**COMPILER DESIGN PROJECT**

A Mini Project Report Submitted by

Jayesh Kumar Deon Victor Lobo

USN:4NM17CS071 USN:4NM17CS054

Section B Section B



UNDER THE GUIDANCE OF

Ms. MINU P ABRAHAM

Assistant Professor

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 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

 Jayesh Kumar Deon Victor Lobo

USN:4NM17CS071 USN:4NM17CS054

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 **Ms. MINU PABRAHAM**,Assistant 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, NitteMahalingaAdyantaya 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.

Jayesh Kumar(4NM17CS071)

Deon Victor Lobo (4NM17CS054)

**TABLE OF CONTENTS**

TITLE PAGE NUMBER

• INTRODUCTION : 5

• LEXICAL ANALYSIS : 8

• LEX CODE : 9

• SYNTAX ANALYSIS : 12

• CONTEXT FREE GRAMMAR : 14

• PARSE TREE : 15

• TYPES OF PARSING : 16

• PARSING TABLE : 21

• PARSER CODE : 24

• CONCLUSION : 32

**INTRRODUCTION**

What is a compiler?

A compiler isa computerprogram that translates computercode written in one programming language (the source language) intoanother language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program .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:

1. Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
2. The compiler can spot some obvious programming mistakes.
3. Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
4. 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.
5. 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.
6. A good compiler will, however, be able to get very close to the speed of handwritten machine code when translating well-structured programs.

# **Phases of 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.



### **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 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.

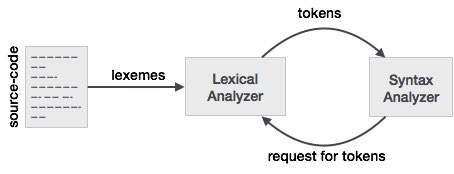
### **Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's 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.

**Lexical Analysis:**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



## **Tokens**

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

intvalue=100;

contains the tokens:

int(keyword),value(identifier),=(operator),100(constant)and;(symbol).

**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:**

X: integer ;

Procedure foo( b : integer )

b := 13;

If x = 12 and b = 13 then

printf( “by copy-in copy-out” );

elseif x = 13 and b = 13 then

printf( “by address” );

else

printf( “ mystery” );

end if;

end foo

**LEXICAL PROGRAM FOR THE ABOVE PROBLEM STATEMENT**

def iskeyword(key):  
 #print(key)  
 keywords=['integer','char','main','Procedure','end','if','else','elseif','then','and','printf']  
  
 status=0  
 for j in keywords:  
 if key==j:  
 status=1  
 return status  
  
  
  
f=open("input.txt","r",encoding="utf8")  
operator=['+','-','\*','/','=']  
buffer=""  
  
while(1):  
 text=f.read(1)  
 if len(text)==0:  
 if len(buffer)!=0:  
 if iskeyword(buffer)==1:  
 print(buffer+" is keyword")  
 buffer=""  
 else:  
 print(buffer+" is identifier")  
 buffer=""  
 print("EOF")  
 break  
 else:  
 for j in operator:  
 if j==text:  
 if len(buffer)!=0:  
 if iskeyword(buffer)==1:  
 print(buffer+" is keyword")  
 buffer=""  
 else:  
 print(buffer+" is identifier")  
 buffer=""  
  
 print(text+" is operator")  
 if text.isalnum():  
 buffer=buffer+text  
 elif ((text==" " or text=="\n") and (len(buffer)!=0)):

#buffer=buffer+'\0'

if iskeyword(buffer)==1:

print(buffer+" is keyword")

buffer=""

elif buffer[0] in '0123456789':

print(buffer+" is number")

buffer=""

else:

print(buffer+" is identifier")

buffer=""

**Output:**

X is identifier

integer is keyword

Procedure is keyword

foo is identifier

b is identifier

integer is keyword

b is identifier

= is operator

13 is number

if is keyword

x is identifier

= is operator

12 is number

and is keyword

b is identifier

= is operator

13 is number

then is keyword

printf is keyword

by is identifier

copy is identifier

- is operator

in is identifier

copy is identifier

- is operator

out is identifier

elseif is keyword

x is identifier

= is operator

13 is number

and is keyword

b is identifier

= is operator

13 is number

then is keyword

printf is keyword

by is identifier

address is identifier

else is keyword

printf is keyword

A is identifier

mystery is identifier

end is keyword

if is keyword

end is keyword

foo is identifier

EOF

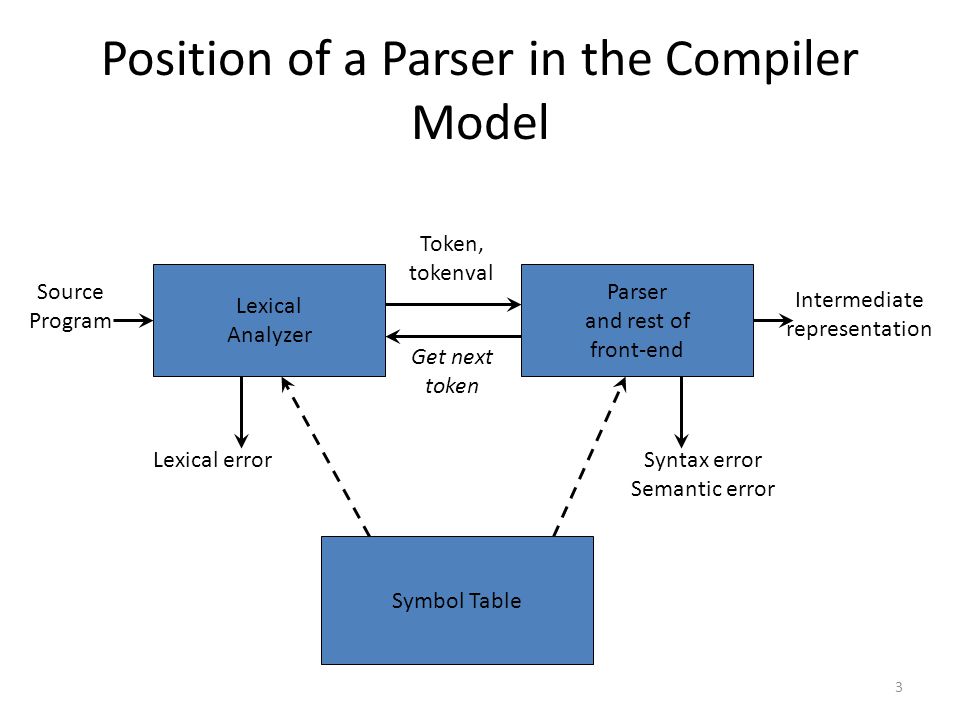
**SYNTAX ANALYSIS**

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

**Role of the Parser**

In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.

The parser returns any syntax error for the source language.



. There are three general types’ parsers for grammars.

· Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley’salgorithmcan 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;

o Collecting information about various tokens into the symbol table.

o Performing type checking and other kinds of semantic analysis.

o Generating intermediate code.

o Syntax Error Handling:

o Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.

o The program can contain errors at many different levels. e.g.,

§ Lexical – such as misspelling an identifier, keyword, or operator.

§ Syntax – such as an arithmetic expression with unbalanced parenthesis.

§ Semantic – such as an operator applied to an incompatible operand.

§ Logical – such as an infinitely recursive call.

. Much of the error detection and recovery in a compiler is centered on the syntax analysis phase.

o One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.

o Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.

· The error handler in a parser has simple goals:

o It should the presence of errors clearly and accurately.

o It should recover from each error quickly enough to be able to detect subsequent errors.

• It should not significantly slow down the processing of correct programs.

Error-Recovery Strategies:

o There are many different general strategies that a parser can employ to recover from a syntactic error.

§ Panic mode

§ Phrase level

§ Error production

§ Global correction

**CONTEXT-FREE GRAMMAR**

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four 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 tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.

• 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.

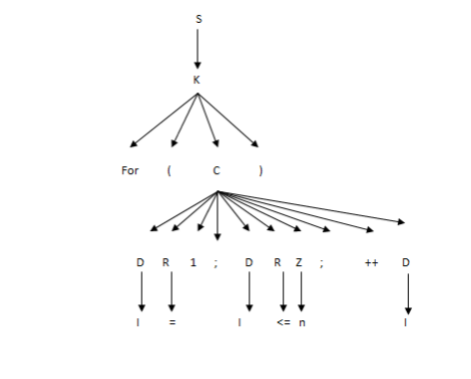
• One of the non-terminals is designated as the start symbol (S); from where the production begins. The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

**Parse Tree**

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b \* c

The left-most derivation is:



In a parse tree:

• All leaf nodes are terminals.

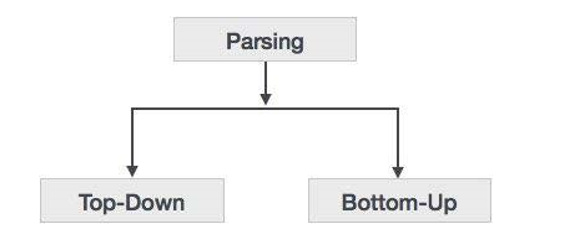
• All interior nodes are non-terminals.

• In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

**Types Of Parser**

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types :topdown parsing and bottom-up parsing.



**Top-down Parsing**

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

 Recursive descent parsing : It is a common form of top-down parsing. It is called recursive, as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.

 Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

**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.

**NOTE:THE TYPE OF PARSER WE HAVE USED IS BOTTOM-UP**  PARSER

**LR(k) Parsing**

The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the \**L**" is for left-to-right scanning of the input, the \**R**" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases k = 0 or k = 1 are of practical interest, and we shall only consider LR parsers with k <= 1 here. When (k) is omitted, k is assumed to be 1.

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of \items." An LR(0) **item** (item for short) of a grammar **G** is a production of **G** with a dot at some position of the body.

Thus, production **A ->XYZ** yields the four items

**A ->.XYZ**

**A ->X.YZ**

**A -> XY.Z**

**A -> XYZ.**

**Grammar for above problem statement is:**

S -> K

K -> D

D -> B : C

D -> B : C SC

SC -> ;

B -> X

B -> x

B -> b

C -> integer

K -> PR F ( D )

PR -> Procedure

F -> foo

K -> I E N E T

I -> if

I -> elseif

E -> B = U

U -> 12

U -> 13

K -> B : = U

N -> and

T -> then

K -> R ( " M " ) SC

R -> printf

M -> Q G - W G - W

Q -> by

G -> copy

W -> in

W -> out

M -> Q AD

AD -> address

K -> else

M -> V Y

V -> A

Y -> mystery

K -> EN I SC

K -> EN F

EN -> end

**AUGMENTED GRAMMAR:**

0: EN -> end

1: S' -> S

2: AD -> address

3: PR -> Procedure

4: C -> integer

5: B -> X

6: B -> x

7: B -> b

8: E -> B = U

9: D -> B : C

10: D -> B : C SC

11: G -> copy

12: F -> foo

13: I -> if

14: I -> elseif

15: K -> D

16: K -> PR F ( D )

17: K -> I E N E T

18: K -> B : = U

19: K -> R ( " M " ) SC

20: K -> else

21: K -> EN I SC

22: K -> EN F

23: M -> Q G - W G - W

24: M -> Q AD

25: M -> V Y

26: N -> and

27: Q -> by

28: S -> K

29: R -> printf

30: U -> 12

31: U -> 13

32: T -> then

33: W -> in

34: W -> out

35: V -> A

36: Y -> mystery

37: SC -> ;

**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:**

**EN = { end }**

**S' = { X , x , b , printf , if , elseif , else , and }**

**AD = { address }**

**PR = { Procedure }**

**C = { integer }**

**B = { X , x , b }**

**E = { X , x , b }**

**D = { X , x , b }**

**G = { copy }**

**F = { foo }**

**I = { if , elseif }**

**K = { X , x , b , printf , if , elseif , else , and }**

**M = { by , A }**

**N = { and }**

**Q = { by }**

**S = { X , x , b , printf , if , elseif , else , and }**

**R = { printf }**

**U = { 12 , 13 }**

**T = { then }**

**W = { in , out }**

**V = { A }**

**Y = { mystery }**

**SC = { ; }**

**FOLLOW:**

**EN = { if , elseif , foo }**

**S' = { $ }**

**AD = { " }**

**PR = { foo }**

**C = { $ , ) , ; }**

**B = { = , : }**

**E = { and }**

**D = { $ , ) }**

**G = { - }**

**F = { ( , $ }**

**I = { X , x , b , ; }**

**K = { $ }**

**M = { " }**

**N = { X , x , b }**

**Q = { copy , address }**

**S = { $ }**

**R = { ( }**

**U = { and , $ }**

**T = { $ }**

**W = { copy }**

**V = { mystery }**

**Y = { " }**

**SC = { $ , ) }**

**PARSING TABLE**

**Construction of SLR parsing table –**

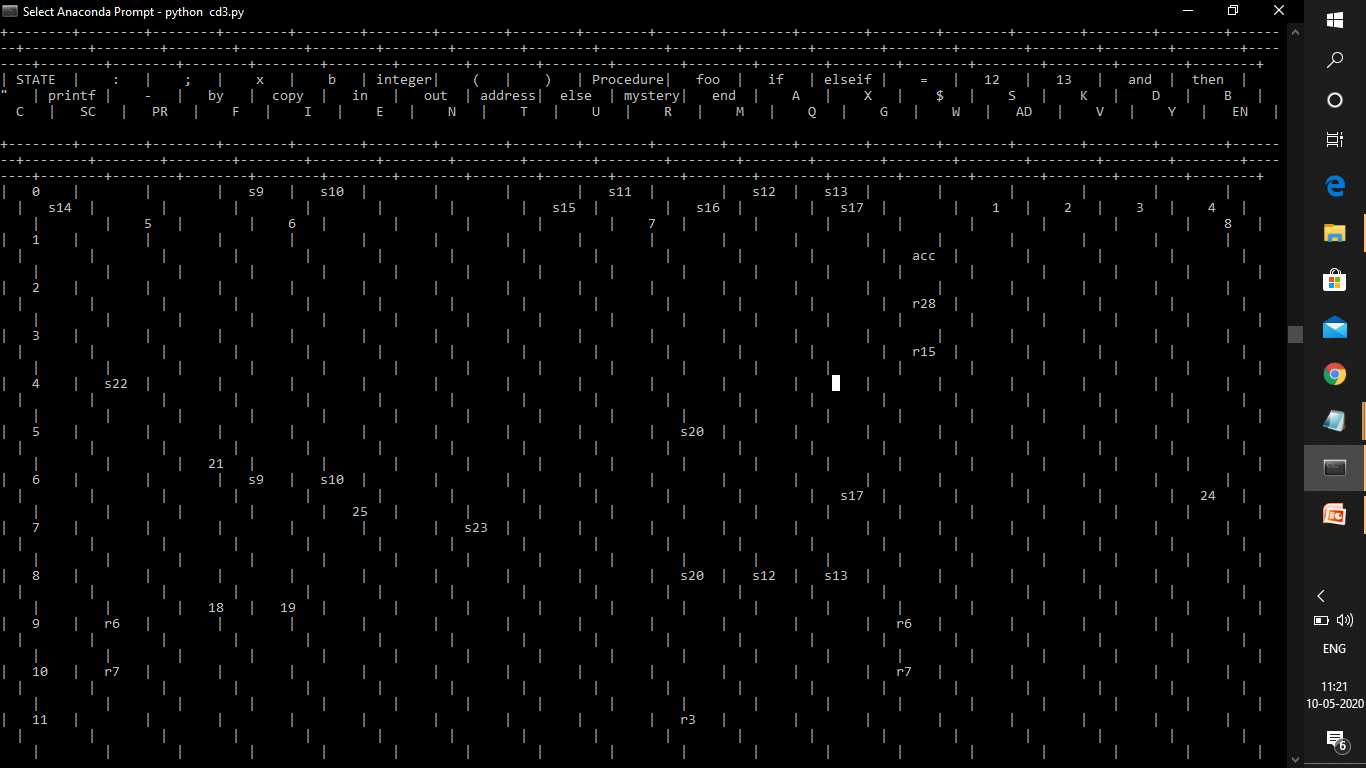
1. Construct C = { I0, I1, ……. In}, the collection of sets of LR(0) items for G’.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :

* If [ A -> ?**.**a? ] is in Ii and GOTO(Ii , a) = Ij , then set ACTION[i, a] to “shift j”. Here a must be terminal.
* 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’.
* 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.

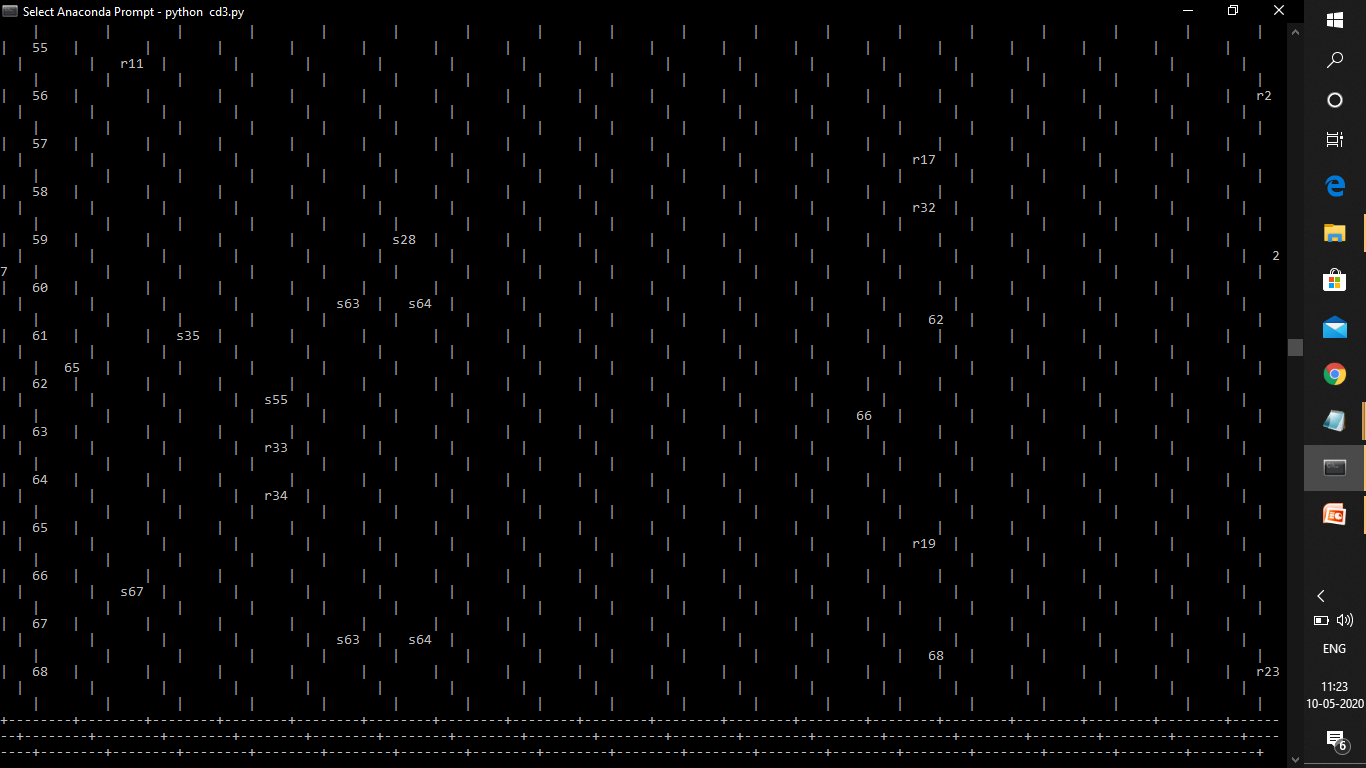
**4.** All entries not defined by rules 2 and 3 are made error.









****

**Parser code for the above problem statement is :**

grammars = open("grammar.txt")  
G = {}  
C = {}  
start = ""  
terminals = []  
nonterminals = []  
symbols = []  
error=0  
  
def parse\_grammar():  
 global G, start, terminals, nonterminals, symbols  
 for line in grammars:  
 line = " ".join(line.split())  
 if line == '\n':  
 break  
 head = line[:line.index("->")].strip()  
 prods = [l.strip().split(' ') for l in ''.join(line[line.index("->") + 2:]).split('|')]  
 if not start:  
 start = head + "'"  
 G[start] = [[head]]  
nonterminals.append(start)  
 if head not in G:  
 G[head] = []  
 if head not in nonterminals:  
nonterminals.append(head)  
 for prod in prods:  
 G[head].append(prod)  
 for char in prod:  
 if not char.isupper() and char not in terminals:  
terminals.append(char)  
elifchar.isupper() and char not in nonterminals:  
nonterminals.append(char)  
 G[char] = [] #non terminals dont produce other symbols

for i in G.copy():

#print(len(value))

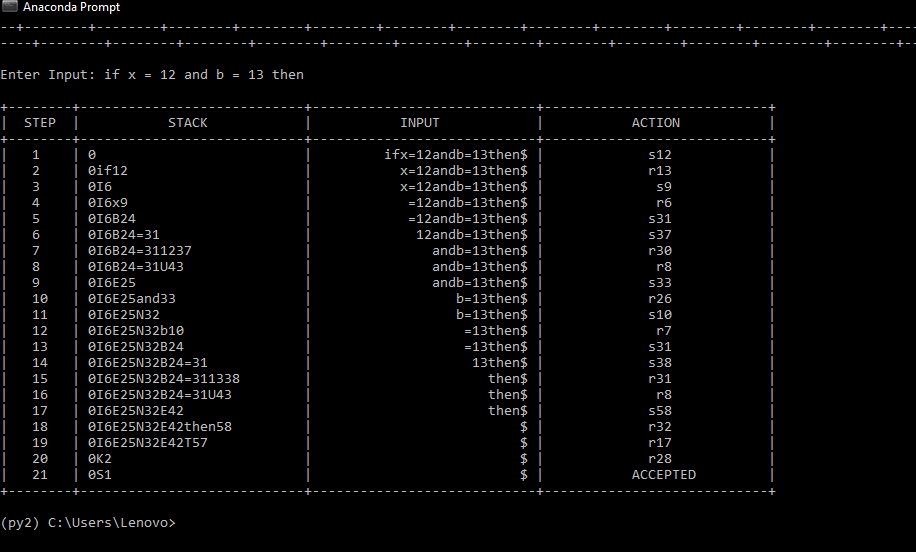
if not G[i]:

G.pop(i)

nonterminals.remove(i)

terminals.append(i)  
symbols = nonterminals+terminals  
first\_seen = []  
  
def FIRST(X):  
 global first\_seen  
 first = []  
first\_seen.append(X)  
 if X in terminals: # CASE 1  
first.append(X)  
elif X in nonterminals:  
 for prods in G[X]: # CASE 2  
 if prods[0] in terminals and prods[0] not in first:  
first.append(prods[0])  
 else: # CASE 3  
 for nonterm in prods:  
 if nonterm not in first\_seen:  
 for terms in FIRST(nonterm):  
 if terms not in first:  
first.append(terms)  
first\_seen.remove(X)  
 return first  
  
  
follow\_seen = []  
def FOLLOW(A):  
 global follow\_seen  
 follow = []  
follow\_seen.append(A)  
 if A == start: # CASE 1  
follow.append('$')  
 for heads in G.keys():  
 for prods in G[heads]:  
follow\_head = False  
 if A in prods:  
next\_symbol\_pos = prods.index(A) + 1  
 if next\_symbol\_pos<len(prods): # CASE 2  
 for terms in FIRST(prods[next\_symbol\_pos]):  
 if terms not in follow:  
follow.append(terms)  
 else: # CASE 3  
follow\_head = True  
 if follow\_head and heads not in follow\_seen:  
 for terms in FOLLOW(heads):  
 if terms not in follow:  
follow.append(terms)  
follow\_seen.remove(A)  
 return follow  
  
def closure(I):  
 J = I  
 while True:  
item\_len = len(J) + sum(len(v) for v in J.itervalues())  
 for heads in J.keys():  
 for prods in J[heads]:  
dot\_pos = prods.index('.') #checks if final item or not  
 if dot\_pos + 1 <len(prods):  
prod\_after\_dot = prods[dot\_pos + 1]  
 if prod\_after\_dot in nonterminals:  
 for prod in G[prod\_after\_dot]:   
 item = ["."] + prod  
 if prod\_after\_dot not in J.keys():  
 J[prod\_after\_dot] = [item]  
elif item not in J[prod\_after\_dot]:  
 J[prod\_after\_dot].append(item)  
 if item\_len == len(J) + sum(len(v) for v in J.itervalues()):  
 return J  
  
def GOTO(I, X):  
goto = {}  
 for heads in I.keys():  
 for prods in I[heads]:  
 for i in range(len(prods) - 1):  
 if "." == prods[i] and X == prods[i + 1]:  
temp\_prods = prods[:]  
temp\_prods[i], temp\_prods[i + 1] = temp\_prods[i + 1], temp\_prods[i]  
prod\_closure = closure({heads: [temp\_prods]})  
 for keys in prod\_closure:  
 if keys not in goto.keys():  
goto[keys] = prod\_closure[keys]  
elifprod\_closure[keys] not in goto[keys]:  
 for prod in prod\_closure[keys]:  
goto[keys].append(prod)  
 return goto  
  
def items():  
 global C  
i = 1  
 C = {'I0': closure({start: [['.'] + G[start][0]]})}  
 while True:  
item\_len = len(C) + sum(len(v) for v in C.itervalues())  
 for I in C.keys():  
 for X in symbols:  
 if GOTO(C[I], X) and GOTO(C[I], X) not in C.values():  
 C['I' + str(i)] = GOTO(C[I], X)  
i += 1  
 if item\_len == len(C) + sum(len(v) for v in C.itervalues()):  
 return  
  
  
def ACTION(i, a):  
 global error  
 for heads in C['I' + str(i)]:  
 for prods in C['I' + str(i)][heads]:  
 for j in range(len(prods) - 1):  
 if prods[j] == '.' and prods[j + 1] == a:  
for k in range(len(C)):  
 if GOTO(C['I' + str(i)], a) == C['I' + str(k)]:  
 if a in terminals:  
 if "r" in parse\_table[i][terminals.index(a)]:  
 if error!=1:  
 print "ERROR: Shift-Reduce Conflict at State " + str(i) + ", Symbol \'" + str(terminals.index(a))+"\'"  
 error=1  
 if "s"+str(k) not in parse\_table[i][terminals.index(a)]:  
parse\_table[i][terminals.index(a)] = parse\_table[i][terminals.index(a)]+ "/s" + str(k)  
 return parse\_table[i][terminals.index(a)]  
 else:  
parse\_table[i][terminals.index(a)] = "s" + str(k)  
 else:  
parse\_table[i][len(terminals) + nonterminals.index(a)] = str(k)  
 return "s" + str(k)  
 for heads in C['I' + str(i)]:  
 if heads != start:  
 for prods in C['I' + str(i)][heads]:  
 if prods[-1] == '.': #final item   
 k = 0  
 for head in G.keys():  
 for Gprods in G[head]:  
 if head == heads and (Gprods == prods[:-1] ) and (a in terminals or a == '$'):  
 for terms in FOLLOW(heads):  
 if terms == '$':  
 index = len(terminals)  
 else:  
 index = terminals.index(terms)  
 if "s" in parse\_table[i][index]:  
 if error!=1:  
 print "ERROR: Shift-Reduce Conflict at State " + str(i) + ", Symbol \'" + str(terms)+"\'"  
 error=1  
 if "r"+str(k) not in parse\_table[i][index]:  
parse\_table[i][index] = parse\_table[i][index]+ "/r" + str(k)  
 return parse\_table[i][index]  
elifparse\_table[i][index] and parse\_table[i][index] != "r" + str(k):  
 if error!=1:  
 print "ERROR: Reduce-Reduce Conflict at State " + str(i) + ", Symbol \'" + str(terms)+"\'"  
 error=1  
 if "r"+str(k) not in parse\_table[i][index]:  
parse\_table[i][index] = parse\_table[i][index]+ "/r" + str(k)  
 return parse\_table[i][index]   
 else:  
parse\_table[i][index] = "r" + str(k)  
 return "r" + str(k)  
 k += 1  
 if start in C['I' + str(i)] and G[start][0] + ['.'] in C['I' + str(i)][start]:  
parse\_table[i][len(terminals)] = "acc"  
 return "acc"  
 return ""  
  
def print\_info():  
 print "GRAMMAR:"  
 for head in G.keys():  
 if head == start:  
 continue  
 print "{:>{width}} ->".format(head, width=len(max(G.keys(), key=len))),  
num\_prods = 0  
 for prods in G[head]:  
 if num\_prods> 0:  
 print "|",  
 for prod in prods:  
 print prod,  
num\_prods += 1  
 print  
print "\nAUGMENTED GRAMMAR:"  
i = 0  
 for head in G.keys():  
 for prods in G[head]:  
 print "{:>{width}}:".format(str(i), width=len(str(sum(len(v) for v in G.itervalues()) - 1))),  
 print "{:>{width}} ->".format(head, width=len(max(G.keys(), key=len))),  
 for prod in prods:  
 print prod,  
 print  
i += 1  
 print "\nTERMINALS :", terminals  
 print "NONTERMINALS:", nonterminals  
 print "SYMBOLS :", symbols  
 print "\nFIRST:"  
 for head in G:  
 print "{:>{width}} =".format(head, width=len(max(G.keys(), key=len))),  
 print "{",  
num\_terms = 0  
 for terms in FIRST(head):  
 if num\_terms> 0:  
 print ", ",  
 print terms,  
num\_terms += 1  
 print "}"  
  
 print "\nFOLLOW:"  
 for head in G:  
 print "{:>{width}} =".format(head, width=len(max(G.keys(), key=len))),  
 print "{",  
num\_terms = 0  
 for terms in FOLLOW(head):  
 if num\_terms> 0:  
 print ", ",  
 print terms,  
num\_terms += 1  
 print "}"  
  
 print "\nITEMS:"  
 for i in range(len(C)):  
 print 'I' + str(i) + ':'  
 for keys in C['I' + str(i)]:  
 for prods in C['I' + str(i)][keys]:  
 print "{:>{width}} ->".format(keys, width=len(max(G.keys(), key=len))),  
 for prod in prods:  
 print prod,  
 print  
print  
  
 for i in range(len(parse\_table)): #len gives number of states  
 for j in symbols:  
 ACTION(i, j)  
  
 print "PARSING TABLE:"  
 print "+" + "--------+" \* (len(terminals) + len(nonterminals) + 1)  
 print "|{:^8}|".format('STATE'),  
 for terms in terminals:  
 print "{:^7}|".format(terms),  
 print "{:^7}|".format("$"),  
 for nonterms in nonterminals:  
 if nonterms == start:  
 continue  
 print "{:^7}|".format(nonterms),  
 print "\n+" + "--------+" \* (len(terminals) + len(nonterminals) + 1)  
 for i in range(len(parse\_table)):  
 print "|{:^8}|".format(i),  
 for j in range(len(parse\_table[i]) - 1):  
 print "{:^7}|".format(parse\_table[i][j]),  
 print  
print "+" + "--------+" \* (len(terminals) + len(nonterminals) + 1)  
  
def process\_input():  
get\_input = raw\_input("\nEnter Input: ")  
to\_parse = " ".join((get\_input + " $").split()).split(" ")  
 pointer = 0  
 stack = ['0']  
  
 print "\n+--------+----------------------------+----------------------------+----------------------------+"  
 print "|{:^8}|{:^28}|{:^28}|{:^28}|".format("STEP", "STACK", "INPUT", "ACTION")  
 print "+--------+----------------------------+----------------------------+----------------------------+"  
  
 step = 1  
 while True:  
curr\_symbol = to\_parse[pointer]  
top\_stack = int(stack[-1])  
stack\_content = ""  
input\_content = ""  
  
 print "|{:^8}|".format(step),  
 for i in stack:  
stack\_content += i  
 print "{:27}|".format(stack\_content),  
i = pointer  
 while i<len(to\_parse):  
input\_content += to\_parse[i]  
i += 1  
 print "{:>26} | ".format(input\_content),  
  
 step += 1  
get\_action = ACTION(top\_stack, curr\_symbol)  
 if "/" in get\_action:  
 print "{:^26}|".format(get\_action+". So conflict")  
 break  
 if "s" in get\_action:  
 print "{:^26}|".format(get\_action)  
stack.append(curr\_symbol)  
stack.append(get\_action[1:])  
 pointer += 1  
elif "r" in get\_action:  
 print "{:^26}|".format(get\_action)  
i = 0  
 for head in G.keys():  
 for prods in G[head]:  
 if i == int(get\_action[1:]):  
 for j in range(2 \* len(prods)):  
stack.pop()  
 state = stack[-1]  
stack.append(head)  
stack.append(parse\_table[int(state)][len(terminals) + nonterminals.index(head)])  
i += 1  
elifget\_action == "acc":  
 print "{:^26}|".format("ACCEPTED")  
 break  
 else:  
 print "ERROR: Unrecognized symbol", curr\_symbol, "|"  
 break  
 print "+--------+----------------------------+----------------------------+----------------------------+"  
  
def main():   
parse\_grammar()  
 items()  
 global parse\_table  
parse\_table = [["" for c in range(len(terminals) + len(nonterminals) + 1)] for r in range(len(C))]  
print\_info()  
process\_input()  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**Output:**



**CONCLUSION**

While doing the 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 top down 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 else a syntax error is displayed.