

REPORT ON

CD Mini Project

Carried out on

**Designing a lexical Parser and LL(1) Parser using
Python based on the hypothetical language**

Submitted to

NMAM INSTITUTE OF TECHNOLOGY, NITTE

(An Autonomous Institution under VTU, Belagavi)

In partial fulfillment of the requirements for the award of the

Degree of Bachelor of Engineering in
Computer Science Engineering

by

Elton Arthur

4NM20CS068

Emmanuel Joshy

4NM20CS069

Harshith Rao

4NM20CS076

Submitted to,

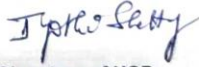
Dr. Pallavi K.N

Associate Professor

CERTIFICATE

Certified that the project work entitled is a bonafide work carried out by Emmanuel Joshy (4NM20CS069), Elton Arthur (4NM20CS068) and Harshith Rao (4NM20CS076) has completed CD mini project on “Designing a lexical Parser and LL(1) Parser using Python based on the hypothetical language” during August 2023 – December 2023 fulfilling the partial requirements for the award of degree of Bachelor of Engineering in Computer Science and Engineering at NMAM Institute of Technology, Nitte


Name & Signature of Mentor


Signature of HOD

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of people who made it possible because “Success is the abstract of hard work and perseverance, but steadfast of all is encouraging guidance.” So, we acknowledge all those whose guidance and encouragement served as a beacon light and crowned my efforts with success.

We would like to thank our principal **Prof. Niranjan N. Chiplunkar** firstly, for providing us with this unique opportunity to do the mini project in the 7th semester of Computer Science and Engineering.

We would like to thank my college administration for providing a conducive environment and also suitable facilities for this mini project. We would like to thank our HOD **Dr. Jyothi Shetty** for showing me the path and providing the inspiration required for taking the project to its completion. It is my great pleasure to thank my mentor **Dr. Pallavi K.N** for her continuous encouragement, guidance, and support throughout this project.

Finally, thanks to staff members of the department of CSE, my parents and friends for their honest opinions and suggestions throughout the course of our mini project.

Elton Arthur (4NM20CS068)
Emmanuel Joshy (4NM20CS069)
Harshith Rao (4NM20CS076)

TABLE OF CONTENTS:

1	ABSTRACT	5
2	INTRODUCTION	6
3	PHASES OF COMPILER	7
4	DESIGN	8-12
5	RESULTS	13-14
6	CONCLUSION	15
7	REFERENCES	15

ABSTRACT

The purpose of this project is to design a lexical analyzer and syntax analyzer for Context Free Grammar. The two stages are the integral part of the 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 is syntactically proper based on given production rules. The main program takes in two parts 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.

We generate the parse table which has entries for each terminal and non-terminal identified in them. Before the generation of the parse table, we identified the FIRST and FOLLOW", of each terminal using a recursive method. The final stage is the parsing which is done by using the standard LL(1) parsing steps.

INTRODUCTION

We know that a compiler 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 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 machine code.

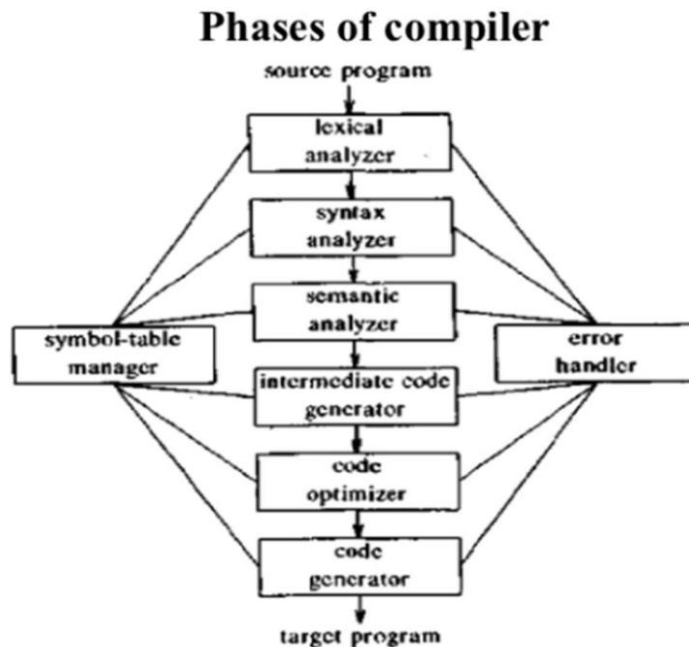
Therefore, we have instruction in terms of an equivalent binary code representation.

For all programs 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 high level 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,

- High level languages are easier for a human being to understand.
- Modifying or updating the code becomes easier providing flexibility.
- 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.
- The programs written in these languages are shorter when compared to those written in machine code.

PHASES OF A COMPILER



1. Lexical Analyzer:

The lexical analyzer, or lexer, scans the source code and breaks it down into tokens, which are the fundamental units of meaning. It removes unnecessary whitespace and comments, producing a stream of tokens for further processing.

2. Syntax Analyzer:

The syntax analyzer, or parser, examines the structure of the source code to ensure it adheres to the grammatical rules of the programming language. It generates a hierarchical representation of the code's syntactic structure, such as a parse tree or an abstract syntax tree (AST).

3. Semantics Analyzer:

The semantics analyzer validates the meaning of the source code, checking for semantic errors and ensuring that the code complies with the language's rules and conventions. It performs tasks such as type checking and generates a symbol table to manage program entities.

4. Intermediate Code Generator:

The intermediate code generator translates the validated source code into an intermediate representation that is independent of the target machine. This step facilitates further optimization and simplifies the process of generating machine code for different architectures.

5. Code Optimizer:

The code optimizer enhances the intermediate code to improve the performance of the final executable. It applies various techniques to eliminate redundancies, reduce execution time, and conserve system resources, aiming to produce more efficient and streamlined code.

6. Code Generator:

The code generator transforms the optimized intermediate code into the machine code or assembly code specific to the target architecture. It maps high-level language constructs to low-level instructions, producing an executable file that can be run on the intended hardware platform.

DESIGN

LEXICAL ANALYSER:

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. • The stream of tokens is sent to the parser for syntax analysis. • It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

TOKENS:

A token is a pair consisting of a token name and an optional attribute value. • The token name is an abstract symbol representing a kind of lexical unit, • e.g., a particular keyword, or a sequence of input characters denoting an identifier. • The token names are the input symbols that the parser processes. • We shall generally write the name of a token in boldface. We will often refer to a token by its token name.

SYNTAX ANALYSER:

The syntax analyzer, also known as a parser, is a crucial phase in the compilation process of a programming language. It examines the syntactic structure of the source code, utilizing a formal grammar to ensure adherence to the language's rules. Taking the stream of tokens generated by the lexical analyzer as input, the syntax analyzer produces a parse tree or abstract syntax tree (AST), providing a hierarchical representation of the code's structure. Its key tasks include error detection, reporting syntax errors, resolving ambiguities, and generating a concise abstract representation of the code. The output of the syntax analyzer serves as the foundation for subsequent stages in the compilation process, facilitating further analysis and code generation.

PROBLEM STATEMENT

Design compiler for the following hypothetical languages Input file: program.txt

```
int main()
begin int n;
do
    expr=expr+expr;
    n=exp;
while(exp) return(n)
end
```

Grammar:

```
S -> A B,
A -> t m
B -> b M W r ( id ) d
M -> t ID ; | #
ID -> cm id ID | id ID | #
E1 -> e a e E ;
E -> o e E | #
E2 -> id a e E ;
W -> do ME w ( C ) | #
C -> e E C1,
C1 -> a a e E | #
ME -> E1 ME | E2 ME | #
```

PARSING TABLE:

```
['t', 'm', 'b', 'd', 'r', 'e', 'o', ';', 'id', 'a', 'do', 'w', '(', ')', 'cm', '$']
['S', 'S->A B', '', '', '', '', '', '', '', '', '', '', '', '', '', '']

['A', 'A->t m', '', '', '', '', '', '', '', '', '', '', '', '', '', '']

['B', '', '', 'B->b M W r ( id ) d', '', '', '', '', '', '', '', '', '', '', '']

['M', 'M->t ID ;', '', '', '', 'M->#', '', '', '', '', '', 'M->#', '', '', '', '']

['ID', '', '', '', '', '', '', 'ID->#', 'ID->id ID', '', '', '', '', 'ID->cm id ID', '']

['E1', '', '', '', '', 'E1->e a e E ;', '', '', '', '', '', '', '', '', '']

['E', '', '', '', '', 'E->o e E', 'E->#', '', 'E->#', '', '', 'E->#', '', '']

['E2', '', '', '', '', 'E2->id a e E ;', '', '', '', '', '', '', '', '']

['W', '', '', '', 'W->#', '', '', 'W->do ME w ( C )', '', '', '', '', '']

['C', '', '', 'C->e E C1', '', '', 'C1->a a e E', 'C1->#', '', '']

['C1', '', '', 'C1->a a e E', 'C1->#', '', '']

['ME', '', 'ME->E1 ME', 'ME->E2 ME', 'ME->#', '']
```

IMPLEMENTATION

Lexical Analyzer:

```
print("----- LEXICAL ANALYZER----- ")
print("Generating tokens")
tokentable_global = []
tokentable_global.append(["Token No", "Lexeme", "Token", "Line No"])
txt = open("question.txt", "r")
tokens = txt.read()
count = 0

tkncount = 0
delimit_flag = 0
program = tokens.split("\n")
for line in program:
    err = 0
    prevct = tkncount
    count = count + 1
    tokentable_local = []
    tokens = line
    tokens = re.findall(r"[A-Za-z0-9_]+|[0-9]+|[(){}]|\\s", tokens)
    tokentable = []
    tokentable.append(["Lexeme", "Token"])
    for token in tokens:
        if isDelimiter(token):
            if token in ["{", "}", "(", ")", ";", ","]:
                tkncount += 1
                tokentable.append([token, "Delimiter"])
                tokentable_local.append([tkncount, token, "Delimiter", count])
            if token in [",;"]:
```

```
print("\nGlobal Token Table: ")
for i in tokentable_global:
    print(i[1], "\t\t", i[2], "\t\t", i[3])
mth_flag = 0
# For the parser tool mth_flag=0 with open('tokens.txt', 'w') as f: str_to_l
mth_flag = 0
with open('tokens.txt', 'w') as f:
    str_to_load = ""
    for token in tokentable_global[1:]:
        if mth_flag > 0:
            mth_flag -= 1
            continue
        if kwd_dict.get(token[1]) is not None:
            sym = kwd_dict.get(token[1])
        elif re.match("^[a-zA-Z][a-zA-Z0-9_]*", token[1]):
            sym = "id"
        else:
            sym = token[1]
        str_to_load += str(sym) + " "
        if token[1] == "main":
            mth_flag = 2
    f.write(f"{str_to_load}\n")
```

Syntax Analyzer:

```
print("-----Syntax Analysis (Parser)-----")
with open('tokens.txt', 'r+') as f:
    for line in f.readlines():
        inps += line
    sample_input_string = inps
    diction = {}
    firsts = {}
    follows = {}
    start_symbol = "S"
    computeFirstsAndFollows()
    (parsing_table, result, tabTerm) = createParseTable()
    if sample_input_string != None:
        print("-----Semantic Analysis:-----")
        validity = validateStringUsingStackAndParseTable(
            parsing_table, result, tabTerm, sample_input_string, term_userdef, start_symbol)
        print(validity)
    else:
        print("\nNo input String detected")
```

```
def computeFirstsAndFollows():
    global rules, nonterm_userdef, term_userdef, diction, firsts, follows

    # Calculate First sets
    for rule in rules:
        k = rule.split("->")
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs

    print(f"\nRules: \n")
    for y in diction:
        print(f"{y}->{diction[y]}")
    print(f"\nAfter elimination of left recursion and After left factoring:\n ")
    diction = eliminateLeftRecursionAndLeftFactoring(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")

    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
```

RESULTS

From Lexical Analyzer:

Global Token Table:

Lexeme	Token	Line No	
int	Keyword	1	
main	Keyword	1	
(Delimiter	1	
)	Delimiter	1	
begin	Keyword	1	
int	Keyword	2	
n	Identifier	2	
;	Delimiter	2	
do	Keyword	3	
expr	Identifier	4	
=	Assignment Operator		4
expr	Identifier	4	
+	Arithmetic Operator		4
expr	Identifier	4	
;	Delimiter	4	
n	Identifier	5	
=	Assignment Operator		5
exp	Identifier	5	
;	Delimiter	5	
while	Keyword	6	
(Delimiter	6	
exp	Identifier	6	
)	Delimiter	6	
return	Keyword	7	
(Delimiter	7	
n	Identifier	7	
)	Delimiter	7	
end	Keyword	8	

From Syntax Analyzer:

Validate String => t m b t id ; do e a e o e ; id a e ; w (e) r (id) d

Stack	Input	Action
S\$	\$d)id(r)e(w;eaid;eoeaedo;idtbmt	T[S][t]=S->A B
AB\$	\$d)id(r)e(w;eaid;eoeaedo;idtbmt	T[A][t]=A->t m
tmB\$	\$d)id(r)e(w;eaid;eoeaedo;idtbmt	Matched:t
mB\$	\$d)id(r)e(w;eaid;eoeaedo;idtbm	Matched:m
B\$	\$d)id(r)e(w;eaid;eoeaedo;idtb	T[B][b]=B->b M W r (id) d
bMWr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;idtb	Matched:b
MWr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;idt	T[M][t]=M->t ID ;
tID;Wr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;idt	Matched:t
ID;Wr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;id	T[ID][id]=ID->id ID
idID;Wr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;id	Matched:id
ID;Wr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;	T[ID][;]=ID->#
;Wr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo;	Matched;;
Wr(id)d\$	\$d)id(r)e(w;eaid;eoeaedo	T[W][do]=W->do ME w (C)
doMEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eoeaedo	Matched:do
MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eoeae	T[ME][e]=ME->E1 ME
E1MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eoeae	T[E1][e]=E1->e a e E ;
eaeE;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eoeae	Matched:e
aeE;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eoea	Matched:a
eE;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eoe	Matched:e
E;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eo	T[E][o]=E->o e E
oeE;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;eo	Matched:o
eE;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;e	Matched:e
E;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;	T[E][;]=E->#
;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid;	Matched;;
MEw(C)r(id)d\$	\$d)id(r)e(w;eaid	T[ME][id]=ME->E2 ME
E2MEw(C)r(id)d\$	\$d)id(r)e(w;eaid	T[E2][id]=E2->id a e E ;
idaaeE;MEw(C)r(id)d\$	\$d)id(r)e(w;eaid	Matched:id
aeE;MEw(C)r(id)d\$	\$d)id(r)e(w;ea	Matched:a
eE;MEw(C)r(id)d\$	\$d)id(r)e(w;e	Matched:e
E;MEw(C)r(id)d\$	\$d)id(r)e(w;	T[E][;]=E->#
;MEw(C)r(id)d\$	\$d)id(r)e(w;	Matched;;
MEw(C)r(id)d\$	\$d)id(r)e(w	T[ME][w]=ME->#
w(C)r(id)d\$	\$d)id(r)e(w	Matched:w
(C)r(id)d\$	\$d)id(r)e(Matched:(
C)r(id)d\$	\$d)id(r)e	T[C][e]=C->e E C1
eEC1)r(id)d\$	\$d)id(r)e	Matched:e
EC1)r(id)d\$	\$d)id(r)	T[E][;]=E->#
C1)r(id)d\$	\$d)id(r)	T[C1][;]=C1->#
)r(id)d\$	\$d)id(r)	Matched:)
r(id)d\$	\$d)id(r	Matched:r
(id)d\$	\$d)id(Matched:(
id)d\$	\$d)id	Matched:id
)d\$	\$d)	Matched:)
d\$	\$d	Matched:d
\$	\$	Valid

CONCLUSION

The lexical analyzer takes the file with the code as input and returns the tokens accordingly. In lexical analysis, which is the first phase of a compiler, when a program is given as input, it is parsed and tokens are generated. In syntax analysis, which is the second phase of a compiler, the input program is checked for its syntax against the grammar provided. The tokens of the input string are used for parsing.

REFERENCES

1. <https://www.geeksforgeeks.org/closure-properties-of-context-free-languages/>
2. <https://www.geeksforgeeks.org/program-calculate-first-follow-sets-givengrammar/>
3. <https://stackoverflow.com/questions/52948675/coding-first-and-follow-sets-in-acfg>

