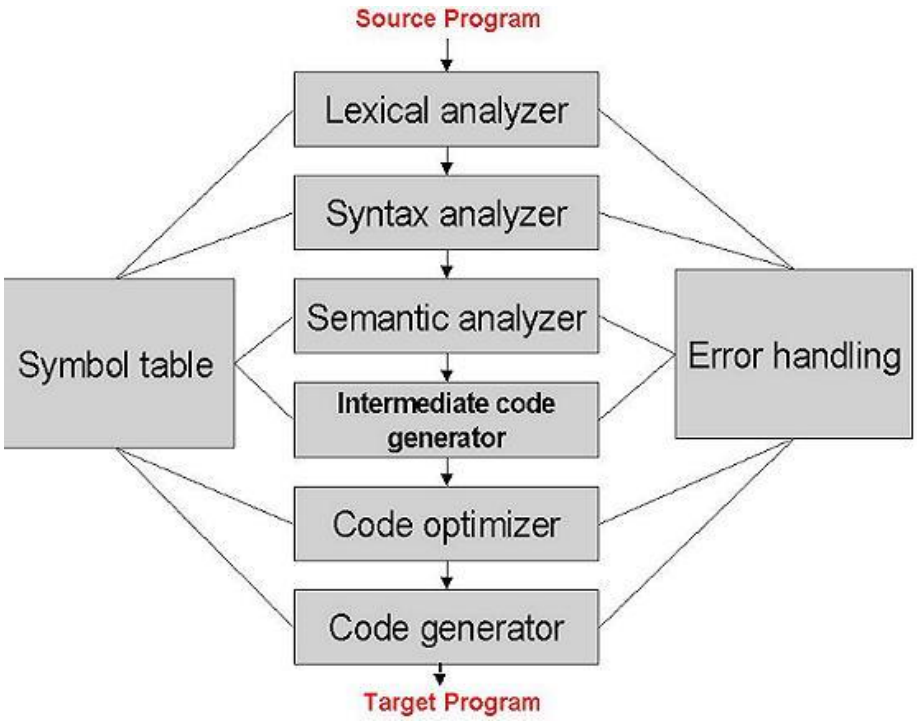

Figure 1: Phases of Compiler

There are mainly two important phases in a compilation process Analysis and Synthesis Phase.

Analysis Phase is concerned with identifying tokens, syntactical meaning, semantic meaning and generating a parse tree for the next phase. This is also known as the Frontend of the compiler since we are just working on the program code but not on generation any machine related code. This phase is further categorized as Lexical Analysis, Syntax Analysis and Semantic Analysis. Figure 1 represents all the different phases in the compiler.

- Lexical Analysis: This is the initial stage of compilation process which involves identifying all the tokens for the given source code. The rules for the tokens are predefined in the compiler. Other than identifying the tokens, this phase is also removes any comments while parsing and also identifiers the line number for each token in case of errors.

- Syntax Analysis: This phase reads in the source code by taking a token at a time. A CFG grammar is defined for the syntax rules and the parser checks the source code against these rules. If a successful derivation of the source code is possible from the available productions, then the program is said to be successfully parsed.

- Semantic Analysis: The last phase of the Analysis phase which includes type checking and conversions. Along with that it produces the final Syntax Tree which is given to the next phase for code generation.

Synthesis Phase is said to be the backend of the compiler which generates the final machine code which actually is dependent on the target machine we want to run the code and hence the name. The input to the phase is an intermediate representation of the program which is later converted to another intermediate form which is suitable to generate the final output. There are 3 main sub phases in this part namely:

- Intermediate Code Generation: The initial phase of Synthesis phase which mainly transforms the syntax tree to another intermediate code which is suitable for converting to the machine code. Some of the intermediate representations are Three Address Code Format, Post Fix Notation, Directed Acyclic Graph, Syntax Tree and so on.

- Code Optimizer: The Intermediate code generally contains set of repeating instructions which could be further optimized and reduced to save space and time while execution.

- Final Code Generation: The final and important stage of compilation where we generate the final code which can be used to execute. Some compilers convert the code to assembly language code and that is converted to executable when needed. Since the instruction set of machines differ from each other, it is important to generate the proper machine code for a particular system by taking note of this.

Some compilers have error handling mechanism so that the compilation process doesn't halt in between. For this purpose, we have an Error Handler whose main task is to correlate the appropriate errors and continue with compiling the further codes. There are many error handling strategies involved in different stages of compiler. In Lexical analyser phase we have a Panic mode recovery which skips next characters until a proper token is found. This scheme is implemented in this project.

Symbol Table Manager is used to keep track of different variables used, functions and various parameters of each such as datatype of variable, size or capacity for storage and scope of the variables in programs. These types of information are very important for the compiler as there could problems like Redeclaraction of variables, accessing variables which are not in the scope of a particular program section and so on.

1.3 Lexical Analyser

Lexical Analyser is initial stage of compilation which involves identifying various tokens present in the program. Hence we need to define the rules to identify each and every tokens.

A Token is smallest unit of the program which contains sequence of characters. Some of the tokens could be Keywords (if, else, for, while etc.), Operators (+, -, *, / etc.), Identifier Names and so on. In some languages whitespaces (tab, spaces) are usually considered while tokenizing but they are ignored as they are just used to separate different tokens.

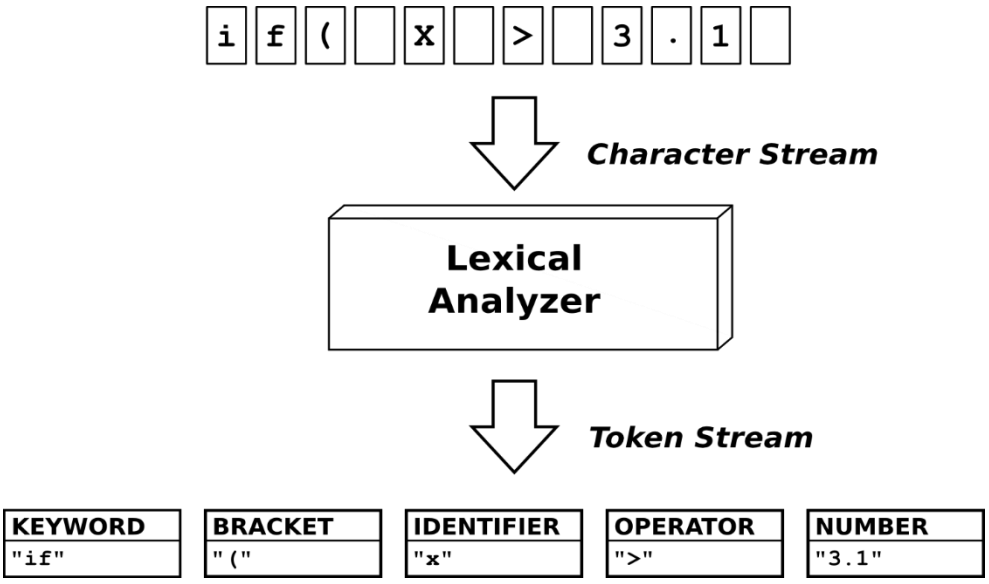

Figure 2: Lexical Analysis

The above Figure 2 shows how a general Lexical Analyser works. It takes in series of characters in the input buffer and tries to identify or group them into Tokens. The syntax analysis phase takes in each of tokens one at a time. If any characters do not match the token rules, then we produce in an error message and continue with processing by using some error recovery scheme which is done by Error Handler. The process of lexical analysis is also termed as Scanning because we are trying to scan the character sequences from the code and try to identify the valid tokens.

1.4 Syntax Analyser

Syntax Analyser takes in input as token by token and then tries to match it with a production rule and derive the parsing steps. The production rules or the grammar rules are defined by a special set of grammar known as Context Free Grammar (CFG).

A CFG is defined by a 4 tuple system containing the following terms:

- Terminals: Finite set of symbols or tokens which are the basic unit of the grammar.
- Non-Terminals: Finite set of syntactic variables that denotes a set of strings.
- Productions: There are the rules which are of the form A - > B where A is the production head and B is the body.
- Starting Symbol: All the parsing actions taken from an initial non-terminal called as the Starting symbol.

The above grammar is used for defining the productions of any language which can be used to check the syntax meaning of the program. If the given CFG is unable to produce a particular string or source code, then we say that the input source does not belong the language of CFG. The language defines the set of all possible valid strings accepted by the CFG.

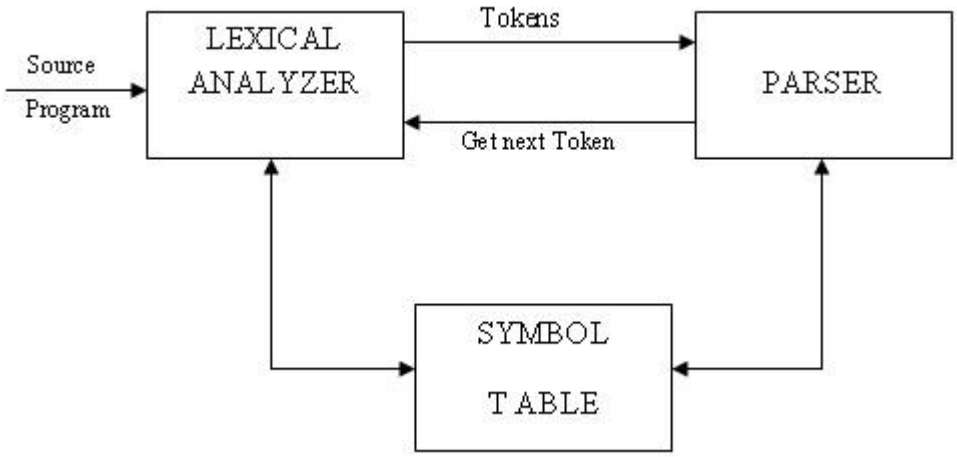

Figure 3: Working of Parser

In the Figure 3 the parser requests for token by issuing *GetNextToken*( ) method from the Lexical Analyser. In scanning and parsing phase the symbol table is updated to add the information about various variables, constants, objects, functions and so on.

Many programmers make mistakes while writing the grammar code according to the syntax and to manage that we have an error handler which based on particular scheme continues with parsing and also shows the developers with the appropriate error message.

Mainly there are two types of parser, Top-Down and Bottom-Up Parsers.

A Top-Down parser starts parsing the program code from starting symbol and generates until the leaf nodes or tokens. LL(k) Parser uses a Top-Down based approach for parsing. These are further classified as Recursive Descent and Predictive Parser.

A Bottom-Up parser on the other hand starts at the leaf nodes and continues parsing until we derive the starting symbol of the grammar. A commonly used bottom up parsers are LR(*k*) parsers which scans the input string from left to right and so is the first 'L', while 'R' in LR stands for Right most derivation in reverse. *k* usually refers to the number of look ahead symbols. Simple LR, Canonical LR and Look Ahead LR are the most popular Bottom-up parsers.

1.5 Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in structured format.
- To verify the declaration of variables.
- To implement type checking, type casting.
- To determine the scope of any entity.

A symbol table can be implemented either by a linear list or by making use of Hash Table or a Tree data structure could be used.

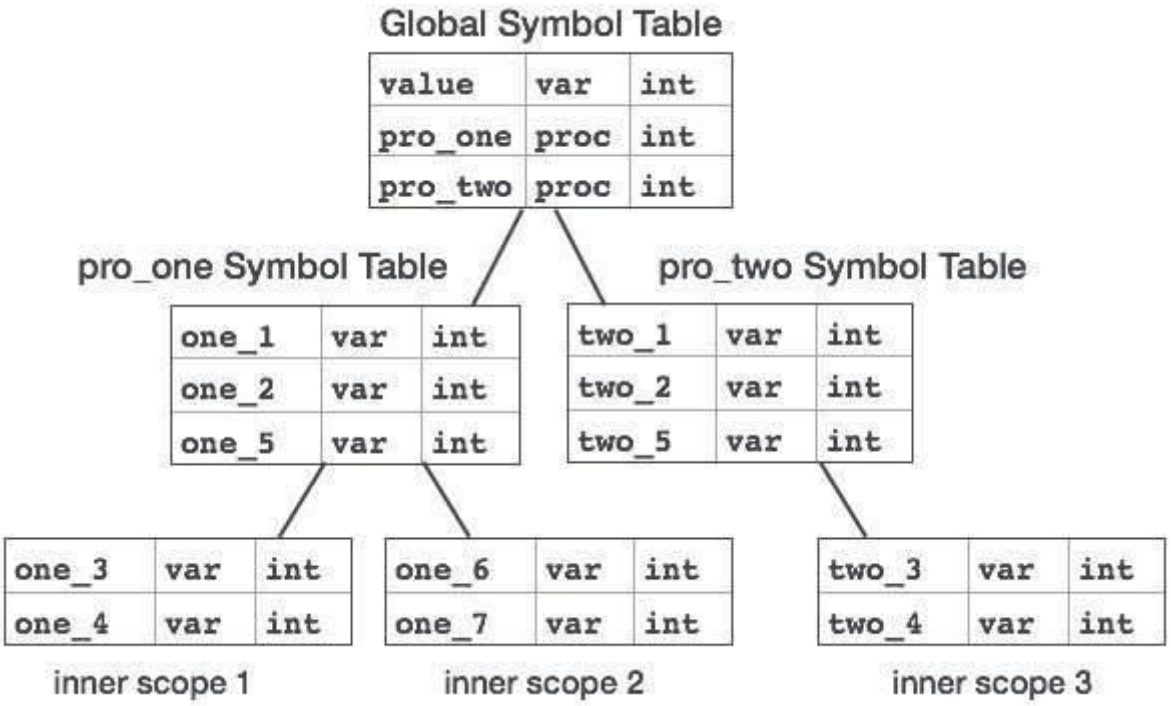

Figure 4: Tree structured Symbol Table

Figure 4 shows an example of Symbol Table represented in a tree structure. Each symbol table is defined for a scope and procedures. Each of the entry represents the information about variables present in that scope.

CHAPTER 2

IMPLEMENTATION

2.1 LL(1) Parser

An LL parser is called an LL(*k*) parser if it uses *k* tokens of look ahead when parsing a sentence. A grammar is called an LL(*k*) grammar if an LL(*k*) parser can be constructed from it. The output of these parsers are sequence of steps or derivations which are the same as left most derivation of that string from the rules. The first L in the LL(*k*) stands for Left to Right input scanning and the second L stands for Left most derivation output. The *k* refers to the number of look ahead symbols that are considered.

These classes of parsers are easy to design because of its simplicity. It makes use of a table based approach while performing the parsing actions. The parser is a deterministic pushdown automaton with the ability to peek on the next input symbols without reading. This capability can be emulated by storing the look ahead buffer contents in the finite state space, since both buffer and input alphabet are finite in size. As a result, this does not make the automaton more powerful, but is a convenient abstraction.

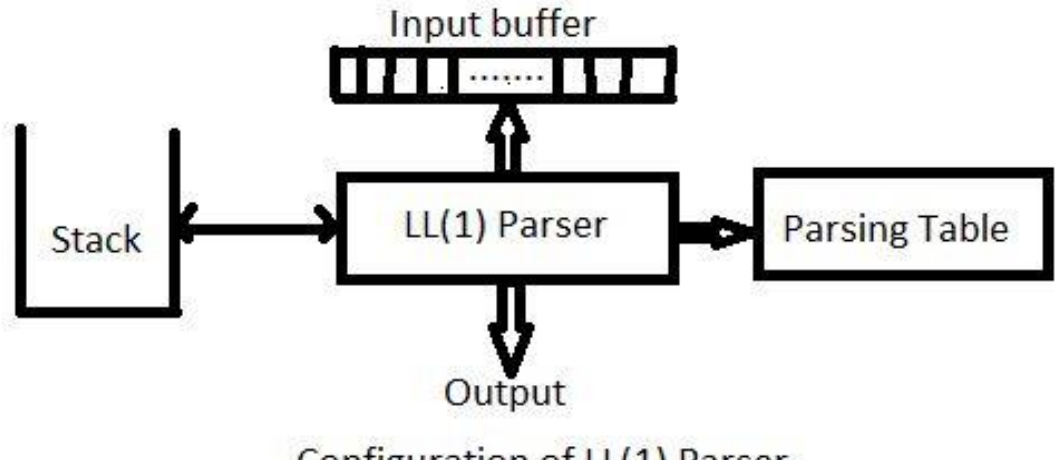

Figure 5: LL(1) parser

The input to be parsed is stored in the input buffer and LL(1) parser makes use of stack and parsing table to make decision on which production to apply for the current symbol of the input. Figure 5 shows the general configuration of how a LL(1) parser works.

Unlike to LR grammars, LL(1) grammar have a constraint and cannot parser for all the productions. The given grammar rules must be left factored and we need to remove any left recursions if present. For the above two part we could write a method just to pre-process the grammar. But again if the grammar is inherently unambiguous then the grammar can't be used for parsing. LL(1) is a predictive parser where it uses 1 look ahead symbol to make decision on which production rule to apply.

The given project is built completely on Pure Python without using any external modules or libraries. The project is divides into two main classes namely Lexer or Lexical Analyser and Parser or Syntax Analyser. They both are containing methods for doing their part in the compilation process.

2.2 Token Rules

The token rules are used to identify individual tokens from the source code. For the project we made use of regex which is a Regular Expression system provided by Python to identify each and every tokens present in the code.

Token Rule	Token Description	Token	Description
=	ASSIGN	<=	LE
==	EQ	<	LT
%	MOD	>=	GE
/	DIV	>	GT
*	MUL	if	IF
-	SUB	main	PGM_START
+	ADD	begin	BLOCK_START
!=	NE	end	BLOCK_END
!	NOT	printf	DISPLAY
||	OR	float	FLOAT
&&	AND	char	CHAR
(	LEFT_PARA	int	INTEGER
)	RIGHT_PARA	,	SEPERATOR
;	EOS	".*"	STRING
[a-zA-Z][a-zA-Z0-9]*	IDENTIFIER	'.?'	CHARACTER
[0-9]+	DIGITS		

The above table represents all the token rules for our grammar. Regular Expression of each of the token is used to identify which token does the given character sequence forms and the Token Description is used to store the information about token which can be later used while Parsing. By modifying the *lexer.py* file we can insert new token rules and store their description to grow the compiler capability.

2.3 Lexical Analyser

The tokens defined in the token rules section are concatenated to create a group of regex rules were only the regular expression system identifies all the tokens in one go. It tries to identify to which group does the token belong to by setting that particular token string and all others to null. Then the lexer will identify the first token which the given sequence of characters are set to.

Here we have a class called Tokenizer which contains the method *tokenize()* which takes in source code as input and generates a list which has information about a token in a format as a tuple which contains the following: *(token, token_desc, line_no)*. Here the *token* represents the lexeme, *token_desc* tells about the token information as shown in the table and *line_no* is used to identify on which line of source code does that token belongs in.

```
import re # Regular expression library
TOKENS -> “List of REGEX tokens”

TOKEN_DESC -> “List of Descriptions for each REGEX tokens”
def tokenize(code):
	tokenSet="("+")|(".join(TOKENS)+")"
	tokens=[]
	lc_no = 1
	lines=[]
	for line in code.split('\n'):
		p=re.findall(tokenSet,line)
		for ele in p:
			for item in ele:
				if item!='':
					tokens.append(item)
					lines.append(lc_no)
		lc_no+=1
	Token=[]
	for i,token in enumerate(tokens):
		if token not in TOKEN_DESC:
			if re.match(r'\".*\"',token):
				Token.append((token,'STRING',lines[i]))
			elif re.match(r'\'.?\'',token):
				Token.append((token,'CHARACTER',lines[i]))
			elif re.match("'",token):
				Token.append((token,'SINGLE_QUOTE',lines[i]))
			elif re.match('"',token):
				Token.append((token,'DOUBLE_QUOTE',lines[i]))
			elif re.match(r'[a-zA-Z][a-zA-Z0-9]*',token):
				Token.append((token,'IDENTIFIER',lines[i]))
			elif re.match(r'[0-9]+',token):
				Token.append((token,'DIGITS',lines[i]))
			else:
				Token.append((token,TOKEN_DESC[token],lines[i]))

	return Token
```

Figure 6: Lexical Analyser

Figure 6 code snippet returns a list of tokens, where each token has the description mentioned before.

2.4 Production Rules

The following are the grammar rules based on which the implementation of the parser is done.

```
S -> DATATYPE PGM_START LEFT_PARA RIGHT_PARA BLOCK_START STMTS BLOCK_END
STMTS -> STMT STMTS | #
DATATYPE -> INTEGER | FLOAT | CHAR
STMT -> CONDITION | FUNCTION EOS | DECLARATION EOS | ASSIGNMENT EOS
ASSIGNMENT -> IDENTIFIER ASSIGN E
DECLARATION -> DATATYPE STMT1
STMT1 -> LIST STMT2
STMT2 -> SEPERATOR STMT1 | #
LIST -> IDENTIFIER STMT3
STMT3 -> ASSIGN E | #
CONDITION -> IF LEFT_PARA E RIGHT_PARA BLOCK_START STMTS BLOCK_END
FUNCTION -> DISPLAY LEFT_PARA MSG RIGHT_PARA
MSG -> STRING | IDENTIFIER | CHARACTER
E -> E2 E1
E1 -> relop E2 E1 | #
E2 -> T E3
E3 -> OP1 T E3 | #
T -> F T1
T1 -> OP2 F T1 | #
OP1 -> ADD | SUB
OP2 -> MUL | DIV
F -> LEFT_PARA E RIGHT_PARA | SUB F | IDENTIFIER | DIGITS | CHARACTER
relop -> EQ | NE | AND | OR | LE | LT | GE | GT
```

Figure 7: Production Rules

In the grammar rules shown in Figure 7 we need to identify the grammar tokens and variables associated with them. This is done by taking all the head of the productions as non-terminals or variables and remaining symbols as terminals.

```
Terminals: ['CHARACTER', 'BLOCK_END', 'OR', 'STRING', 'AND', 'RIGHT_PARA', 'ADD',
'LEFT_PARA', 'NE', 'SEPERATOR', 'ASSIGN', 'MUL', 'EOS', 'BLOCK_START', 'DISPLAY',
'SUB', 'IDENTIFIER', 'EQ', 'GT', '$', 'FLOAT', 'CHAR', 'INTEGER', 'DIV', 'LE', 'GE',
'LT', 'PGM_START', 'IF', 'DIGITS']

Non-Terminals: ['E2', 'OP2', 'relop', 'OP1', 'E', 'STMTS', 'STMT2', 'F', 'DATATYPE',
'E1', 'T', 'FUNCTION', 'DECLARATION', 'T1', 'S', 'LIST', 'CONDITION', 'STMT1', 'E3',
'STMT3', 'ASSIGNMENT', 'STMT', 'MSG']
```

Figure 8: Terminals and Non-Terminals

We assume that the head of the first production in the list of productions is the Starting Symbol which in our case is “S”. After processing our grammar rules we obtain set of terminals and non-terminals that is shown in Figure 8.

2.5 FIRST and FOLLOWS:

FIRST set of a grammar is the set of starting terminals or lambda productions with which a non-terminal lead to and FOLLOW set of a grammar is the set of symbols which follow the non-terminal appearing in the right-hand side of the production. These two methods are the starting step for creating the Parse table.

Both the functions are implemented by recursion calls to already found sets. FOLLOW's of a set has a special case where it goes to an infinite loop, in order to tackle this problem the program is designed in such a way that all the function calls are noted down at each recursion tree and are blocked from further expanding after a threshold limit.

For the above grammar we have found the FIRST and FOLLOWS sets by implementing the following code below.

```
def first(symb, parser):
    if symb is '|':
        return ['#']
    if symb in parser.terminals:
        return [symb]
    elif symb is '#':
        return ['#']
    ans = []
    body = parser.prods[symb]
    found = 0
    waitKey = 0
    for item in body:
        if item == '|':
            if waitKey == 1:
                ans.append('#')
                waitKey = 0
            found = 0
            continue
        if found == 0:
            if item is '#':
                ans += ['#']
            elif item in parser.terminals:
                waitKey = 0
                ans += [item]
            else:
                subFirst = first(item, parser)
                ans += [x for x in subFirst if x != '#']
                if '#' in subFirst:
                    waitKey = 1
                    continue
            found = 1
    if waitKey == 1:
        ans.append('#')
    return list(set(ans))
```

Figure 9: First Function

```
def follow(symb,parser,startSymb=""):
    if visited[symb]==-1:
        return parser.followSet[symb]
    if visited[symb]==10:
        visited[symb]=0
        return [] if startSymb!="" else ['$']
    visited[symb]+=1
    ans=[]
    if startSymb!="":
        ans.append('$')
    for prod in parser.prods:
        body=parser.prods[prod]
        if symb in body:
            f=1
            beforeEp=0
            for item in body:
                #open('log.txt','a').write(" ".join([symb,prod,item]) + "\n" )
                if f==0:
                    if item == "#":
                        beforeEp=1
                        continue
                    elif item == "|" and beforeEp==1:
                        ans+=follow(prod,parser,startSymb if prod is startSymb else "")
                        beforeEp=0
                    elif item in parser.terminals:
                        f=1
                        beforeEp=0
                        ans+=[item]
                    elif item in parser.variables:
                        firstSet=parser.firstSet[item]
                        ans += [x for x in firstSet if x != "#"]
                        beforeEp=1
                        f=0
                        if "#" not in firstSet:
                            beforeEp=0
                            f=1


                elif item == symb:
                    f=0

            if f==0:
                ans+=follow(prod,parser,startSymb  if prod is startSymb else "")
    visited[symb]=-1
    parser.followSet[symb]=list(set(ans))
    return parser.followSet[symb]
```

Figure 10: Follow Function

Figure 9 shows the code snippet to find the FIRST set of a non-terminal. The *first( )* is implemented by calling the function in recursive manner for each variable symbols. This returns back a list of terminals along with epsilon (#) if they are present in the FIRST sets. Figure 10 shows the code snippet find the FOLLOW set of a non-terminal. The *follow( )* procedure is again implemented in recursive way but here there is a chance of the productions defined could go on an infinite loop, a precaution is taken to call the *follow( )* procedure a given amount of times.

After applying the above methods on to the given grammar we obtain the following FIRST and FOLLOW Set for different terminals. Figure 11 represent the FIRST set and Figure 12 represents the FOLLOW set.

```
First Set:
S:  ['FLOAT', 'CHAR', 'INTEGER']
STMTS:  ['INTEGER', '#', 'DISPLAY', 'FLOAT', 'IDENTIFIER', 'IF', 'CHAR']
DATATYPE:  ['FLOAT', 'INTEGER', 'CHAR']
STMT:  ['INTEGER', 'DISPLAY', 'FLOAT', 'IDENTIFIER', 'IF', 'CHAR']
ASSIGNMENT:  ['IDENTIFIER']
DECLARATION:  ['FLOAT', 'CHAR', 'INTEGER']
STMT1:  ['IDENTIFIER']
STMT2:  ['#', 'SEPERATOR']
LIST:  ['IDENTIFIER']
STMT3:  ['#', 'ASSIGN']
CONDITION:  ['IF']
FUNCTION:  ['DISPLAY']
MSG:  ['CHARACTER', 'IDENTIFIER', 'STRING']
E:  ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
E1:  ['#', 'OR', 'AND', 'LE', 'GE', 'NE', 'LT', 'EQ', 'GT']
E2:  ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
E3:  ['SUB', 'ADD', '#']
T:  ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
T1:  ['#', 'DIV', 'MUL']
OP1:  ['SUB', 'ADD']
OP2:  ['DIV', 'MUL']
F:  ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
relop: ['OR', 'AND', 'LE', 'GE', 'NE', 'LT', 'EQ', 'GT']
```

Figure 11: First set

```
Follow Set:
S:  ['$']
STMTS:  ['$', 'BLOCK_END']
DATATYPE:  ['IDENTIFIER', 'PGM_START']
STMT:  ['INTEGER', 'BLOCK_END', 'DISPLAY', '$', 'FLOAT', 'IDENTIFIER', 'IF', 'CHAR']
ASSIGNMENT:  ['EOS']
DECLARATION:  ['EOS']
STMT1:  ['$', 'EOS']
STMT2:  ['EOS', '$']
LIST:  ['EOS', '$', 'SEPERATOR']
STMT3:  ['$', 'EOS', 'SEPERATOR']
CONDITION:  ['DISPLAY']
FUNCTION:  ['EOS']
MSG:  ['RIGHT_PARA']
E: ['$', 'EOS', 'SEPERATOR', 'RIGHT_PARA']
E1:  ['EOS', 'RIGHT_PARA', '$', 'SEPERATOR']
E2:  ['EOS', 'OR', 'AND', 'RIGHT_PARA', 'LE', 'GE', '$', 'NE', 'LT', 'EQ', 'GT',
'SEPERATOR']
E3:  ['EOS', 'OR', 'AND', 'RIGHT_PARA', 'LE', 'GE', '$', 'NE', 'LT', 'EQ', 'GT',
'SEPERATOR']
```

```
T:  ['EOS', 'OR', 'SUB', 'ADD', 'AND', 'RIGHT_PARA', 'LE', 'GE', '$', 'NE', 'LT',
'EQ', 'GT', 'SEPERATOR']
T1:  ['EOS', 'OR', 'SUB', 'ADD', 'AND', 'RIGHT_PARA', 'LE', 'GE', '$', 'NE', 'LT',
'EQ', 'GT', 'SEPERATOR']
OP1:  ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
OP2:  ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
F:  ['EOS', 'OR', 'SUB', 'DIV', 'ADD', 'AND', 'RIGHT_PARA', 'LE', 'GE', '$', 'NE',
'LT', 'IDENTIFIER', 'EQ', 'GT', 'SEPERATOR', 'MUL']
relop: ['CHARACTER', 'SUB', 'LEFT_PARA', 'IDENTIFIER', 'DIGITS']
```

Figure 12: Follow Set

2.6 Parse Table

In order to parse using LL(1) we require a parse table which contains information about which action to take for a particular input string symbol. The below code makes use of FIRST and FOLLOW methods to generate the parse table for the given constructs specified in the rules. All the information about terminals, non-terminals, FIRST sets, FOLLOW sets and the entire parse table information is stored in a log file which can be later reviewed.

```
class Parser:

    def __init__(self, code):
        self.code = code

    def createEmptyTable(self):
        self.variables = []
        self.terminals = ["$"]
        # Get the variables
        for var in self.prods.keys():
            self.variables.append(var)
            visited[var] = 0
        self.variables = list(set(self.variables))
        # get the teminals
        for (key, value) in self.prods.items():
            for item in value:
                if item not in self.variables and item != '|' and item \
                    != '#':
                    self.terminals.append(item)

        self.terminals = list(set(self.terminals))
        self.table = []
        # Each row is for one variable
        for e in self.variables:
            self.table.append([])
        # add columns for each row
        for r in range(len(self.variables)):
            for c in range(len(self.terminals)):
                self.table[r].append([])

    def parseProduction(self, code):
        # Production of the form A -> B | A ;
        (head, body) = code.split('->')
        head = head.strip()
        body = [x.strip() for x in body.split()]
        return (head, body)
```

Figure 13: Parser class

A class *Parser* defined in Figure 13 is used to define the process of creation of parsing table. It contains various pre and post processing methods to format the grammar and product the parsing table with proper entries for both valid and invalid cases. The code in *processProductions*() defined in Figure 14 is used to identify set of first and follow symbols by scanning the grammar production and by identifying all the variables and filling these details into the class.

```
def processProductions(self):

     # find each productions by using ; as splitter

     prods = [x.strip() for x in self.code.split('\n')]
     prods = [x for x in prods if len(x) != 0]
     prods = [self.parseProduction(x) for x in prods]

     prodsD = {}
     for item in prods:
         head = item[0]
         body = item[1]
         prodsD[head] = body
     self.prods = prodsD

     self.createEmptyTable()
     ff=open('logparse.txt','w')
     ff.write('First Set:\n')
     firstSet = {}
    f = 0
     for prod in prodsD:
         if f == 0:
             firstSet[prod] = first(prod, self)
         else:
             firstSet[prod] = first(prod, self)
         ff.write(prod+":  " + str(firstSet[prod])+"\n")

     self.firstSet = firstSet
     ff.write('\nFollow Set:\n')
     self.followSet = {}
     f = 0
     for prod in prodsD:
         if f == 0 and prod == prods[0][0]:
             self.followSet[prod] = follow(prod, self, prods[0][0])
             f = 1
         else:
             self.followSet[prod] = follow(prod, self,"")
         ff.write(prod+":  " + str(self.followSet[prod])+"\n")

     for prod in self.prods:
        self.postProcessTable(prod,self.prods[prod])

     for prodIndex,prodH in enumerate(self.followSet):
             prodIndex=self.variables.index(prodH)
             followset = self.followSet[prodH]
             for i in followset:
                 if len(self.table[prodIndex][self.terminals.index(i)])==0:
                     self.table[prodIndex][self.terminals.index(i)]=['sync']

     ff.write('\nTerminals:'+str(self.terminals)+"\n")
     ff.write('\nNon-Terminals:'+str(self.variables)+"\n")
     ff.write("\nParsing Table:\n")
     for row in range(len(self.table)):
        ff.write( self.variables[row] +" :   " + str(self.table[row])+"\n\n")
```

Figure 14: processProductions Procedure

The *postProcessTable*() method shown in Figure 15, fills the empty parse table with entries.

```
def postProcessTable(self,head,production):
     subprods=[]
     t=[]
     for i in production:
           if i!="|":
                 t.append(i)
           else:
                 subprods.append(t)
                 t=[]
     if len(t)!=0:
         subprods.append(t)
     for body in subprods:
         tempFirst=[]
         found=0
         for item in body:
             if item in self.terminals:
                tempFirst.append(item)
                found=0
                break
             else:
                 if item is "#":
                       found=1
                       continue
                 first=self.firstSet[item]
                 tempFirst+=[x for x in first if x!="#"]
                 if "#" in first:
                     found=1
                     continue
                 else:
                       found=0
                       break
         if found==1:
             tempFirst+=self.followSet[head]
         varIndex=self.variables.index(head)
         for term in tempFirst:
            termIndex=self.terminals.index(term)
                          self.table[varIndex][termIndex]+=body
```

Figure 15: postProcessTable procedure

2.7 Parser

The below algorithm is a straightforward implementation of LL(1) parser which is used to parse the given sequence of tokens into grammatical sentence. Panic mode recovery scheme is used for handling the error in parsing stage so that the parsing process doesn't halt in between. The error handler shows the line number information about where the error occurred. The parser recovers back from the error and continues parsing till the end.

Parser shows output which contains a table with 3 fields namely Stack Top, Current input symbol and Action taken for that symbol. If the parse successfully parses the entire source code, then a "Valid input" message is show otherwise an appropriate error details and presented to the programmer. Figure 19 shows the code snippet which is used by the LL(1) parser to parse the given input string.

```
from prettytable import PrettyTable

def parse(inp,startSymbol,nonTerminals,terminals,parsingTable,symbolTable):
    inp.append(("$","$"))
    stack = ["$", startSymbol ]
    i, j = 0, 1
    matched = []
    error=[]
    errorFlag=False
    table = PrettyTable(["Stack top", "Current input symbol", "Action"])
    while(inp[i][1] != "$" and stack[j]!="$"):
        # print(i, j)
        try:
            if(stack[j] == inp[i][1]):
                table.add_row([stack[j], inp[i][1], "Match " + str(inp[i][1])])
                symbolTable.updateMatch(inp[i])
                matched.append(inp[i])
                errorFlag=False
                stack.pop()
                i += 1
                j -= 1
                # print(stack, inp, i)
            elif(stack[j] in nonTerminals):
                production =
parsingTable[nonTerminals.index(stack[j])][terminals.index(inp[i][1])]
                if len(production)==0:
                    table.add_row([stack[j], inp[i][1], "ERROR! skip " + inp[i]])
                    if errorFlag==False:
                        error.append("Error near line no. "+str(inp[i][2]))
                        errorFlag=True
                    i+=1
                elif "sync" == production[0]:
                    table.add_row([stack[j], inp[i][1], "ERROR! pop " + stack[j]])
                    if errorFlag==False:
                        error.append("Error near line no. "+str(inp[i][2]))
                        errorFlag=True
                    stack.pop()
                    j-=1
                elif inp[i][1] != stack[j] and len(production)==0:
                    if len(stack)==2:
                        table.add_row([stack[j], inp[i][1], "ERROR! skip " + inp[i]])
                        if errorFlag==False:
                            error.append("Error near line no. "+str(inp[i][2]))
                            errorFlag=True
                        i+=1
                    else:
                        table.add_row([stack[j], inp[i][1], "ERROR! pop " + stack[j]])
                        if errorFlag==False:
                            error.append("Error near line no. "+str(inp[i][2]))
                            errorFlag=True
                        stack.pop()
                        j-=1
                else:
                    f=(nonTerminals[nonTerminals.index(stack[j])]+ "->" + "
".join(production))

symbolTable.updateOutput(nonTerminals[nonTerminals.index(stack[j])],production)
                    table.add_row([stack[j], inp[i][1],"Output " + str(f)])
                    errorFlag=False
                    stack.pop()
                    j -= 1
                    if("#" not in production):
                        for ele in production[::-1]:
                            stack.append(ele)
                            j += 1
                        # print(stack)
```

```
            else:
                # manage error
                table.add_row([stack[j], inp[i][1], "ERROR! skip " + inp[i]])
                if errorFlag==False:
                    error.append("Error near line no. "+str(inp[i][2]))
                    errorFlag=True
                i+=1
        except:
            break
    # print(matched, inp[:-1], stack)
    print(table)
    if(matched != inp[:-1] or stack != ["$"] or len(error)!=0):
        print("Result:")
        print("Invalid input!")
        print("Errors:")
        for line in error:
            print(line)
    else:
        print("Result:")
        print("Valid input!")
    return [symbolTable]
```

Figure 16: Parse( ) method

In the program shown in Figure 16, the panic mode recovery is used to manage the error and recover back the parser. The entries with sync are used to make a decision on what symbols to skip and if the top stack do not match the input string we pop the items.

2.8 Symbol Table

The symbol table is a data structure used to store the information about all possible variables, functions, classes, interfaces, objects and so on. In the implementation we only considered for storing the information about all variables along with the size and datatype associated with them.

The below implemented symbol table makes use of python dictionary which provides a Hash Map table as its data structure making, it efficient to access any elements or update on them. Each entry is again a dictionary of information specifying the fields like “DESCRIPTION”, “DATATYPE” and “SIZE”.

```
class SymbolTable:

    def __init__(self):
        self.symbTable={}
        self.symbolDT=""


    def lookup(self,id):
        if id in self.symbTable:
            return self.symbTable[id]
        return None
```

```
    def insertNewItem(self,id):
        if id not in self.symbTable:
            self.symbTable[id]={}
        else:
            print("Warning:",id,'already declared')

    def setAttribute(self,id,attr,val):
        if id in self.symbTable:
            self.symbTable[id][attr]=val

    def getAttribute(self,id,attr):
        if id in self.symbTable:
            return self.symbTable[id][attr]
        else:
            print('Warning',attr,'not found')


    def updateTable(self,tokens):
        for token in tokens:
            lexeme = token[0]
            desc = token[1]
            if desc == "IDENTIFIER":
                self.insertNewItem(lexeme)
                self.setAttribute(lexeme,"DESC",desc)


    def updateOutput(self,head,body):

        if head=="DATATYPE":
            self.symbolDT=body[0]

    def updateMatch(self,token):
        if token[1] == "IDENTIFIER":
            self.setAttribute(token[0],"TYPE",self.symbolDT)
            size=mapSize(self.symbolDT)
            self.setAttribute(token[0],"SIZE",size)

def mapSize(dt):
    if dt=="INTEGER":return 2
    elif dt=="FLOAT":return 4
    elif dt=="CHAR":return 1
    elif dt=="DOUBLE":return 8
```

Figure 17: Symbol Table manager

The entire class for Symbol Table manager is defined in Figure 17. The implementation contains *updateTable( )* which is used after lexical analysis to insert various components into the symbol table. The method *updateOutput( )* takes in a production and tries to decide what action to be taken, in our case we choose to take the datatype when we have *head* input as DATATYPE. Similarly *updateMatch( )* is used to add the attribute details for each entry. The *updateOutput*( ) and *updateMatch*( ) are used by the parser to handle the symbol table.

CHAPTER 3

RESULTS

Output

In this chapter we will show the output for some of the test cases.

The lexical analyser generates a sequence of tokens as its output. This output is given to the symbol table to update the variables and to the Parser for parsing. The output of the parser is a sequence of derivations in left most order to derive the input source code. At any stage if error occurs then the input symbols are skipped until we get a proper handle.

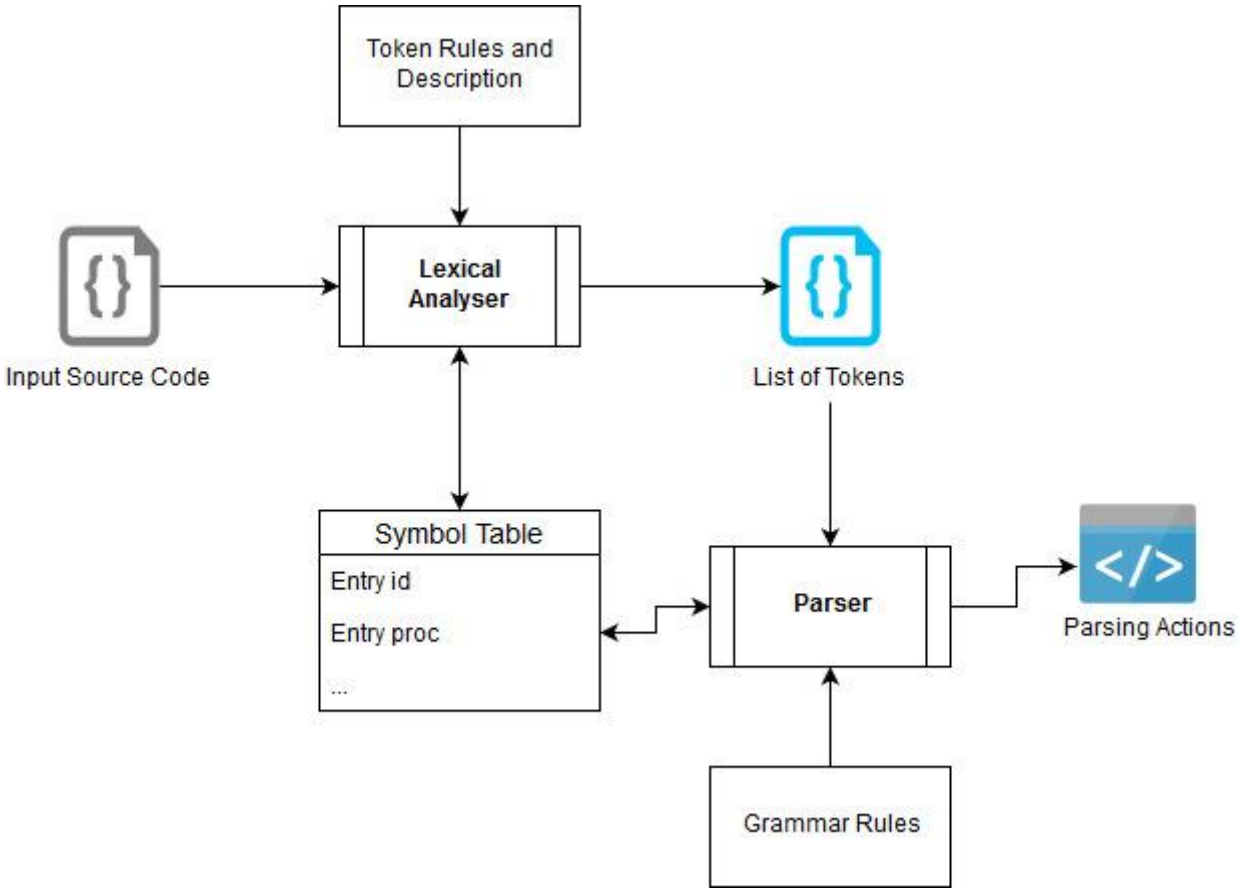

Figure 18: Overall Lexer and Parser Structure

Figure 18 shows the overall structure of the compiler stages involved in this project. The input for our lexical analyser are Source Code and Token rules. At this stage it generates List of Tokens and adds each identifier details into symbol table. The Parser takes these list of tokens as input along with Grammar Rules and produces Parsing Actions in Left Most Derivation order and updates the symbol table with suitable information.

CASE 1: Invalid Input

Consider below a piece of code which is having an extra "*comma*" in line number 3 which makes the input invalid. The sample source code for this is shown in Figure 20.

```
int main()
begin
    int a, ;
end
```

Figure 19: Error Input

```
Output of Lexer
+-------+-------------+
| Token |    Lexeme   |
+-------+-------------+
|  int  |   INTEGER   |
|  main |  PGM_START  |
|   (   |  LEFT_PARA  |
|   )   |  RIGHT_PARA |
| begin | BLOCK_START |
|  int  |   INTEGER   |
|   a   |  IDENTIFIER |
|   ,   |  SEPERATOR  |
|   ;   |     EOS     |
|  end  |  BLOCK_END  |
+-------+-------------+
```

Figure 20: Lexical Analyser Output

```
Output of Parser
+-------------+---------------------+-----------------------------------------------------------------------------------------+
|  Stack top  | Current input symbol |                                          Action                                         |
+-------------+---------------------+-----------------------------------------------------------------------------------------+
|      S      |       INTEGER       | Output S->DATATYPE PGM_START LEFT_PARA RIGHT_PARA BLOCK_START STMTS BLOCK_END |
|   DATATYPE  |       INTEGER       |                                Output DATATYPE->INTEGER                                 |
|   INTEGER   |       INTEGER       |                                     Match INTEGER                                       |
|  PGM_START  |      PGM_START      |                                    Match PGM_START                                      |
|  LEFT_PARA  |      LEFT_PARA      |                                    Match LEFT_PARA                                      |
|  RIGHT_PARA |      RIGHT_PARA     |                                    Match RIGHT_PARA                                     |
| BLOCK_START |     BLOCK_START     |                                   Match BLOCK_START                                     |
|    STMTS    |       INTEGER       |                               Output STMTS->STMT STMTS                                  |
|     STMT    |       INTEGER       |                             Output STMT->DECLARATION EOS                                |
| DECLARATION |       INTEGER       |                          Output DECLARATION->DATATYPE STMT1                             |
|   DATATYPE  |       INTEGER       |                                Output DATATYPE->INTEGER                                 |
|   INTEGER   |       INTEGER       |                                     Match INTEGER                                       |
|    STMT1    |      IDENTIFIER     |                                Output STMT1->LIST STMT2                                 |
|     LIST    |      IDENTIFIER     |                             Output LIST->IDENTIFIER STMT3                               |
|  IDENTIFIER |      IDENTIFIER     |                                    Match IDENTIFIER                                     |
|    STMT3    |      SEPERATOR      |                                    Output STMT3->#                                      |
|    STMT2    |      SEPERATOR      |                               Output STMT2->SEPERATOR STMT1                             |
|  SEPERATOR  |      SEPERATOR      |                                    Match SEPERATOR                                      |
|    STMT1    |         EOS         |                                    ERROR! pop STMT1                                     |
|     EOS     |         EOS         |                                       Match EOS                                         |
|    STMTS    |      BLOCK_END      |                                    Output STMTS->#                                      |
|  BLOCK_END  |      BLOCK_END      |                                    Match BLOCK_END                                      |
+-------------+---------------------+-----------------------------------------------------------------------------------------+
Result:
Invalid input!
Errors:
Error near line no. 3

Symbol Table
+-------+-------------+-----------------+
| Token | Description | Size (in bytes) |
+-------+-------------+-----------------+
|   a   |  IDENTIFIER |        2        |
+-------+-------------+-----------------+
```

Figure 19: Parser and Symbol Table Output

The outputs generated by the program for the source code, shown in Figure 19 is depicted in Figure 20 and Figure 21. In Figure 20 the output is generated after passing our source code to the Lexical Analyser. Here we get a list of tokens along with its Description. This information along with line number of each token is sent to the Parser for further actions. Parser generated output is shown in Figure 21. Since there was a mistake in the source code, the parser shown that error message saying that there is error near line number 3 and further continued parsing. Symbol Table is drawn which shows the available variables in our program.

CASE 2: Valid Input

The given code in Figure 22 is used to show a valid case output. Here we have declaration, conditional statements and printf or display statement.

```
int main()
begin
        int a, b = 5 + 7;
        if ( a + b  <= 3 )
        begin
                printf("hello");
        end
end
```

Figure 20: Valid input

```
Output of Lexer
+---------+-------------+
|  Token  |    Lexeme   |
+---------+-------------+
|   int   |   INTEGER   |
|   main  |  PGM_START  |
|    (    |  LEFT_PARA  |
|    )    |  RIGHT_PARA |
|  begin  | BLOCK_START |
|   int   |   INTEGER   |
|    a    |  IDENTIFIER |
|    ,    |  SEPERATOR  |
|    b    |  IDENTIFIER |
|    =    |    ASSIGN   |
|    5    |    DIGITS   |
|    +    |     ADD     |
|    7    |    DIGITS   |
|    ;    |     EOS     |
|    if   |      IF     |
|    (    |  LEFT_PARA  |
|    a    |  IDENTIFIER |
|    +    |     ADD     |
|    b    |  IDENTIFIER |
|    <=   |      LE     |
|    3    |    DIGITS   |
|    )    |  RIGHT_PARA |
|  begin  | BLOCK_START |
```

```
|  printf |   DISPLAY   |
|    (    |  LEFT_PARA  |
| "hello" |    STRING   |
|    )    |  RIGHT_PARA |
|    ;    |     EOS     |
|   end   |  BLOCK_END  |
|   end   |  BLOCK_END  |
+---------+-------------+
```

Figure 21: Lexical Analyser Output

Output of Parser

Stack top	Current input symbol	Action
S	INTEGER	Output S->DATATYPE PGM_START LEFT_PARA RIGHT_PARA BLOCK_START STMTS BLOCK_END
DATATYPE	INTEGER	Output DATATYPE->INTEGER
INTEGER	INTEGER	Match INTEGER
PGM_START	PGM_START	Match PGM_START
LEFT_PARA	LEFT_PARA	Match LEFT_PARA
RIGHT_PARA	RIGHT_PARA	Match RIGHT_PARA
BLOCK_START	BLOCK_START	Match BLOCK_START
STMTS	INTEGER	Output STMTS->STMT STMTS
STMT	INTEGER	Output STMT->DECLARATION EOS
DECLARATION	INTEGER	Output DECLARATION->DATATYPE STMT1
DATATYPE	INTEGER	Output DATATYPE->INTEGER
INTEGER	INTEGER	Match INTEGER
STMT1	IDENTIFIER	Output STMT1->LIST STMT2
LIST	IDENTIFIER	Output LIST->IDENTIFIER STMT3
IDENTIFIER	IDENTIFIER	Match IDENTIFIER
STMT3	SEPERATOR	Output STMT3->#
STMT2	SEPERATOR	Output STMT2->SEPERATOR STMT1
SEPERATOR	SEPERATOR	Match SEPERATOR
STMT1	IDENTIFIER	Output STMT1->LIST STMT2
LIST	IDENTIFIER	Output LIST->IDENTIFIER STMT3
IDENTIFIER	IDENTIFIER	Match IDENTIFIER
STMT3	ASSIGN	Output STMT3->ASSIGN E
ASSIGN	ASSIGN	Match ASSIGN
E	DIGITS	Output E->E2 E1
E2	DIGITS	Output E2->T E3
T	DIGITS	Output T->F T1
F	DIGITS	Output F->DIGITS
DIGITS	DIGITS	Match DIGITS
T1	ADD	Output T1->#
E3	ADD	Output E3->OP1 T E3
OP1	ADD	Output OP1->ADD
ADD	ADD	Match ADD
T	DIGITS	Output T->F T1
F	DIGITS	Output F->DIGITS
DIGITS	DIGITS	Match DIGITS
T1	EOS	Output T1->#
E3	EOS	Output E3->#
E1	EOS	Output E1->#
STMT2	EOS	Output STMT2->#
EOS	EOS	Match EOS
STMTS	IF	Output STMTS->STMT STMTS
STMT	IF	Output STMT->CONDITION
CONDITION	IF	Output CONDITION->IF LEFT_PARA E RIGHT_PARA BLOCK_START STMTS BLOCK_END
IF	IF	Match IF
LEFT_PARA	LEFT_PARA	Match LEFT_PARA
E	IDENTIFIER	Output E->E2 E1
E2	IDENTIFIER	Output E2->T E3
T	IDENTIFIER	Output T->F T1
F	IDENTIFIER	Output F->IDENTIFIER

```
|  IDENTIFIER |      IDENTIFIER      |                                  Match IDENTIFIER                                  |
|      T1     |         ADD          |                                    Output T1->#                                    |
|      E3     |         ADD          |                                 Output E3->OP1 T E3                                |
|     OP1     |         ADD          |                                   Output OP1->ADD                                  |
|     ADD     |         ADD          |                                      Match ADD                                     |
|      T      |      IDENTIFIER      |                                   Output T->F T1                                   |
|      F      |      IDENTIFIER      |                                 Output F->IDENTIFIER                               |
|  IDENTIFIER |      IDENTIFIER      |                                  Match IDENTIFIER                                  |
|      T1     |          LE          |                                    Output T1->#                                    |
|      E3     |          LE          |                                    Output E3->#                                    |
|      E1     |          LE          |                                 Output E1->relop E2 E1                             |
|    relop    |          LE          |                                   Output relop->LE                                 |
|      LE     |          LE          |                                      Match LE                                      |
|      E2     |        DIGITS        |                                   Output E2->T E3                                  |
|      T      |        DIGITS        |                                   Output T->F T1                                   |
|      F      |        DIGITS        |                                  Output F->DIGITS                                  |
|    DIGITS   |        DIGITS        |                                    Match DIGITS                                    |
|      T1     |      RIGHT_PARA      |                                    Output T1->#                                    |
|      E3     |      RIGHT_PARA      |                                    Output E3->#                                    |
|      E1     |      RIGHT_PARA      |                                    Output E1->#                                    |
|  RIGHT_PARA |      RIGHT_PARA      |                                  Match RIGHT_PARA                                  |
| BLOCK_START |     BLOCK_START      |                                 Match BLOCK_START                                  |
|    STMTS    |       DISPLAY        |                              Output STMTS->STMT STMTS                              |
|     STMT    |       DISPLAY        |                             Output STMT->FUNCTION EOS                              |
|   FUNCTION  |       DISPLAY        |                Output FUNCTION->DISPLAY LEFT_PARA MSG RIGHT_PARA                   |
|   DISPLAY   |       DISPLAY        |                                   Match DISPLAY                                    |
|  LEFT_PARA  |      LEFT_PARA       |                                  Match LEFT_PARA                                   |
|     MSG     |        STRING        |                                 Output MSG->STRING                                 |
|    STRING   |        STRING        |                                    Match STRING                                    |
|  RIGHT_PARA |      RIGHT_PARA      |                                  Match RIGHT_PARA                                  |
|      EOS    |         EOS          |                                      Match EOS                                     |
|    STMTS    |      BLOCK_END       |                                    Output STMTS->#                                 |
|  BLOCK_END  |      BLOCK_END       |                                   Match BLOCK_END                                  |
|    STMTS    |      BLOCK_END       |                                    Output STMTS->#                                 |
|  BLOCK_END  |      BLOCK_END       |                                   Match BLOCK_END                                  |
+-------------+----------------------+------------------------------------------------------------------------------------+
Result:
Valid input!

Symbol Table
+-------+-------------+-----------------+
| Token | Description | Size (in bytes) |
+-------+-------------+-----------------+
|   a   |  IDENTIFIER |        2        |
|   b   |  IDENTIFIER |        2        |
+-------+-------------+-----------------+
```

Figure 22: Parser and Symbol Table Output

The outputs for the code snippet shown in Figure 22 is shown in Figure 23 and Figure 24. Figure 23 shows the output of the lexical analyser phase which generated all the tokens from the source code. This list of tokens is given to the parser for parsing which generated a Valid Output as shown in Figure 24. The symbol table is drawn which shows different variables needed for our program.

CHAPTER 4

CONCLUSION

The project mainly aimed at implementing Lexical and Syntax Analyser which are the first two stages in Compiler Analysis Phase. The modules that are implemented for the above two are provided by two classes and this can be easily reused for parsing different grammars. A Regular Expression system is used to define the token rules and identify every tokens from the program. The parser used in this project is LL(1) parser which produces the output in Left Most Derivation order. If there are any syntax errors in the source code or the input string, the parser will identify and notify them to the users. The error recovery scheme is implemented using Panic mode recovery. The project is implemented entirely in Python. RE module is used to implement the Regular expression system and to define the Lexical Analyser.

The challenging part of the project was to construct the parsing table based on given set of rules. In order to automate this, LL(1) parser makes use of FIRST and FOLLOW sets which in here is implemented by making use of recursive calls. Further the panic mode recovery had to be integrated with the parser part in order to recover back from error.

The project can be further extended by adding in the remaining stages of the compiler namely Semantic Analysis, Intermediate Code Generation, Code Optimization and Final Code Generation. A complete compiler for a custom language or any languages can be implemented by including various constructs.