**COMPILER DESIGN PROJECT**

A Mini Project Report Submitted by

ANISHA SHETTY ANUSHA JAGANNATH SOORENJI

(4NM17CS019) (4NM17CS027)

UNDER THE GUIDANCE OF

**Dr.JYOTHI SHETTY**

Professor

Department of Computer Science and Engineering

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

Bachelor of Engineering in

Computer Science & Engineering

from

Visveshvaraya 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 2020**



(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

“Compiler Design Project”

is a bonafide work carried out by

ANISHA SHETTY

(4NM17CS019)

ANUSHA JAGANNATH SOORENJI

(4NM17CS027)

in partial fulfillment 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 2019-2020.

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 Signatureof HOD

i

# **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 **Dr.Jyothi Shetty**, Professor, 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.

Anisha Shetty (4NM17CS019)

Anusha Jagannath Soorenji (4NM17CS027)

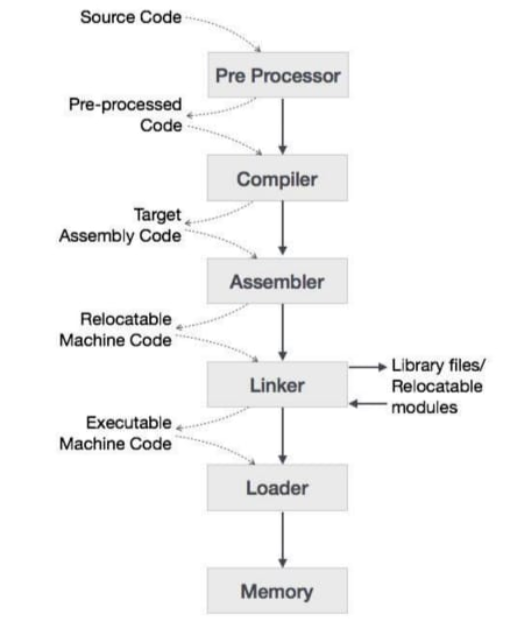
# 

ii

**ABSTRACT**

Computers are balanced mixture of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is counterpart of binary language in software programming. Binary language has only two characters ‘0’ and ‘1’ which forms the low level language. Compiler is a software which converts a program written in high level language(Source Language) to low level language(Object/Target/Machine Language).

LANGUAGE PROCESSING SYSTEM



**TABLE OF CONTENTS**

TITLE PAGE NUMBER

• INTRODUCTION : 2 - 4

• TOKENS : 5

• GRAMMAR : 6

• TRANSITION DIAGRAM : 7

• PARSE TABLE : 8

• ROLE OF PARSER : 9-11

• IMPLEMENTATION : 12-13

• RESULT & CONCLUSION : 14-16

**INTRODUCTION**

What is a compiler?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed.

The main reasons for this are:

• Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.

• The compiler can spot some obvious programming mistakes.

• Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

• Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

• On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some timecritical programs are still written partly in machine language.

• A good compiler will, however, be able to get very close to the speed of handwritten machine code when translating well-structured programs.

**THE PHASES OF A COMPILER**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

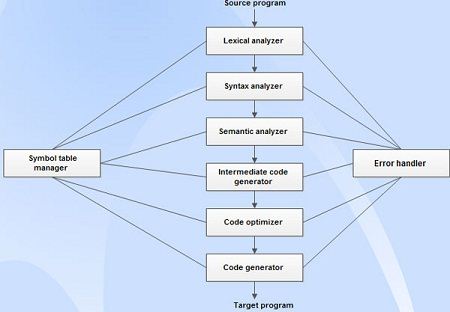


Fig: Phases of Compiler

**Lexical Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

**<token-name, attribute-value>**

**Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this 3 phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

**Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

**Intermediate Code Generation**

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

**Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

**Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

**SYMBOL TABLE**

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers’ names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

**DESIGN**

**Token**:

Token is a sequence of characters that can be treated as a single logical entity.

Typical tokens are,

1. Identifiers
2. keywords
3. operators
4. special symbols
5. constants

**Pattern**:

A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

**Lexeme**:

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

**Problem statement:**

Design a compiler for the following pseudocode

int main()

{

do

{

var=expr+expr;

var=exp;

}

while(T);

return(num);

}

Tokens for above problem statement are:

int,main,(,),{,},do,var,=,expr,+,exp,while,T,return,num,;

**GRAMMAR FOR ABOVE PROBLEM STATEMENT IS:**

S->D MAI

MAI->MAIN ( ) { ST RE }

ST->DO { EXP } WHILE ( COND ) SC

EXP->EX EXPS

EX->ID OR VAR OP VAR SC

EXPS->ID OR VARC SC

RE->RETURN ( NUM ) SC

D->int

MAIN->main

DO->do

COND->T

VAR->expr

ID->var

OP->+

OR->=

WHILE->while

SC->;

NUM->num

RETURN->return

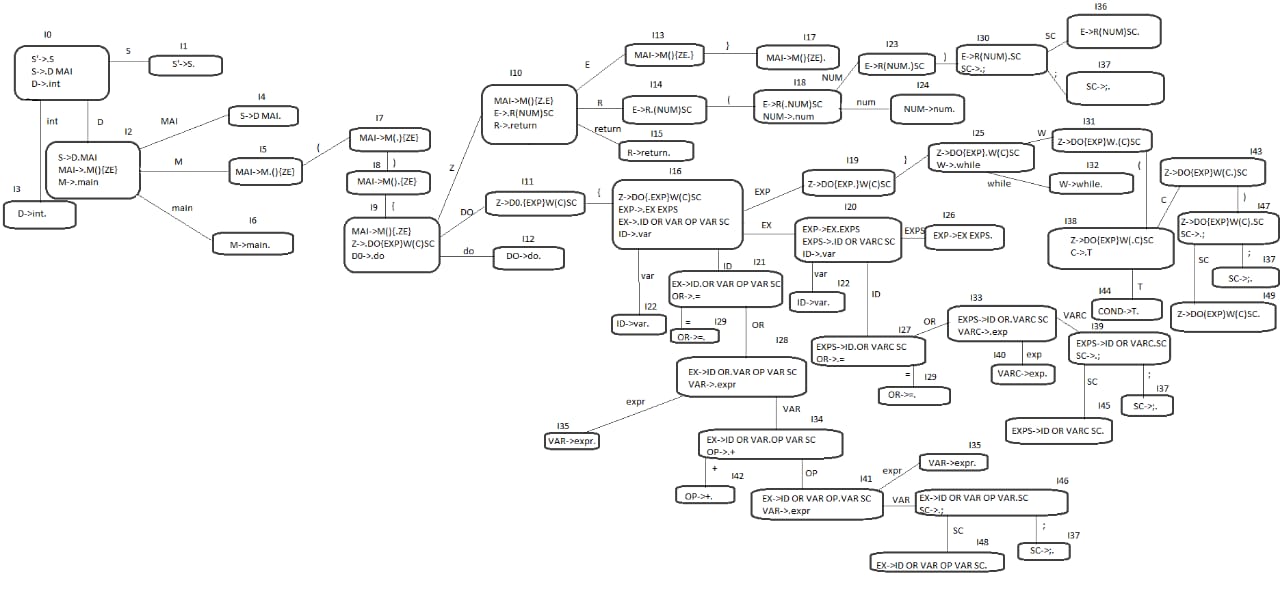
VARC->exp

**TRANSITION DIAGRAM FOR ABOVE GRAMMAR**

Grammar has following components

• A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

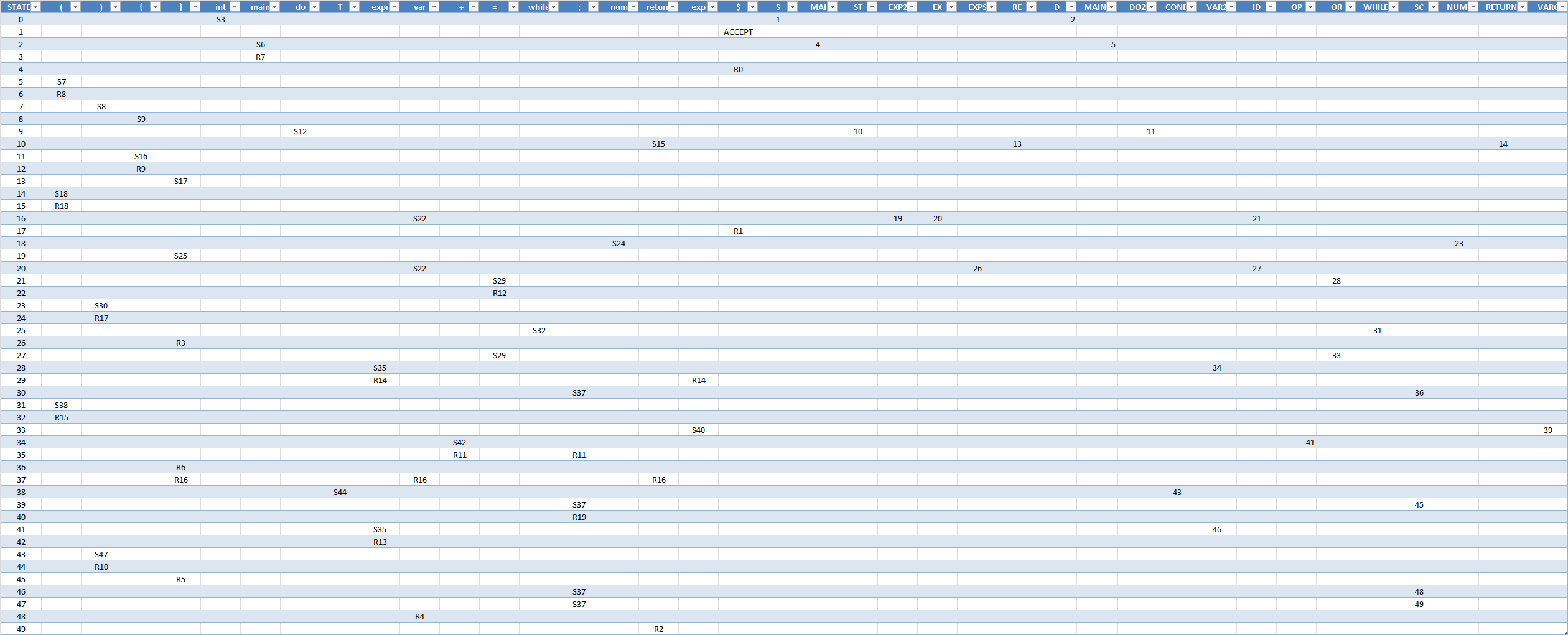
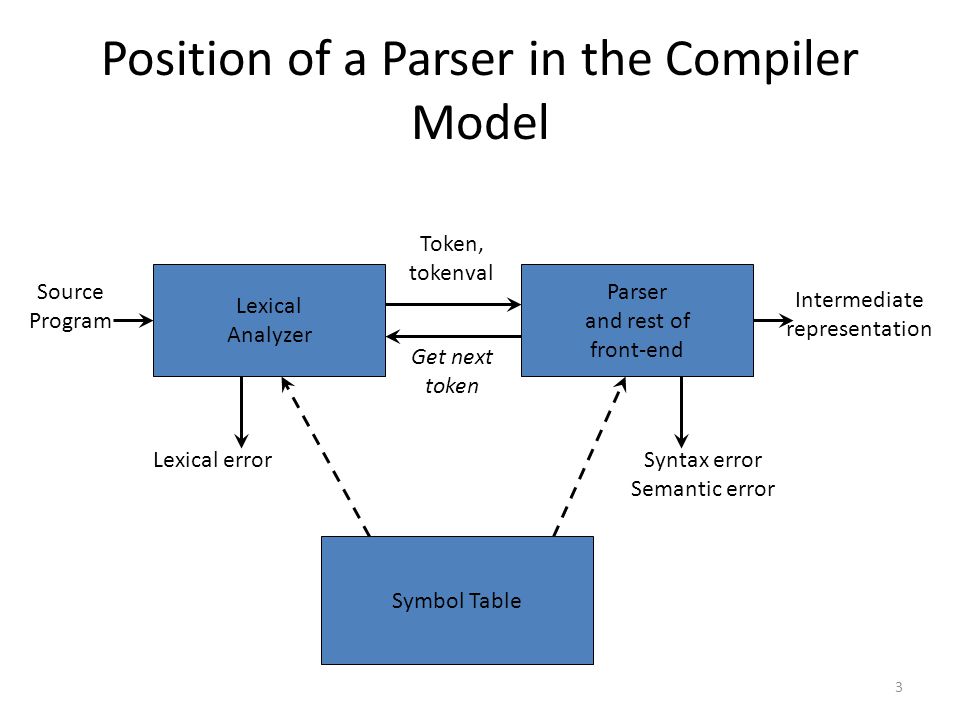
• A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production. The transition diagram is as follows:



**PARSE TABLE FOR THE ABOVE GRAMMAR IS:**

**Construction of SLR parsing table –**

1. Construct C = { I0, I1, ……. In}, the collection of sets of LR(0) items for G’.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :
   1. If [ A -> ?.a? ] is in Ii and GOTO(Ii , a) = Ij , then set ACTION[i, a] to “shift j”. Here a must be terminal.
   2. If [A -> ?.] is in Ii, then set ACTION[i, a] to “reduce A -> ?” for all a in FOLLOW(A); here A may not be S’.
   3. Is [S -> S.] is in Ii, then set action[i, $] to “accept”. If any conflicting actions are generated by the above rules we say that the grammar is not SLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule:  
   if GOTO( Ii , A ) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

There are three general types’ parsers for grammars.

Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley’s algorithm can parse any grammar. These methods are too inefficient to use in production compilers.

· The methods commonly used in compilers are classified as either top-down parsing or bottom-up parsing.

· Top-down parsers build parse trees from the top (root) to the bottom (leaves).

· Bottom-up parsers build parse trees from the leaves and work up to the root.

· In both case input to the parser is scanned from left to right, one symbol at a time.

. The output of the parser is some representation of the parse tree for the stream of tokens.

· There are number of tasks that might be conducted during parsing. Such as;

**Bottom-up Parsing**

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

**LR(0) Grammars**

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. In the SLR (1) parsing, we place the reduce move only in the follow of left hand side. Various steps involved in the SLR (1) Parsing:

* For the given input string write a context free grammar
* Check the ambiguity of the grammar
* Add Augment production in the given grammar
* Create Canonical collection of LR (0) items
* Draw a data flow diagram (transition diagram)
* Construct a SLR (1) parsing table

**First and Follow Sets**

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

**First Set**

This set is created to know what terminal symbol is derived in the first position by a nonterminal. For example,

α → t β

That is, α derives t (terminal) in the very first position. So, t ∈ FIRST(α).

**Algorithm for Calculating First Set**

Look at the definition of FIRST(α) set:

 if α is a terminal, then FIRST(α) = { α }.

 if α is a non -terminal and α → ℇ is a production, then FIRST(α) = { ℇ }.

 if α is a non -terminal and α → 𝜸1 𝜸2 𝜸3 … 𝜸n and any FIRST(𝜸) contains t, then t is in FIRST(α).

First set can be seen as: FIRST(α) = { t | α →\* t β } ∪ { ℇ | α →\* ε}

**FOLLOW SET**

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

**Algorithm for Calculating Follow Set:**

 if α is a start symbol, then FOLLOW() = $

 if α is a non -terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except ℇ.

 if α is a non -terminal and has a production α → AB, where B ℇ, then FOLLOW(A) is in FOLLOW(α).

Follow set can be seen as: FOLLOW(α) = { t | S \*αt\*}

**First and Follow for the above problem statement is:**

FIRST(S)={int} FIRST(VAR)={expr}

FIRST(MAI)={main} FIRST(ID)={var}

FIRST(ST)={do} FIRST(OP)={+}

FIRST(EXP)={var} FIRST(OR)={=}

FIRST(EX)={var} FIRST(WHILE)= {while}

FIRST(EXPS)={var} FIRST(SC)={;}

FIRST(RE)={return} FIRST(NUM)={num}

FIRST(D)={int} FIRST(RETURN)={return}

FIRST(MAIN)={main} FIRST(VARC)={exp}

FIRST(DO)={do}

FIRST(COND)={T}

FOLLOW(S1)={$} FOLLOW(COND)={)}

FOLLOW(S)={$} FOLLOW(VAR)={+ ;}

FOLLOW(MAI)={$} FOLLOW(ID)={=}

FOLLOW(ST)={return} FOLLOW(OP)={expr}

FOLLOW(EXP)={}} FOLLOW(OR)={expr,exp}

FOLLOW(EX)={var} FOLLOW(WHILE)={(}

FOLLOW(EXPS)={}} FOLLOW(SC)={return,var,}}

FOLLOW(RE)={}} FOLLOW(NUM)={)}

FOLLOW(D)={main} FOLLOW(RETURN)={(}

FOLLOW(MAIN)={(} FOLLOW(VARC)={;}

FOLLOW(DO)={{}

**IMPORTANT FUNCTION CODE USED IN IMPLEMENTATION**

**LEXICAL FUNCTION:**

#tokenizer method

def tokenizer(program\_string):

program\_string+='$'

pointer=0

lis=[]

while program\_string[pointer] != "$":

current\_string=""

if program\_string[pointer].isalpha():

current\_string+=program\_string[pointer]

pointer+=1

while program\_string[pointer].isalnum() or program\_string[pointer]=="\_":

current\_string+=program\_string[pointer]

pointer+=1

if current\_string in tokens:

lis.append(tokens[tokens.index(current\_string)])

else:

lis.append('id')

elif program\_string[pointer].isnumeric():

current\_string+=program\_string[pointer]

pointer+=1

dot\_count=0

while program\_string[pointer].isnumeric() or program\_string[pointer]==".":

if program\_string[pointer]==".":

dot\_count+=1

current\_string+=program\_string[pointer]

pointer+=1

if dot\_count>1:

x=error\_handler(pointer)

print('Error in Tokenizer Line No. : '+str(x)+' unidentified token '+current\_string)

exit()

else:

lis.append('num')

else:

if program\_string[pointer]!=" " and program\_string[pointer]!="\n":

lis.append(tokens[tokens.index(program\_string[pointer])])

pointer+=1

else:

pointer+=1

return lis

**SYNTAX ANAYSER**

def syntax\_analyser(token\_list):

top=0

stack\_token=['0']

j=len(token\_list)

i=0

while(i<j):

x=parse\_table[stack\_token[top]][token\_list[i]].split()

if(x[0]=='S'):

print('shift')

stack\_token.append(token\_list[i])

stack\_token.append(x[1])

top+=2

i+=1

print(stack\_token)

elif(x[0]=='R'):

print('reduce')

stack\_token=stack\_token[:(top-no\_ele[x[1]])+1]

top-=no\_ele[x[1]]

stack\_token.append(gram\_name[x[1]])

top+=1

stack\_token.append(parse\_table[stack\_token[top-1]][stack\_token[top]])

top+=1

print(stack\_token)

elif(x[0]=='NA'):

print("syntax error..!")

exit()

elif(x[0]=='A'):

print("accepted..!\n")

exit()

else:

break;

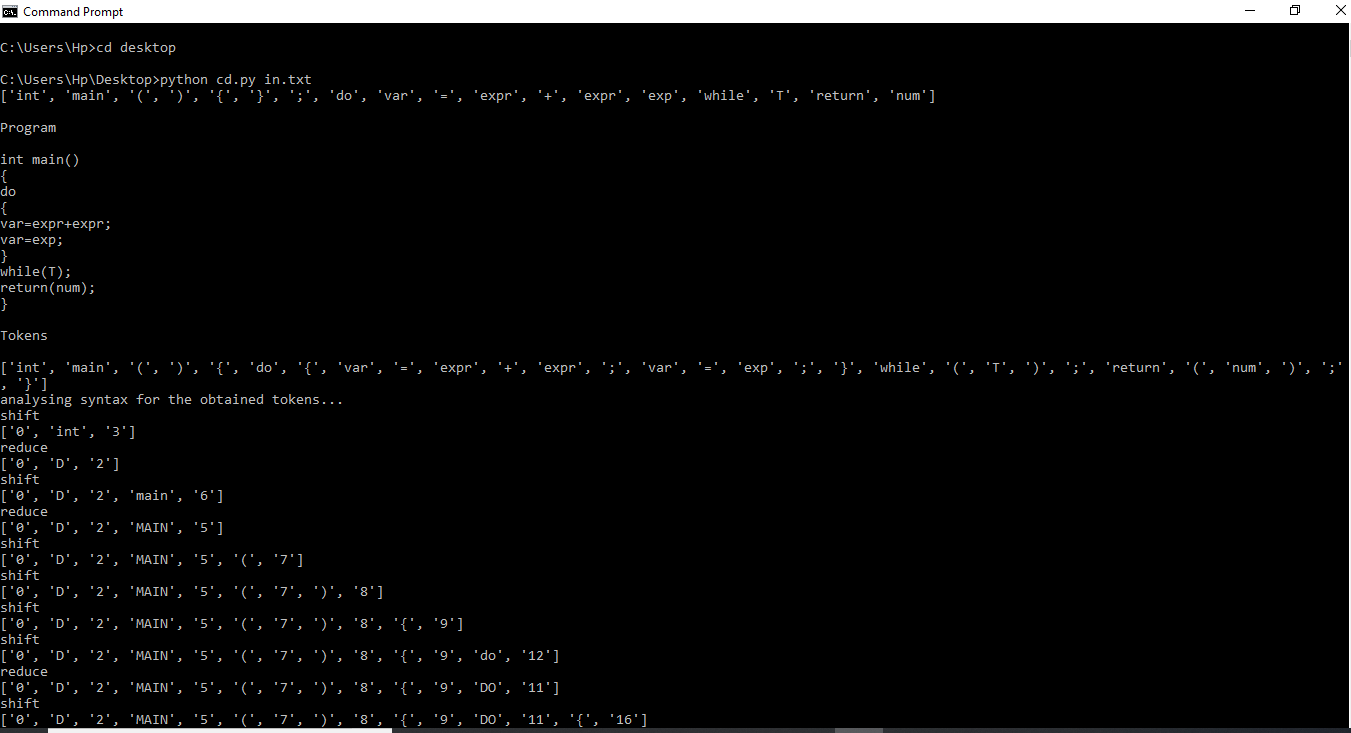
**Explanation for the above code**

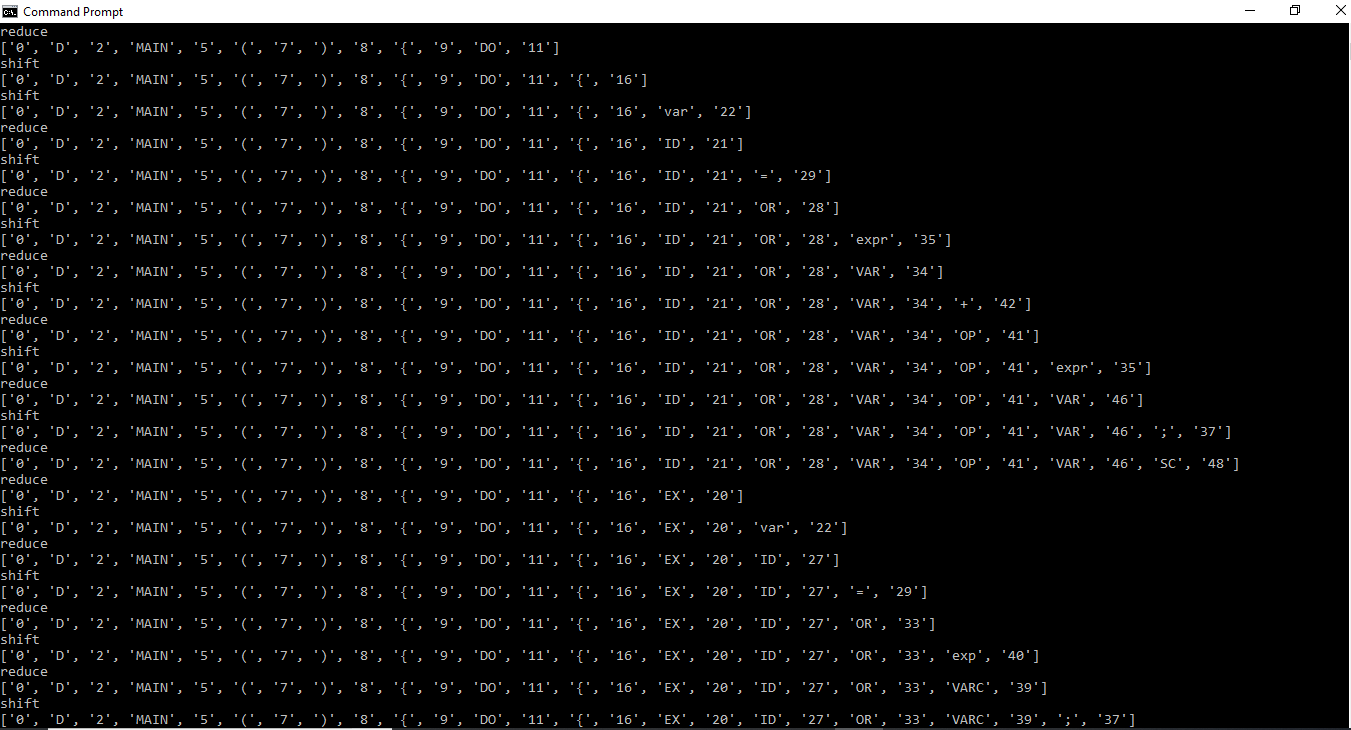
In the above python code , an array of keywords are used in the program have been declared.

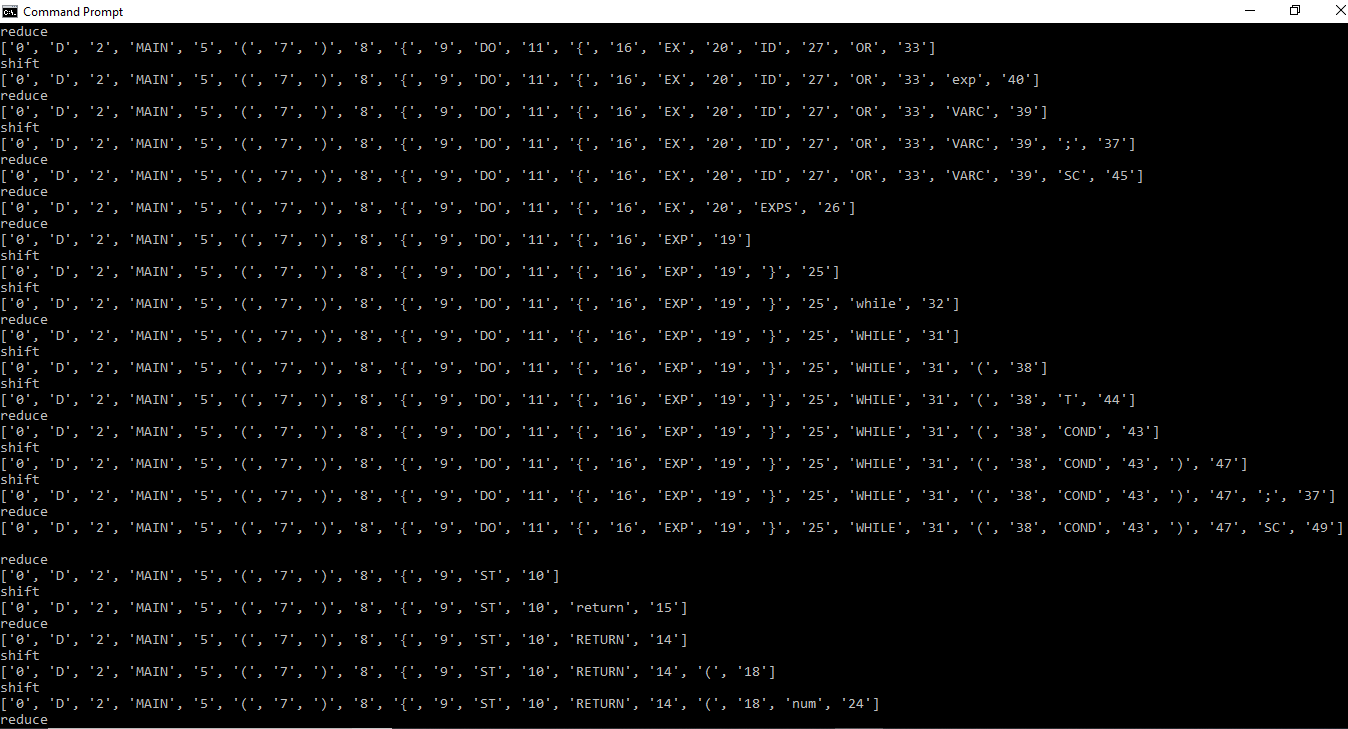
When a file containing the problem statement is given as input to the program, the tokenizer function divides the input with space,semicolon and comma as delimiter and assigns it to a variable named current\_string everytime in the loop.

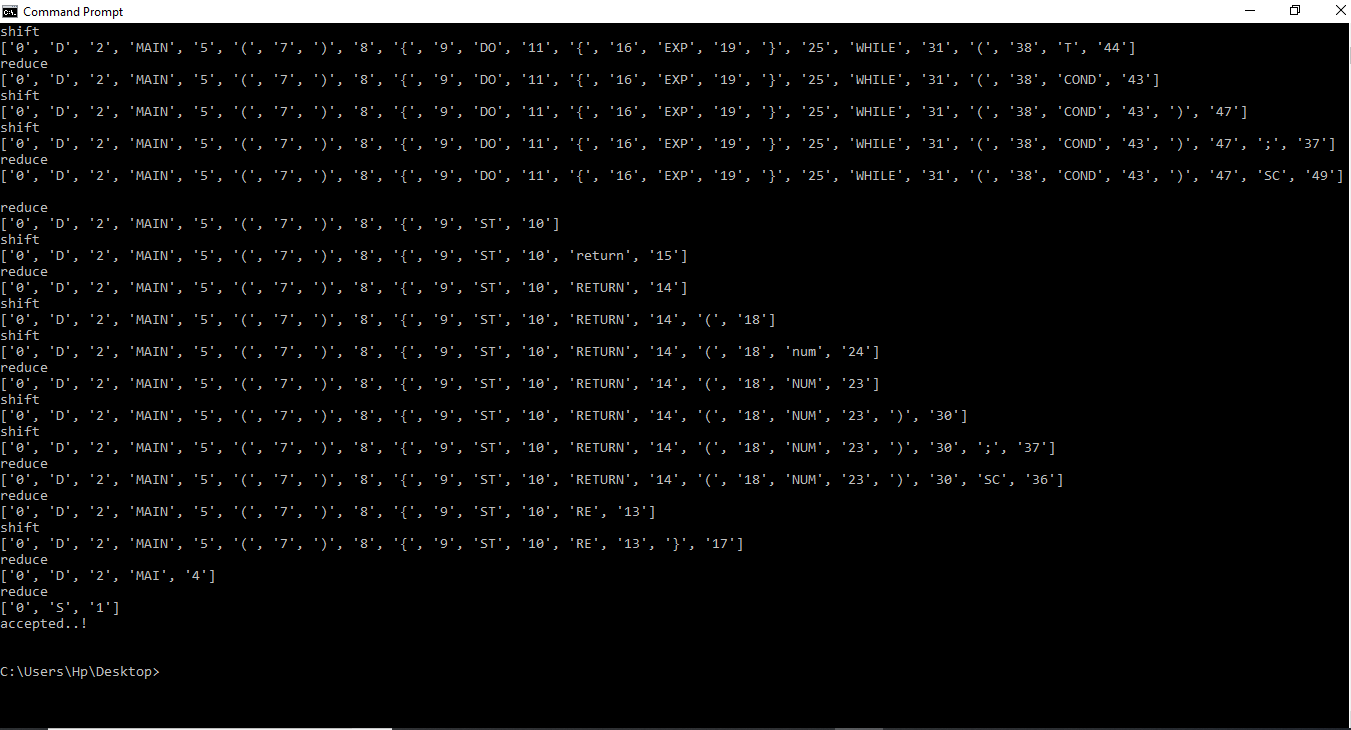
Every time the token is compared with each keyword in the array and if it matches any of them,then it is considered as a keyword. In syntax analyser function parsing table is declared int the program.The tokens are grouped as a single input.If action [ S,a ]=shift S1 then push S1 on top of the stack.If action[S,a]=Reduce A->b then pop 2\*|b| elements from stack.If action[S,a]=A,then string is accepted.

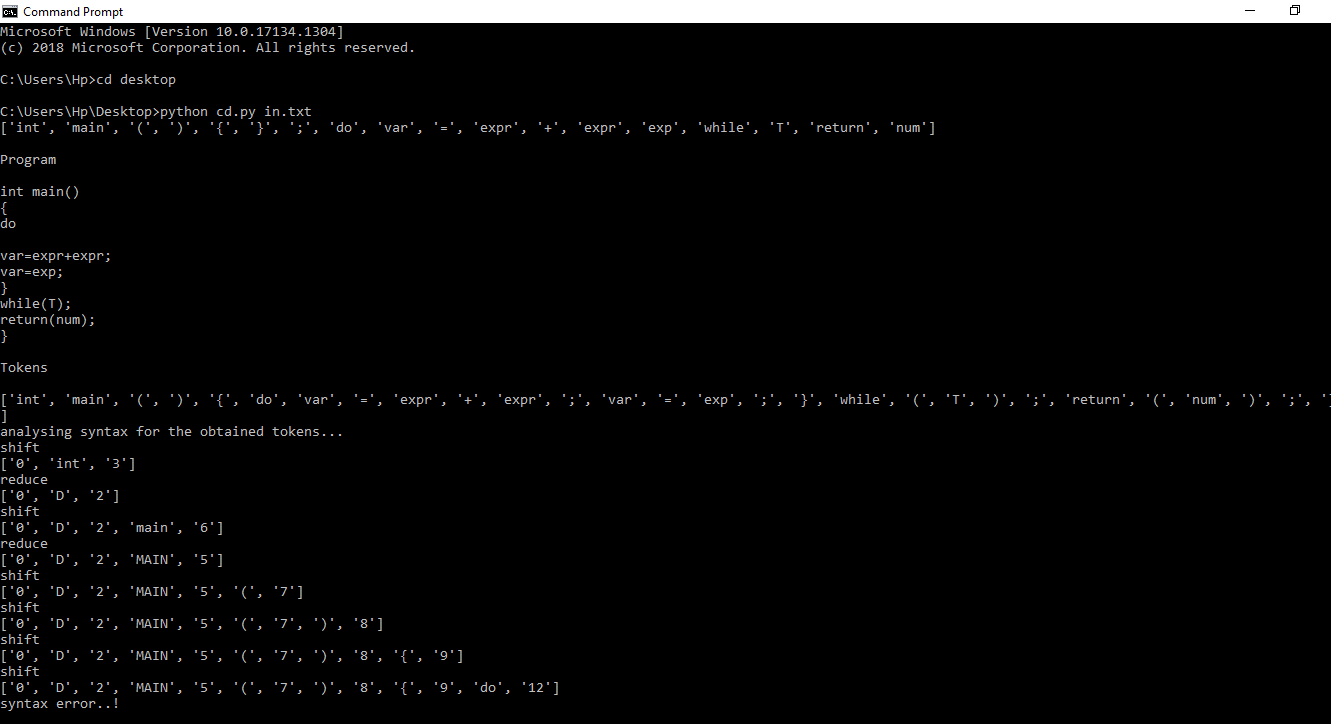
**Output**

****



****



Output for syntax error 

**CONCLUSION**

In lexical analysis when we give a program statement as input ,the keywords

,identifiers ,operators and numbers are displayed. Each of these are the tokens of the

statement.

In bottom up parser, on giving an input string, the string is parsed as per the grammar

and parsing table given in the program. If the string was parsed completely then the

string is accepted in the output else a syntax error is displayed.