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

NITTE EDUCATION TRUST

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.

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 Compiler

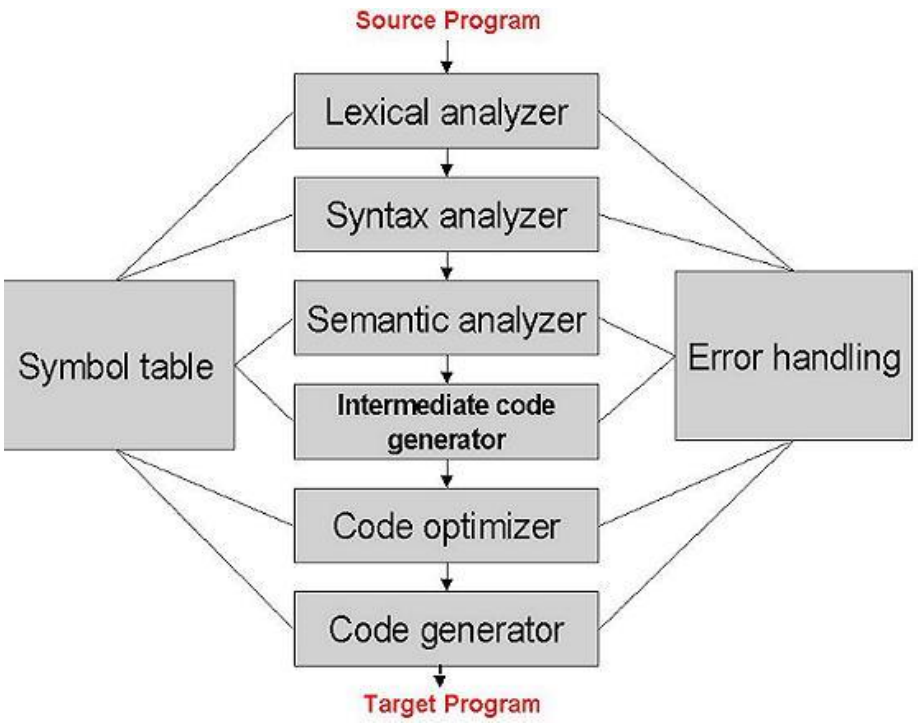
We know that computer is a logical assembly of both hardware and software. The hardware consists of all the physical components interconnected to function as needed and the software is used to control and manage the software. But when we look into the actual implementation we have the basic blocks which 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



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.

- 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.

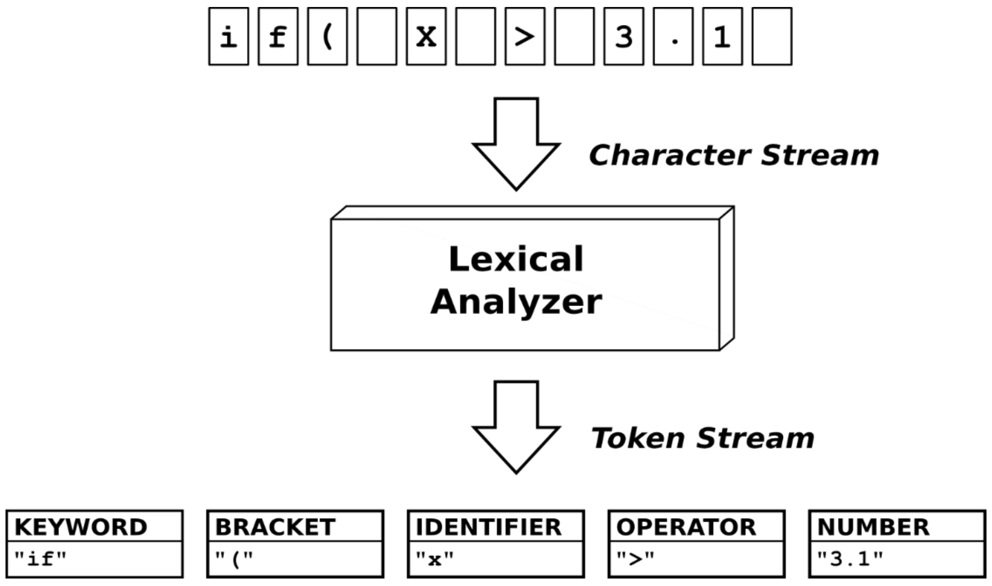
Some compilers have error handling mechanism so that the compilation process doesn’t halt in between. For this purpose, we have an Error Handler whose main task is to correlate the appropriate errors and continue with compiling the further codes. There are many error handling strategies involved in different 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.



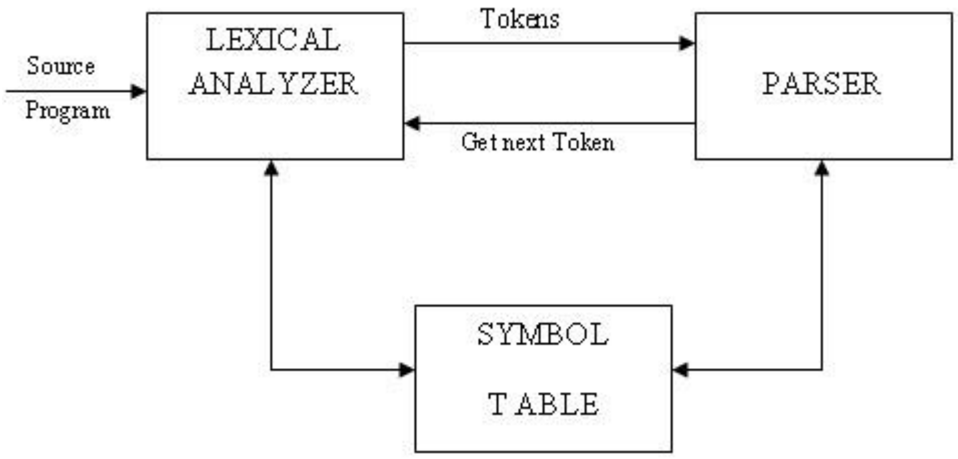
The syntax analysis phase takes in each of tokens one at a time. If any characters do not match the token rules, then we produce in an error message and continue with processing by using some error recovery scheme which is done by Error Handler. The process of lexical analysis is also termed as Scanning 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.



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.

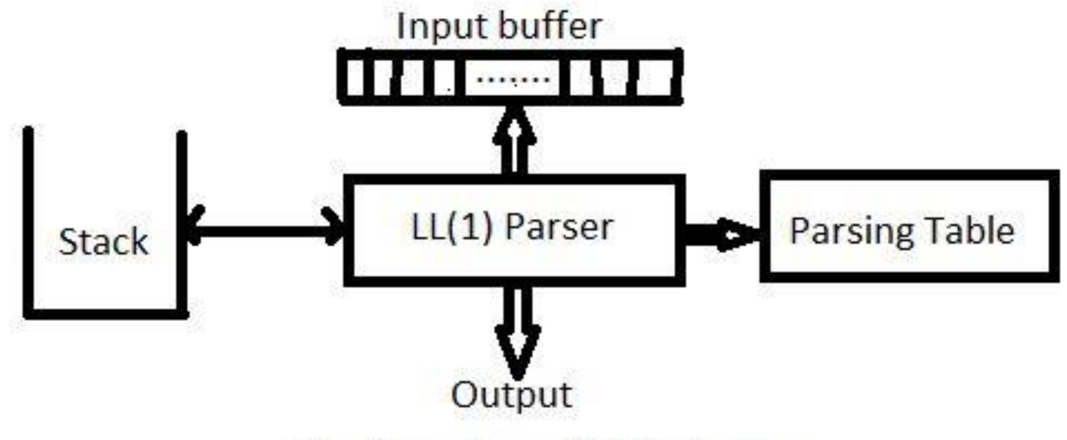
CHAPTER 2

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.



Configuration of 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.

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		

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

TOKENS = ["=","==","%","/","\*","-
","\+","!=","!","\|\|","&&","<=","<",">=",">","if","main","begin","end","printf
","float","char","int",r'\(',r'\)',",",";",r'\".*\"',r'\'.?\'',r'[a-zA-Z][a-zA-
Z0-9]*',r'[0-9]+',"'",'"']

DATATYPES = ["INTEGER","FLOAT","CHAR"]

TOKEN_DESC={ "=":"ASSIGN" ,
             "==":"EQ",
             "%":"MOD",
             "/":"DIV",
             "*":"MUL",
             "-":"SUB",
             "+":"ADD",
             "!=":"NE",
             "!":"NOT",
             "||":"OR",
             "&&":"AND",
             "<=":"LE",
             "<":"LT",
             ">=":"GE",
             ">":"GT",
             "if": "IF",
             "main": "PGM_START",
             "begin":"BLOCK_START",
             "end":"BLOCK_END",
             "printf":"DISPLAY",
             "int":"INTEGER",
             "float":"FLOAT",
             "char":"CHAR",
             "(":"LEFT_PARA",
             ")":"RIGHT_PARA",
             ",":"SEPERATOR",
             ";":"EOS"
         }

class Tokenizer:
    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
```

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
DECLARATION -> DATATYPE STMT1
STMT1 -> IDENTIFIER STMT2
STMT2 -> SEPERATOR STMT1 | ASSIGN E STMT3 | #
STMT3 -> SEPERATOR STMT1 | #
CONDITION -> IF LEFT_PARA E RIGHT_PARA BLOCK_START STMTS BLOCK_END
FUNCTION -> DISPLAY LEFT_PARA MSG RIGHT_PARA
MSG -> STRING | IDENTIFIER | CHARACTER
E -> T E1
E1 -> relop T E1 | #
T -> F T1
T1 -> ADD F T1 | SUB F T1 | #
F -> H F1
F1 -> MUL H F1 | DIV H F1 | #
H -> LEFT_PARA E RIGHT_PARA | SUB H | IDENTIFIER | DIGITS
relop -> EQ | NE | AND | OR | LE | LT | GE | GT

In the previous grammar we need to identify the grammar symbols and variables associated in out program. This is done by taking all the head of the productions as non-terminals or variables and remaining production symbols are marked as terminals.

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

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

S is the starting production for our grammar.

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.

For finding FIRST:

```
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))
```

For finding the FOLLOWS:

```
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:
                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]
```

After applying the above methods on to the given grammar we obtained the following FIRST and FOLLOW Set for different terminals.

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

Follow Set:
S: ['$']
STMTS: ['$', 'BLOCK_END']
DATATYPE: ['IDENTIFIER', 'PGM_START']
STMT: ['$', 'IF', 'BLOCK_END', 'CHAR', 'DISPLAY', 'INTEGER', 'FLOAT']
DECLARATION: ['EOS']
STMT1: ['EOS', '$']
STMT2: ['EOS', '$']
LIST: ['SEPERATOR', '$', 'EOS']
STMT3: ['SEPERATOR', '$', 'EOS']
CONDITION: ['DISPLAY']
FUNCTION: ['EOS']
MSG: ['RIGHT_PARA']
E: ['RIGHT_PARA', 'SEPERATOR', '$', 'EOS']
E1: ['SEPERATOR', 'RIGHT_PARA', '$', 'EOS']

E2: ['EOS', '$', 'NE', 'EQ', 'RIGHT_PARA', 'GT', 'SEPERATOR', 'LT', 'LE', 'GE', 'OR', 'AND']
E3: ['EOS', '$', 'NE', 'EQ', 'RIGHT_PARA', 'GT', 'SEPERATOR', 'LT', 'LE', 'GE', 'OR', 'AND']
T: ['EOS', '$', 'NE', 'SUB', 'EQ', 'RIGHT_PARA', 'GT', 'SEPERATOR', 'ADD', 'LT', 'LE', 'GE', 'OR', 'AND']
T1: ['EOS', '$', 'NE', 'SUB', 'EQ', 'RIGHT_PARA', 'GT', 'SEPERATOR', 'ADD', 'LT', 'LE', 'GE', 'OR', 'AND']
OP1: ['DIGITS', 'LEFT_PARA', 'IDENTIFIER', 'SUB']
OP2: ['LEFT_PARA', 'IDENTIFIER', 'SUB', 'DIGITS']
F: ['EOS', '$', 'NE', 'SUB', 'EQ', 'IDENTIFIER', 'MUL', 'RIGHT_PARA', 'GT', 'DIV', 'SEPERATOR', 'ADD', 'LT', 'LE', 'GE', 'OR', 'AND']
relop: ['LEFT_PARA', 'IDENTIFIER', 'SUB', 'DIGITS']

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.

```python
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 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")


    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

    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)
```

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.

```
def parse(inp,startSymbol,nonTerminals,terminals,parsingTable):
    inp.append(("$","$"))
    stack = ["$", startSymbol ]
    i, j = 0, 1
    matched = []
    error=[]
    errorFlag=False
    rounds=1
    while(inp[i][1] != "$" and stack[j]!="$"):
        print("-"*100)
        print("Round:",rounds)
        rounds+=1
        # print(i, j)
        try:
            if(stack[j] == inp[i][1]):
                print("Stack:  "+",".join(str(x) for x in stack),"Input: " +
",".join(str(x[0]) for x in inp[i:]),"Action: Match "+str(inp[i][1]),sep="\n" )
```

```
                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:
                    print("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]:
                    print("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:
                        print("Error skip:",inp[i])
                        if errorFlag==False:
                            error.append("Error near line no. "+str(inp[i][2]))
                            errorFlag=True
                        i+=1
                    else:
                        print("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))
```

```python
                    print("Stack:  "+",".join(str(x) for x in stack),"Input: " + ",".join(str(x[0]) for x in inp[i:]),"Action: Output "+str(f),sep="\n" )
                    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
                print("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)
    if(matched != inp[:-1] or stack != ["$"] or len(error)!=0):
        print("-"*100)
        print("Result:")
        print("Invalid input!")
        print("Errors:")
        for line in error:
            print(line)

    else:
        print("-"*100)
        print("Result:")
        print("Valid input!")
```

CHAPTER 3

RESULTS

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

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.

Test case 1: Valid input Verification
Consider the following input

```
int main()
begin
   int a=45, b;
   if(a > 3)
   begin
      printf("hello");
   end
end
```

The output will be:

```
Output of Lexer
Token               Lexeme
------------------------------
int                 INTEGER
main                PGM_START
(                   LEFT_PARA
)                   RIGHT_PARA
begin               BLOCK_START
int                 INTEGER
a                   IDENTIFIER
=                   ASSIGN
45                  DIGITS
,                   SEPERATOR
b                   IDENTIFIER
;                   EOS
if                  IF
(                   LEFT_PARA
a                   IDENTIFIER
>               GT
3               DIGITS
)               RIGHT_PARA
begin           BLOCK_START
printf          DISPLAY
(               LEFT_PARA
"hello"         STRING
)               RIGHT_PARA
;               EOS
end             BLOCK_END
end             BLOCK_END
```

Output of Parser

```
----------------------------------------------------------------------------------------------------
Round: 1
Stack:  $,S
Input: int,main,(,),begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output S->DATATYPE PGM_START LEFT_PARA RIGHT_PARA
BLOCK_START STMTS BLOCK_END
----------------------------------------------------------------------------------------------------
Round: 2
Stack:
$,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA,PGM_STAR
T,DATATYPE
Input: int,main,(,),begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output DATATYPE->INTEGER
----------------------------------------------------------------------------------------------------
Round: 3
Stack:
$,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA,PGM_STAR
T,INTEGER
Input: int,main,(,),begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match INTEGER
----------------------------------------------------------------------------------------------------
Round: 4
Stack:
$,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA,PGM_STAR
T
Input: main,(,),begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match PGM_START
----------------------------------------------------------------------------------------------------
```

```
Round: 5
Stack:  $,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA
Input: (,),begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match LEFT_PARA
----------------------------------------------------------------------------------------------------
Round: 6
Stack:  $,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA
Input: ),begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match RIGHT_PARA
----------------------------------------------------------------------------------------------------
Round: 7
Stack:  $,BLOCK_END,STMTS,BLOCK_START
Input: begin,int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match BLOCK_START
----------------------------------------------------------------------------------------------------
Round: 8
Stack:  $,BLOCK_END,STMTS
Input: int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMTS->STMT STMTS
----------------------------------------------------------------------------------------------------
Round: 9
Stack:  $,BLOCK_END,STMTS,STMT
Input: int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT->DECLARATION EOS
----------------------------------------------------------------------------------------------------
Round: 10
Stack:  $,BLOCK_END,STMTS,EOS,DECLARATION
Input: int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output DECLARATION->DATATYPE STMT1
----------------------------------------------------------------------------------------------------
Round: 11
Stack:  $,BLOCK_END,STMTS,EOS,STMT1,DATATYPE
Input: int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output DATATYPE->INTEGER
----------------------------------------------------------------------------------------------------
Round: 12
Stack:  $,BLOCK_END,STMTS,EOS,STMT1,INTEGER
Input: int,a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match INTEGER
----------------------------------------------------------------------------------------------------
Round: 13
Stack:  $,BLOCK_END,STMTS,EOS,STMT1
```

```
Input: a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT1->LIST STMT2
-----------------------------------------------------------------------------------------------
Round: 14
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,LIST
Input: a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output LIST->IDENTIFIER STMT3
-----------------------------------------------------------------------------------------------
Round: 15
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,STMT3,IDENTIFIER
Input: a,=,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match IDENTIFIER
-----------------------------------------------------------------------------------------------
Round: 16
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,STMT3
Input: =,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT3->ASSIGN E
-----------------------------------------------------------------------------------------------
Round: 17
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E,ASSIGN
Input: =,45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match ASSIGN
-----------------------------------------------------------------------------------------------
Round: 18
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E
Input: 45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E->E2 E1
-----------------------------------------------------------------------------------------------
Round: 19
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E2
Input: 45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E2->T E3
-----------------------------------------------------------------------------------------------
Round: 20
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T
Input: 45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output T->F T1
-----------------------------------------------------------------------------------------------
Round: 21
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T1,F
Input: 45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output F->DIGITS
```

```
-----------------------------------------------------------------------------------------------
Round: 22
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T1,DIGITS
Input: 45,,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match DIGITS
-----------------------------------------------------------------------------------------------
Round: 23
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T1
Input: ,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output T1->#
-----------------------------------------------------------------------------------------------
Round: 24
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3
Input: ,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E3->#
-----------------------------------------------------------------------------------------------
Round: 25
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1
Input: ,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E1->#
-----------------------------------------------------------------------------------------------
Round: 26
Stack:  $,BLOCK_END,STMTS,EOS,STMT2
Input: ,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT2->SEPERATOR STMT1
-----------------------------------------------------------------------------------------------
Round: 27
Stack:  $,BLOCK_END,STMTS,EOS,STMT1,SEPERATOR
Input: ,,b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match SEPERATOR
-----------------------------------------------------------------------------------------------
Round: 28
Stack:  $,BLOCK_END,STMTS,EOS,STMT1
Input: b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT1->LIST STMT2
-----------------------------------------------------------------------------------------------
Round: 29
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,LIST
Input: b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output LIST->IDENTIFIER STMT3
-----------------------------------------------------------------------------------------------
Round: 30
```

```
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,STMT3,IDENTIFIER
Input: b,;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match IDENTIFIER
-----------------------------------------------------------------------------------------------
Round: 31
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,STMT3
Input: ;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT3->#
-----------------------------------------------------------------------------------------------
Round: 32
Stack:  $,BLOCK_END,STMTS,EOS,STMT2
Input: ;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT2->#
-----------------------------------------------------------------------------------------------
Round: 33
Stack:  $,BLOCK_END,STMTS,EOS
Input: ;,if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match EOS
-----------------------------------------------------------------------------------------------
Round: 34
Stack:  $,BLOCK_END,STMTS
Input: if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMTS->STMT STMTS
-----------------------------------------------------------------------------------------------
Round: 35
Stack:  $,BLOCK_END,STMTS,STMT
Input: if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output STMT->CONDITION
-----------------------------------------------------------------------------------------------
Round: 36
Stack:  $,BLOCK_END,STMTS,CONDITION
Input: if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output CONDITION->IF LEFT_PARA E RIGHT_PARA BLOCK_START
STMTS BLOCK_END
-----------------------------------------------------------------------------------------------
Round: 37
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E,L
EFT_PARA,IF
Input: if,(,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match IF
-----------------------------------------------------------------------------------------------
```

```
Round: 38
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E,L
EFT_PARA
Input: (,a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match LEFT_PARA
-----------------------------------------------------------------------------------------------
Round: 39
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E
Input: a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E->E2 E1
-----------------------------------------------------------------------------------------------
Round: 40
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E2
Input: a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E2->T E3
-----------------------------------------------------------------------------------------------
Round: 41
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T
Input: a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output T->F T1
-----------------------------------------------------------------------------------------------
Round: 42
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T1,F
Input: a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output F->IDENTIFIER
-----------------------------------------------------------------------------------------------
Round: 43
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T1,IDENTIFIER
Input: a,>,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match IDENTIFIER
-----------------------------------------------------------------------------------------------
Round: 44
```

```
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T1
Input: >,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output T1->#
-----------------------------------------------------------------------------------------------
Round: 45
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3
Input: >,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E3->#
-----------------------------------------------------------------------------------------------
Round: 46
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1
Input: >,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E1->relop E2 E1
-----------------------------------------------------------------------------------------------
Round: 47
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E2,relop
Input: >,3,),begin,printf,(,"hello",),;,end,end,$
Action: Output relop->GT
-----------------------------------------------------------------------------------------------
Round: 48
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E2,GT
Input: >,3,),begin,printf,(,"hello",),;,end,end,$
Action: Match GT
-----------------------------------------------------------------------------------------------
Round: 49
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E2
Input: 3,),begin,printf,(,"hello",),;,end,end,$
Action: Output E2->T E3
-----------------------------------------------------------------------------------------------
Round: 50
```

```
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T
Input: 3,),begin,printf,(,"hello",),;,end,end,$
Action: Output T->F T1
----------------------------------------------------------------------------------------------------
Round: 51
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T1,F
Input: 3,),begin,printf,(,"hello",),;,end,end,$
Action: Output F->DIGITS
----------------------------------------------------------------------------------------------------
Round: 52
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T1,DIGITS
Input: 3,),begin,printf,(,"hello",),;,end,end,$
Action: Match DIGITS
----------------------------------------------------------------------------------------------------
Round: 53
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3,T1
Input: ),begin,printf,(,"hello",),;,end,end,$
Action: Output T1->#
----------------------------------------------------------------------------------------------------
Round: 54
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1,
E3
Input: ),begin,printf,(,"hello",),;,end,end,$
Action: Output E3->#
----------------------------------------------------------------------------------------------------
Round: 55
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,E1
Input: ),begin,printf,(,"hello",),;,end,end,$
Action: Output E1->#
----------------------------------------------------------------------------------------------------
Round: 56
```

```
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA
Input: ),begin,printf,(,"hello",),;,end,end,$
Action: Match RIGHT_PARA
-----------------------------------------------------------------------------------------------
Round: 57
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,BLOCK_START
Input: begin,printf,(,"hello",),;,end,end,$
Action: Match BLOCK_START
-----------------------------------------------------------------------------------------------
Round: 58
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS
Input: printf,(,"hello",),;,end,end,$
Action: Output STMTS->STMT STMTS
-----------------------------------------------------------------------------------------------
Round: 59
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,STMT
Input: printf,(,"hello",),;,end,end,$
Action: Output STMT->FUNCTION EOS
-----------------------------------------------------------------------------------------------
Round: 60
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS,FUNCTION
Input: printf,(,"hello",),;,end,end,$
Action: Output FUNCTION->DISPLAY LEFT_PARA MSG RIGHT_PARA
-----------------------------------------------------------------------------------------------
Round: 61
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS,RIGHT_PARA,MSG,LEFT_PA
RA,DISPLAY
Input: printf,(,"hello",),;,end,end,$
Action: Match DISPLAY
-----------------------------------------------------------------------------------------------
Round: 62
Stack:
$,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS,RIGHT_PARA,MSG,LEFT_PA
RA
Input: (,"hello",),;,end,end,$
Action: Match LEFT_PARA
-----------------------------------------------------------------------------------------------
Round: 63
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS,RIGHT_PARA,MSG
Input: "hello",),;,end,end,$
```

```
Action: Output MSG->STRING
----------------------------------------------------------------------------------------------------
Round: 64
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS,RIGHT_PARA,STRING
Input: "hello",),;,end,end,$
Action: Match STRING
----------------------------------------------------------------------------------------------------
Round: 65
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS,RIGHT_PARA
Input: ),;,end,end,$
Action: Match RIGHT_PARA
----------------------------------------------------------------------------------------------------
Round: 66
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS,EOS
Input: ;,end,end,$
Action: Match EOS
----------------------------------------------------------------------------------------------------
Round: 67
Stack:  $,BLOCK_END,STMTS,BLOCK_END,STMTS
Input: end,end,$
Action: Output STMTS-># 
----------------------------------------------------------------------------------------------------
Round: 68
Stack:  $,BLOCK_END,STMTS,BLOCK_END
Input: end,end,$
Action: Match BLOCK_END
----------------------------------------------------------------------------------------------------
Round: 69
Stack:  $,BLOCK_END,STMTS
Input: end,$
Action: Output STMTS->#
----------------------------------------------------------------------------------------------------
Round: 70
Stack:  $,BLOCK_END
Input: end,$
Action: Match BLOCK_END
----------------------------------------------------------------------------------------------------
Result:
Valid input!
```

<u>Test case 2</u>: Verification of Error handling in Invalid input

Now consider the following input:

```
int main()
begin
   int a=45, ;
end
```

The output will be:

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


Output of Parser
------------------------------------------------------------------------------------------------
Round: 1
Stack:  $,S
Input: int,main,(,),begin,int,a,=,45,,,;,end,$
Action: Output S->DATATYPE PGM_START LEFT_PARA RIGHT_PARA
BLOCK_START STMTS BLOCK_END
------------------------------------------------------------------------------------------------
Round: 2
Stack:
$,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA,PGM_STAR
T,DATATYPE
Input: int,main,(,),begin,int,a,=,45,,,;,end,$
Action: Output DATATYPE->INTEGER
------------------------------------------------------------------------------------------------
Round: 3
```

```
Stack:
$,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA,PGM_STAR
T,INTEGER
Input: int,main,(,),begin,int,a,=,45,,,;,end,$
Action: Match INTEGER
---------------------------------------------------------------------------------------------------
Round: 4
Stack:
$,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA,PGM_STAR
T
Input: main,(,),begin,int,a,=,45,,,;,end,$
Action: Match PGM_START
---------------------------------------------------------------------------------------------------
Round: 5
Stack:  $,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA,LEFT_PARA
Input: (,),begin,int,a,=,45,,,;,end,$
Action: Match LEFT_PARA
---------------------------------------------------------------------------------------------------
Round: 6
Stack:  $,BLOCK_END,STMTS,BLOCK_START,RIGHT_PARA
Input: ),begin,int,a,=,45,,,;,end,$
Action: Match RIGHT_PARA
---------------------------------------------------------------------------------------------------
Round: 7
Stack:  $,BLOCK_END,STMTS,BLOCK_START
Input: begin,int,a,=,45,,,;,end,$
Action: Match BLOCK_START
---------------------------------------------------------------------------------------------------
Round: 8
Stack:  $,BLOCK_END,STMTS
Input: int,a,=,45,,,;,end,$
Action: Output STMTS->STMT STMTS
---------------------------------------------------------------------------------------------------
Round: 9
Stack:  $,BLOCK_END,STMTS,STMT
Input: int,a,=,45,,,;,end,$
Action: Output STMT->DECLARATION EOS
---------------------------------------------------------------------------------------------------
Round: 10
Stack:  $,BLOCK_END,STMTS,EOS,DECLARATION
Input: int,a,=,45,,,;,end,$
Action: Output DECLARATION->DATATYPE STMT1
```

```
----------------------------------------------------------------------------------------------------
Round: 11
Stack:  $,BLOCK_END,STMTS,EOS,STMT1,DATATYPE
Input: int,a,=,45,,,;,end,$
Action: Output DATATYPE->INTEGER
----------------------------------------------------------------------------------------------------
Round: 12
Stack:  $,BLOCK_END,STMTS,EOS,STMT1,INTEGER
Input: int,a,=,45,,,;,end,$
Action: Match INTEGER
----------------------------------------------------------------------------------------------------
Round: 13
Stack:  $,BLOCK_END,STMTS,EOS,STMT1
Input: a,=,45,,,;,end,$
Action: Output STMT1->LIST STMT2
----------------------------------------------------------------------------------------------------
Round: 14
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,LIST
Input: a,=,45,,,;,end,$
Action: Output LIST->IDENTIFIER STMT3
----------------------------------------------------------------------------------------------------
Round: 15
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,STMT3,IDENTIFIER
Input: a,=,45,,,;,end,$
Action: Match IDENTIFIER
----------------------------------------------------------------------------------------------------
Round: 16
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,STMT3
Input: =,45,,,;,end,$
Action: Output STMT3->ASSIGN E
----------------------------------------------------------------------------------------------------
Round: 17
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E,ASSIGN
Input: =,45,,,;,end,$
Action: Match ASSIGN
----------------------------------------------------------------------------------------------------
Round: 18
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E
Input: 45,,,;,end,$
Action: Output E->E2 E1
----------------------------------------------------------------------------------------------------
Round: 19
```

```
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E2
Input: 45,,,;,end,$
Action: Output E2->T E3
-----------------------------------------------------------------------------------------------
Round: 20
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T
Input: 45,,,;,end,$
Action: Output T->F T1
-----------------------------------------------------------------------------------------------
Round: 21
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T1,F
Input: 45,,,;,end,$
Action: Output F->DIGITS
-----------------------------------------------------------------------------------------------
Round: 22
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T1,DIGITS
Input: 45,,,;,end,$
Action: Match DIGITS
-----------------------------------------------------------------------------------------------
Round: 23
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3,T1
Input: ,,;,end,$
Action: Output T1-># 
-----------------------------------------------------------------------------------------------
Round: 24
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1,E3
Input: ,,;,end,$
Action: Output E3->#
-----------------------------------------------------------------------------------------------
Round: 25
Stack:  $,BLOCK_END,STMTS,EOS,STMT2,E1
Input: ,,;,end,$
Action: Output E1->#
-----------------------------------------------------------------------------------------------
Round: 26
Stack:  $,BLOCK_END,STMTS,EOS,STMT2
Input: ,,;,end,$
Action: Output STMT2->SEPERATOR STMT1
-----------------------------------------------------------------------------------------------
Round: 27
Stack:  $,BLOCK_END,STMTS,EOS,STMT1,SEPERATOR
Input: ,,;,end,$
```

```
Action: Match SEPERATOR
--------------------------------------------------------------------------------------------------
Round: 28
Error pop: STMT1
--------------------------------------------------------------------------------------------------
Round: 29
Stack:  $,BLOCK_END,STMTS,EOS
Input: ;,end,$
Action: Match EOS
--------------------------------------------------------------------------------------------------
Round: 30
Stack:  $,BLOCK_END,STMTS
Input: end,$
Action: Output STMTS->#
--------------------------------------------------------------------------------------------------
Round: 31
Stack:  $,BLOCK_END
Input: end,$
Action: Match BLOCK_END
--------------------------------------------------------------------------------------------------
Result:
Invalid input!
Errors:
Error near line no. 3
```

CHAPTER 4

CONCLUSION

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