**LL(1) Parser**

A Mini Project Report Submitted by

Rahul D Shetty 4NM16CS111

Saurabh D Rao 4NM16CS132

UNDER THE GUIDANCE OF

Mrs. ANISHA P RODRIGUE

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**



(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 RODRIGUE**, 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**

Introduction

Compiler6

Phases in Compiler7

Lexical Analyser9

Syntax Analyser10

Implementation

LL(1) Parser11

Token Rules12

Lexical Analyser13

Production Rules14

First & Follow’s10

Parse Table10

Parser10

Results

Conclusion

**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.

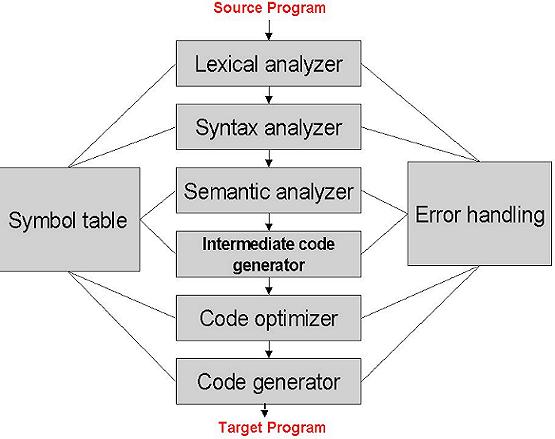
For any novice to expert programmer it is difficult to remember all the code equivalents and programming them in a computer would get quite complex. In order to overcome this problem, we have software programs called “Compiler” whose task is to convert a high lever language code that is easy to understand 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.

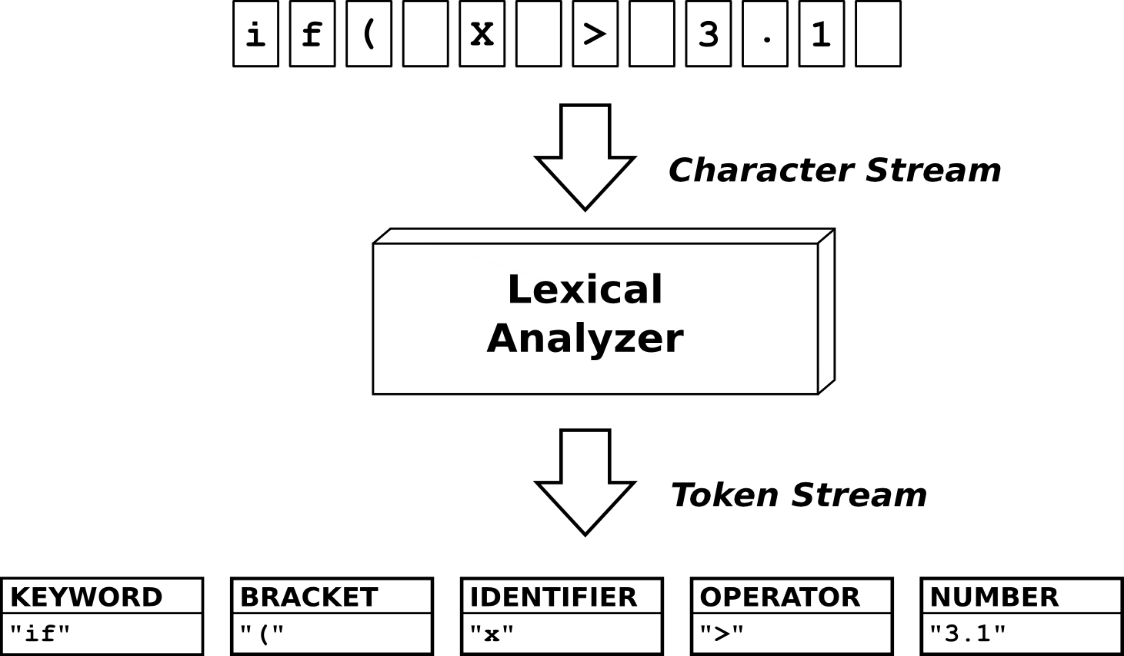
Some compilers have error handling mechanism so that the compilation process doesn’t halt in between. For this purpose, we have an Error Handler whose main task is to correlate the appropriate errors and continue with compiling the further codes. There are many error handling strategies involved in different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



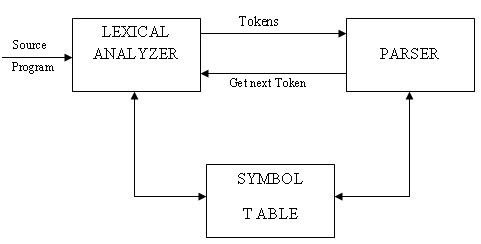
The syntax analysis phase takes in each of tokens one at a time. If any characters do not match the token rules, then we produce in an error message and continue with processing by using some error recovery scheme which is done by Error Handler. The process of lexical analysis is also termed as Scanning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

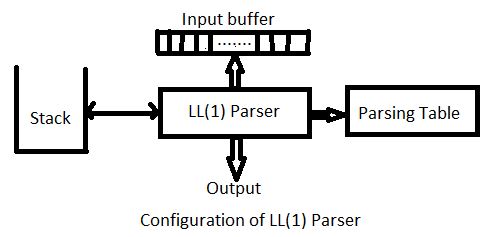
**CHAPTER 2**

## **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 L L ( k ) {\displaystyle LL(k)} parser is a deterministic pushdown automaton with the ability to peek on the next k {\displaystyle k} 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.

****

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,lines,lc\_no=[],[],1

**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**