**NMAM INSTITUTE OF TECHNOLOGY**

(A unit of Nitte Education Trust)

Nitte - 574 110, Karkala taluk, Udupi Dist., Karnataka

**Department of Computer Science and Engineering.**

# MINI PROJECT REPORT ON COMPILER DESIGN

**SUBMITTED BY**

| | |
|---|---|
| ADARSH SHETTY | ASHWIN THOMAS |
| 4NM17CS009 | 4NM17CS036 |
| 6 Sem Sec A | 6 Sem Sec A |
| Dept of CSE | Dept of CSE |

**UNDER THE GUIDANCE OF**

Dr. JYOTHI SHETTY

Dept of CSE

NMAMIT , NITTE

**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

Certified that the project work carried out by Adarsh Shetty, USN 4NM17CS009 and Ashwin Thomas, USN 4NM17CS036 , bonafide students of NMAM Institute of Technology, Nitte in fulfillment for the Compiler Design lab in Computer Science and Engineering during the academic year 2019 – 2020.

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 necessary facilities.

We express our deep sense of gratitude to our guide **Dr. Jyothi Shetty**, Department of Computer Science and Engineering, for her inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.

We sincerely thank **Dr. Uday Kumar Reddy**, Head of  Department of Computer Science and Engineering, NMAM 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.

Adarsh Shetty (4NM17CS009)　　　　Ashwin Thomas (4NM17CS036)

# ABSTRACT

This project implements first two phases of a compiler which are lexical analysis and syntax analysis, for a given piece of code. First is the lexical analysis. This phase is also known as the tokenizer. It takes the modified source code from the language preprocessors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespaces or comments in the source code. A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyser as an instance of that token. The lexical syntax is usually a regular language, with the grammar rules consisting of regular expressions; they define the set of possible character sequences of a token(lexeme). Two common lexical categories are whitespace and comments. These are now handled by lexical analyser. In this phase errors are not detected, except if any characters are used which are not in the character set.

The next phase is the syntax analysis phase. It is also called as parser. It checks the syntactical structure of the given input i.e. whether the given input is in the correct syntax or not. It does so by building a data structure, called a parse tree or syntax tree. The parse tree is constructed by using the predefined grammar of the language and the input string. If the given input string can be produced with the help of syntax tree, the input string is found to be in the correct syntax. If not, error is reported by Syntax analyser. The parser attempts token construct syntax tree from this grammar for the given input string. Its uses the given production rules and applies those as needed to generate the string. 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.

## CONTENTS

# 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 time-critical programs are still written partly in machine language.
- A good compiler will, however, be able to get very close to the speed of hand-written 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.

### 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.

## PROBLEM STATEMENT

To design lexical analyser and syntax analyser for the following hypothetical language

```
int main()
{
/*comment statement*/

var = 10.5 ;
var = 10 ;
var = arithmetic expression;
/*comment statement */
 print (var) ;
}
```

## LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any hitespace 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. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

• Efficiency: A lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non-linear factor involved which may make a separated system smaller

than a combined system.
• Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.
• Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

**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) 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.

## <u>SYNTAX ANALYSIS</u>
Syntax analysis or parsing is the second phase of a compiler. 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.

The syntax analyser 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.

Syntax Error Handling:

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

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.

- 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.

- 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:

- It should the presence of errors clearly and accurately.
- 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:
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

**<u>CONTEXT FREE GRAMMARS</u>**:
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:

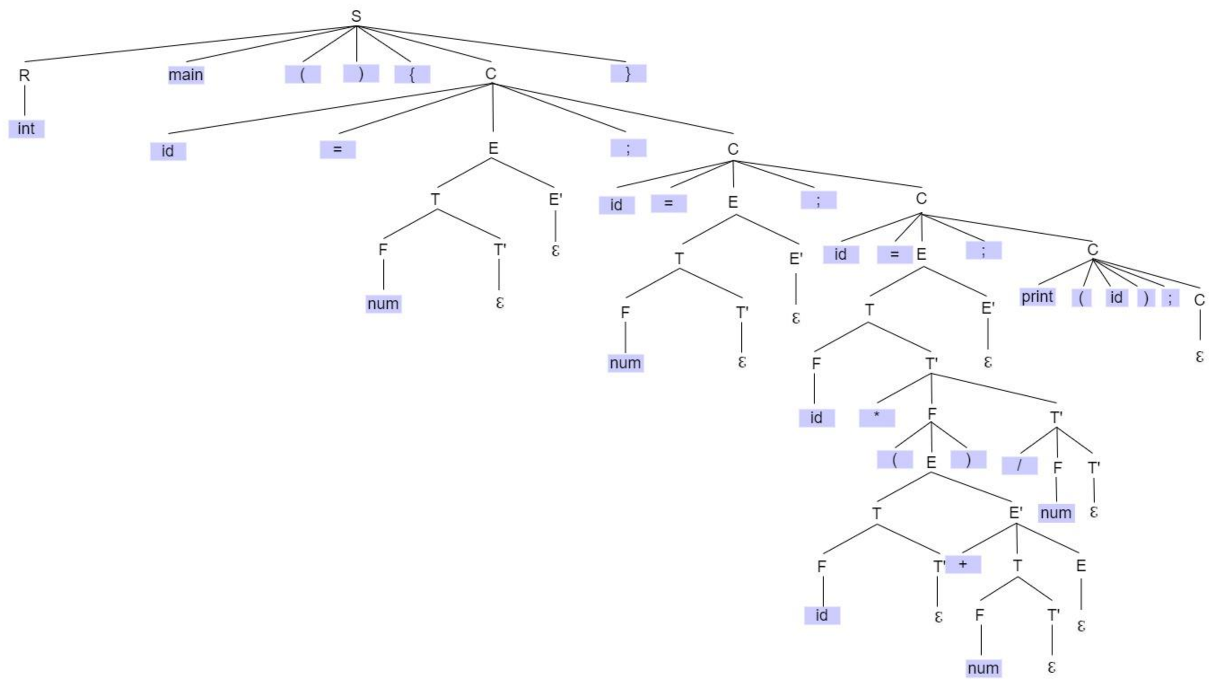
- A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.

- 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 non-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.

Parse tree for the above problem statement:



**<u>TYPES OF PARSING</u>**:

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 : top-down parsing and bottom-up 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. Whereas bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

*NOTE: The type of parser we have used is top down parser using predictive parsing algorithm.*

## **<u>DESIGNING OF SYNTAX ANALYSER</u>**

LL(1) GRAMMARS AND LANGUAGES. A context-free grammar G = (V T , V N , S, P) whose parsing table has no multiple entries is said to be LL(1). In the name LL(1),

- the first L stands for scanning the input from left to right,
- the second L stands for producing a leftmost derivation,
- and the 1 stands for using one input symbol of lookahead at each step to make parsing action decision.

A language is said to be LL(1) if it can be generated by a LL(1) grmmar. It can be shown that LL(1) grammars are

- not ambiguous and
- not left-recursive.

## LL(1) grammar for above problem statement :

S -> R main ( ) { C }

R -> int

R -> void

C -> id = E ; C

C -> print ( id ) ; C

C -> Ɛ

E -> TE`

E` -> +TE`

E` -> -TE`

E` -> Ɛ

T -> FT`

T` -> *FT`

T` -> /FT`

T` -> Ɛ

F -> (E)

F -> {E}

F -> [E]

F -> id

F-> num

## 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 $T[A, t] = \alpha$ with some production rule.

## First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example, $\alpha \rightarrow t\ \beta$ That is, $\alpha$ derives t (terminal) in the very first position. So, $t \in \text{FIRST}(\alpha)$.

Algorithm for Calculating First Set

Look at the definition of FIRST($\alpha$) set:

if $\alpha$ is a terminal, then FIRST($\alpha$) = { $\alpha$ }.

if $\alpha$ is a non-terminal and $\alpha \rightarrow \mathcal{E}$ is a production, then FIRST($\alpha$) = { $\mathcal{E}$ }.

if $\alpha$ is a non-terminal and $\alpha \rightarrow \gamma1\ \gamma2\ \gamma3 \ldots \gamma n$ and any FIRST($\gamma$) contains t, then t is in FIRST($\alpha$).

First set can be seen as: $\text{FIRST}(\alpha) = \{ t \mid \alpha \rightarrow^* t\ \beta \} \cup \{ \mathcal{E} \mid \alpha \rightarrow^* \varepsilon \}$

## Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal $\alpha$ 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 $\alpha$ is a start symbol, then FOLLOW() = \$

if $\alpha$ is a non-terminal and has a production $\alpha \rightarrow AB$, 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 problem statement :

FIRST(S)={int , void}

FIRST(R)={int , void}

FIRST(C)={id , print , Ɛ}

FIRST(E)={ ( , { , [ , id , num}

FIRST(E`)={ + , - , Ɛ }

FIRST(T)={ ( , { , [ , id , num }

FIRST(T`)={* , / , Ɛ }

FIRST(F)={ ( , { , [ , id , num }

FOLLOW(S)={ $ }

FOLLOW(R)={ main }

FOLLOW(C)={ } }

FOLLOW(E)={ ; , ) , } , ] }

FOLLOW(E`)={ ; , ) , } , ] }

FOLLOW(T)={ + , - , ; , ) , } , ] }

FOLLOW(T`)={+ , - , ; , ) , } , ] }

FOLLOW(F)={ * , / , ; , ) , } , ] }

## PARSING TABLE

<u>Algorithm to construct predictive parsing table</u>

Input : Grammar G

Output : Parsing table M

<u>Method</u>

1. For each production A → α of a grammar , do steps 2 and 3
2. For each terminal a in FIRST(α), add A → α to M[A,a]
3. If ε is in FIRST(α), add A → α to M[A,b] for each symbol b in FOLLOW(A)
4. Consider each undefined entry as error

## Parsing Table for the problem statement

| | **int** | **void** | **main** | **print** | **(** | **{** | **[** | **]** |
|---|---|---|---|---|---|---|---|---|
| **S** | S ->R main ( ) { C } | S ->R main ( ) { C } | | | | | | |
| **R** | R ->int | R ->void | | | | | | |
| **C** | | | | C ->print ( id ) ; C | | | | |
| **E** | | | | | E ->TE` | E ->TE` | E ->TE` | |
| **E`** | | | | | | | | E` ->Ɛ |
| **T** | | | | | T ->FT` | T ->FT` | T ->FT` | |
| **T`** | | | | | | | | T` ->Ɛ |
| **F** | | | | | F ->(E) | F ->{E} | F ->[E] | |

| | } | ) | + | - | * | / | ; | id | num | $ |
|---|---|---|---|---|---|---|---|---|---|---|
| **S** | | | | | | | | | | |
| **R** | | | | | | | | | | |
| **C** | C ->Ɛ | | | | | | | C ->id = E ; C | | |
| **E** | | | | | | | | E ->TE` | E ->TE` | |
| **E`** | E` ->Ɛ | E` ->Ɛ | E`->+T E` | E`->-T E` | | | E` ->Ɛ | | | |
| **T** | | | | | | | | T ->FT` | T ->FT` | |
| **T`** | T` ->Ɛ | T` ->Ɛ | T` ->Ɛ | T` ->Ɛ | T`->*F T` | T`->/F T` | T` ->Ɛ | | | |
| **F** | | | | | | | | F->id | F->num | |

## Predictive parsing algorithm to generate parse tree

<u>LL(1) Parsing Algorithm</u>

```
Set ip to point  to the first symbol of input string W$
Repeat
    Begin
        Let X be the top stack symbol and a be the symbol pointed by ip
        If ( X is terminal ) then
            If ( X == a ) then
                Pop X from the stack
                Advance ip
            Else
                Error( )
            Else  /* X is a non terminal */
                  If ( M[ X, a ] = X $\rightarrow Y_1\ Y_2 \dots Y_k$) then
                        Pop X from the stack
                        Push $Y_k$ , $Y_{k-1} \dots Y_1$ on to stack , with Y1 on top
                        Output  production  X  $\rightarrow Y_1\ Y_2 \dots Y_k$
                  Else
                        Error( )
      End
Until( Stack is Empty )
```

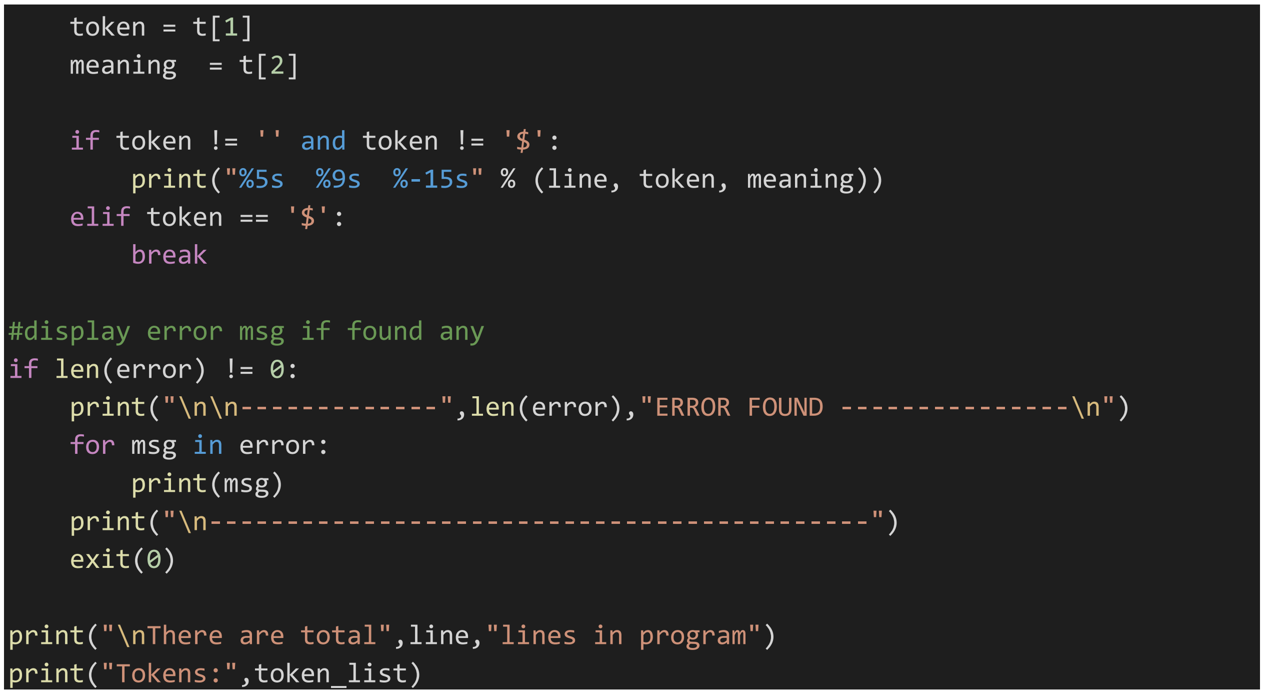
## <u>IMPLEMENTATION OF LEXICAL AND SYNTAX ANALYSER</u>

Lexical Analyser code for the problem statement in python:

```python
import re
import io

keywords=["int","void","main","print"]

operators = { '=': 'Assignment Operator','+': 'Additon Operator', '-' : 'Substraction Operator', '/' : 'Division Operator', '*': 'Multiplication Operator'}
optr_keys = operators.keys()

symbols = {';':'semi_colon','{' : 'left_brace', '}':'right_brace', '(':'left_parenthesis',')':'right_parenthesis' ,'[':'left_sqbracket',']':'right_sqbracket'}
symbol_keys = symbols.keys()

the_ch = " "
the_line = 1
token_list = []
```

```
error = []


#get the next character from the input file
def next_ch():

    global the_ch, the_line
    the_ch = input_file.read(1)
    if the_ch == '\n':
        the_line += 1
    return the_ch


#handle identifiers and numbers
def identifier_or_number(line_no):

    text = ""
    while the_ch.isalnum() or the_ch == '_' or the_ch =='.':
        text += the_ch
        next_ch()

    if len(text) == 0:
        error_msg = "Unrecognized character "+the_ch+" found in line : "+str(line
_no)
        error.append(error_msg)
        next_ch()
        return '' , '' , ''

    elif text in keywords:
        token_list.append(text)
        return line_no, text, "Keyword"

    elif text in re.findall('[_a-zA-Z][_a-zA-Z0-9]*',text):
        token_list.append('id')
        return line_no , text , 'Identifier'

    elif text in re.findall('[0-9]+[.]?[0-9]*',text):
        token_list.append('num')
        return line_no , text , 'Number'

    elif text not in re.findall('[_a-zA-Z ][_a-zA-Z0-9 ]*',text):
        error_msg=text+" is an invalid identifier found in line : "+str(line_no)
        error.append(error_msg)
        return '','',''
```

```python
#return the next token type
def getToken():

    while the_ch.isspace():
        next_ch()

    line_no = the_line

    if len(the_ch) == 0:
        token_list.append('$')
        return line_no, '$' , 'End_of_input'

    elif the_ch in symbol_keys:
        token = the_ch
        token_list.append(token)
        sym = symbols[token]
        next_ch()
        return line_no, token , sym

    elif the_ch in optr_keys:
        token = the_ch
        token_list.append(token)
        opr = operators[token]
        next_ch()
        return line_no, token , opr

    else:
        return identifier_or_number(line_no)


#opening input file
f = open("input.txt", "r")
i = f.read()

program = re.sub('//.*?\n|/\*.*?\*/', '', i, flags=re.S) #removes all comment lin
es from input file
input_file = io.StringIO(program) #converting string to file object

print("\nOutput of Lexical Analyser\n--------------------------\n")
print("%5s %7s %9s" % ("Line No.", "Token","Meaning"))
print("----------------------------")

while True:
    t = getToken()
    line  = t[0]
```



```python
    token = t[1]
    meaning  = t[2]

    if token != '' and token != '$':
        print("%5s  %9s  %-15s" % (line, token, meaning))
    elif token == '$':
        break

#display error msg if found any
if len(error) != 0:
    print("\n\n-------------",len(error),"ERROR FOUND ---------------\n")
    for msg in error:
        print(msg)
    print("\n-------------------------------------------")
    exit(0)

print("\nThere are total",line,"lines in program")
print("Tokens:",token_list)
```

Explanation for the above code:

- The list of keywords , operators and symbols used in problem statement have been declared.
- The input file containing program is read using file operation methods.
- Then using ‘re’ module of python all the comment lines are removed ( i.e. lines starting with // or lines with /*……*/ are removed)
- Then the getToken( ) method , gets token and everytime it compares token with each keyword in the list and if it matches any of them,then it is displayed as a keyword.
- If it matches with any of the operators or symbols , it is displayed as operator or symbol respectively.
- If it matches with valid number or identifier , it is displayed as number or identifier respectively.
- If it does not matches any of these , error message is printed.
- All the valid tokens are appended to input_token list which is given as input

to Syntax Analyser program.

Syntax Analyser code for the problem statement in python:

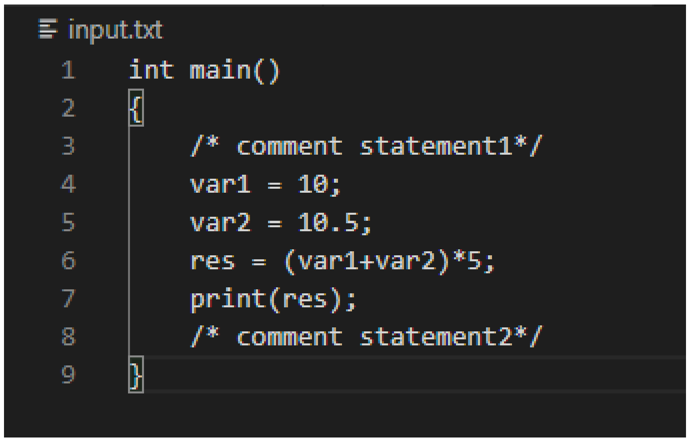
```
from lexical_analyser import * #import lexical analyser code

grammar = [
    ["S","R","main","(",")","{","C","}","$"],
    ["R","int","$"],
    ["R","void","$"],
    ["C","id","=","E",";","C","$"],
    ["C","print","(","id",")",";","C","$"],
    ["C","epsolon","$"],
    ["E","T","E'","$"],
    ["E'","+","T","E'","$"],
    ["E'","-","T","E'","$"],
    ["E'","epsolon","$"],
    ["T","F","T'","$"],
    ["T'","*","F","T'","$"],
    ["T'","/","F","T'","$"],
    ["T'","epsolon","$"],
    ["F","(","E",")","$"],
    ["F","{","E","}","$"],
    ["F","[","E","]","$"],
    ["F","id","$"],
    ["F","num","$"],
]

parsing_table = [
    ["S","int","0"],
    ["S","void","0"],
    ["R","int","1"],
    ["R","void","2"],
    ["C","print","4"],
    ["C","}","5"],
    ["C","id","3"],
    ["E","(","6"],
    ["E","{","6"],
    ["E","[","6"],
    ["E","id","6"],
    ["E","num","6"],
    ["E'","]","9"],
    ["E'","}","9"],
    ["E'",")","9"],
    ["E'","+","7"],
    ["E'","-","8"],
    ["E'",";","9"],
    ["T","(","10"],
    ["T","{","10"],
    ["T","[","10"],
    ["T","id","10"],
    ["T","num","10"],
    ["T'","]","13"],
    ["T'","}","13"],
    ["T'",")","13"],
    ["T'","+","13"],
    ["T'","-","13"],
    ["T'",";","13"],
    ["T'","*","11"],
    ["T'","/","12"],
    ["F","(","14"],
    ["F","{","15"],
    ["F","[","16"],
    ["F","id","17"],
    ["F","num","18"],
]

input_tokens = token_list #token_list is a set of all tokens from lexical analyse
r
ip = 0
stack = ['S'] #Starting symbol S at top of stack
top = 0

def update_stack(x,s):
    global stack
    buffer = []
    print(grammar[x][0],"->",end="")
    y = 1

    while grammar[x][y] != '$':
        print(grammar[x][y],end="")

        if grammar[x][y] == 'epsolon':
            stack.pop()
            r = len(stack) - 1
            print("\n")
            return r

        buffer.append(grammar[x][y])
```

```python
        y = y + 1

    stack.pop()
    for i in reversed(buffer):
        stack.append(i)
    r = len(stack) - 1
    print("\n")

    return r


print("\n\nOutput of Syntax Analyser(Parse Tree)\n-------------------------")

while input_tokens[ip] != '$':

    if top>-1:
        s = stack[top]
    t = input_tokens[ip]

    flag = False

    for i in range(len(parsing_table)):

        if parsing_table[i][0] == s and parsing_table[i][1] == t:
            x = int(parsing_table[i][2])
            top = update_stack(x,s)
            flag = True
            break

        elif s == t:
            #print(t,"matched")
            ip = ip + 1
            stack.pop()
            top = top - 1
            flag = True
            break

    if flag == False:
        print("Syntax error found\nINVALID PROGRAM ")
        exit(1)

if len(stack) == 0: #if is stack empty then valid
    print("******* VALID PROGRAM *********")
else:
    print("Syntax error found INVALID PROGRAM ")
```

The grammar and parsing table is stored in 2D list grammar and parsing_table respectively. The output of  Lexical Analyser i.e. the set of tokens is stored in input_tokens list. The productions are in stack list with start symbol S as top of stack. Then the above mentioned predictive parsing algorithm is applied and ouput(parse tree) is printed. If no syntactical error is encountered it prints Valid Program , if not prints Invalid Program.

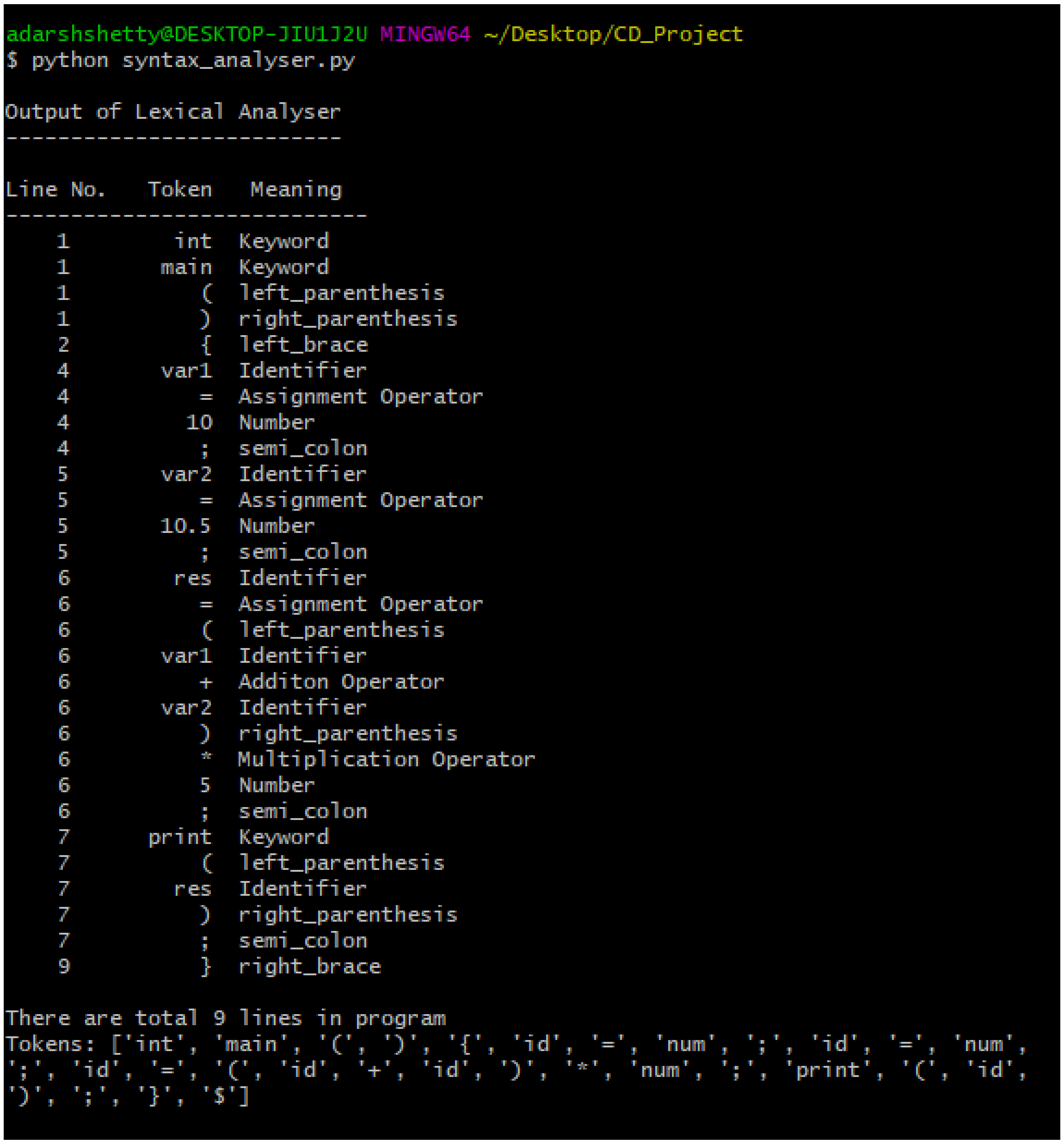
## RESULTS

1. Input : Valid input program

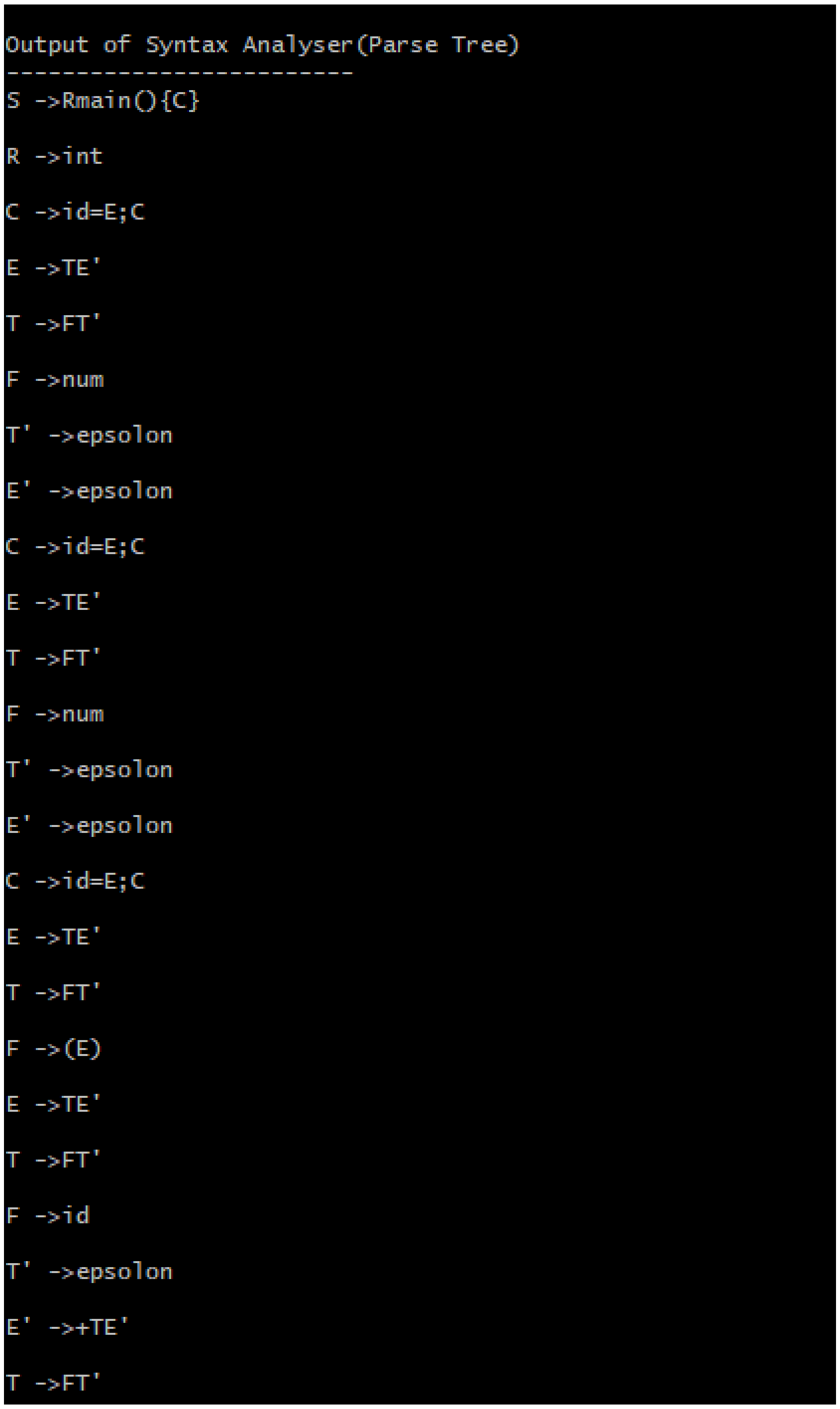


```
input.txt
int main()
{
    /* comment statement1*/
    var1 = 10;
    var2 = 10.5;
    res = (var1+var2)*5;
    print(res);
    /* comment statement2*/
}
```

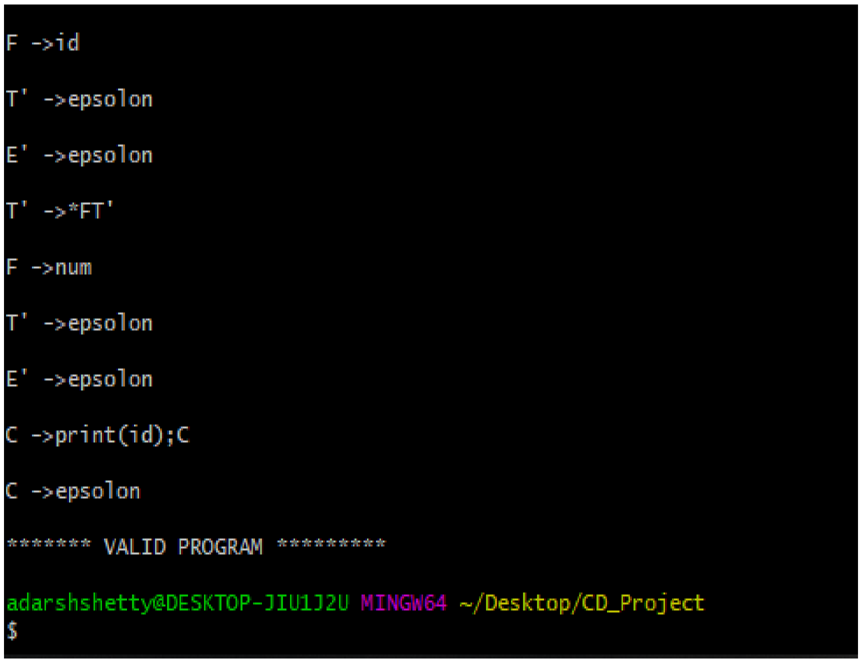
Output:



```
adarshshetty@DESKTOP-JIU1J2U MINGW64 ~/Desktop/CD_Project
$ python syntax_analyser.py

Output of Lexical Analyser
--------------------------

Line No.   Token   Meaning
----------------------------
    1        int  Keyword
    1       main  Keyword
    1          (  left_parenthesis
    1          )  right_parenthesis
    2          {  left_brace
    4       var1  Identifier
    4          =  Assignment Operator
    4         10  Number
    4          ;  semi_colon
    5       var2  Identifier
    5          =  Assignment Operator
    5       10.5  Number
    5          ;  semi_colon
    6        res  Identifier
    6          =  Assignment Operator
    6          (  left_parenthesis
    6       var1  Identifier
    6          +  Additon Operator
    6       var2  Identifier
    6          )  right_parenthesis
    6          *  Multiplication Operator
    6          5  Number
    6          ;  semi_colon
    7      print  Keyword
    7          (  left_parenthesis
    7        res  Identifier
    7          )  right_parenthesis
    7          ;  semi_colon
    9          }  right_brace

There are total 9 lines in program
Tokens: ['int', 'main', '(', ')', '{', 'id', '=', 'num', ';', 'id', '=', 'num',
';', 'id', '=', '(', 'id', '+', 'id', ')', '*', 'num', ';', 'print', '(', 'id',
')', ';', '}', '$']
```

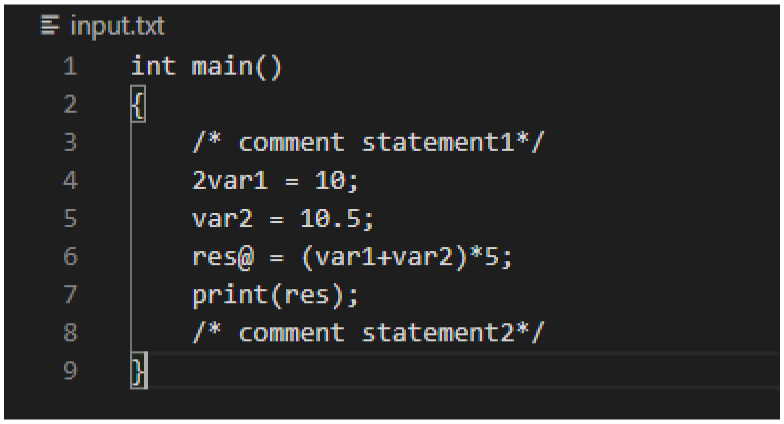


```
Output of Syntax Analyser(Parse Tree)
-------------------------
S ->Rmain(){C}

R ->int

C ->id=E;C

E ->TE'

T ->FT'

F ->num

T' ->epsolon

E' ->epsolon

C ->id=E;C

E ->TE'

T ->FT'

F ->num

T' ->epsolon

E' ->epsolon

C ->id=E;C

E ->TE'

T ->FT'

F ->(E)

E ->TE'

T ->FT'

F ->id

T' ->epsolon

E' ->+TE'

T ->FT'
```



```
F ->id

T' ->epsolon

E' ->epsolon

T' ->*FT'

F ->num

T' ->epsolon

E' ->epsolon

C ->print(id);C

C ->epsolon

******* VALID PROGRAM *********

adarshshetty@DESKTOP-JIU1J2U MINGW64 ~/Desktop/CD_Project
$
```

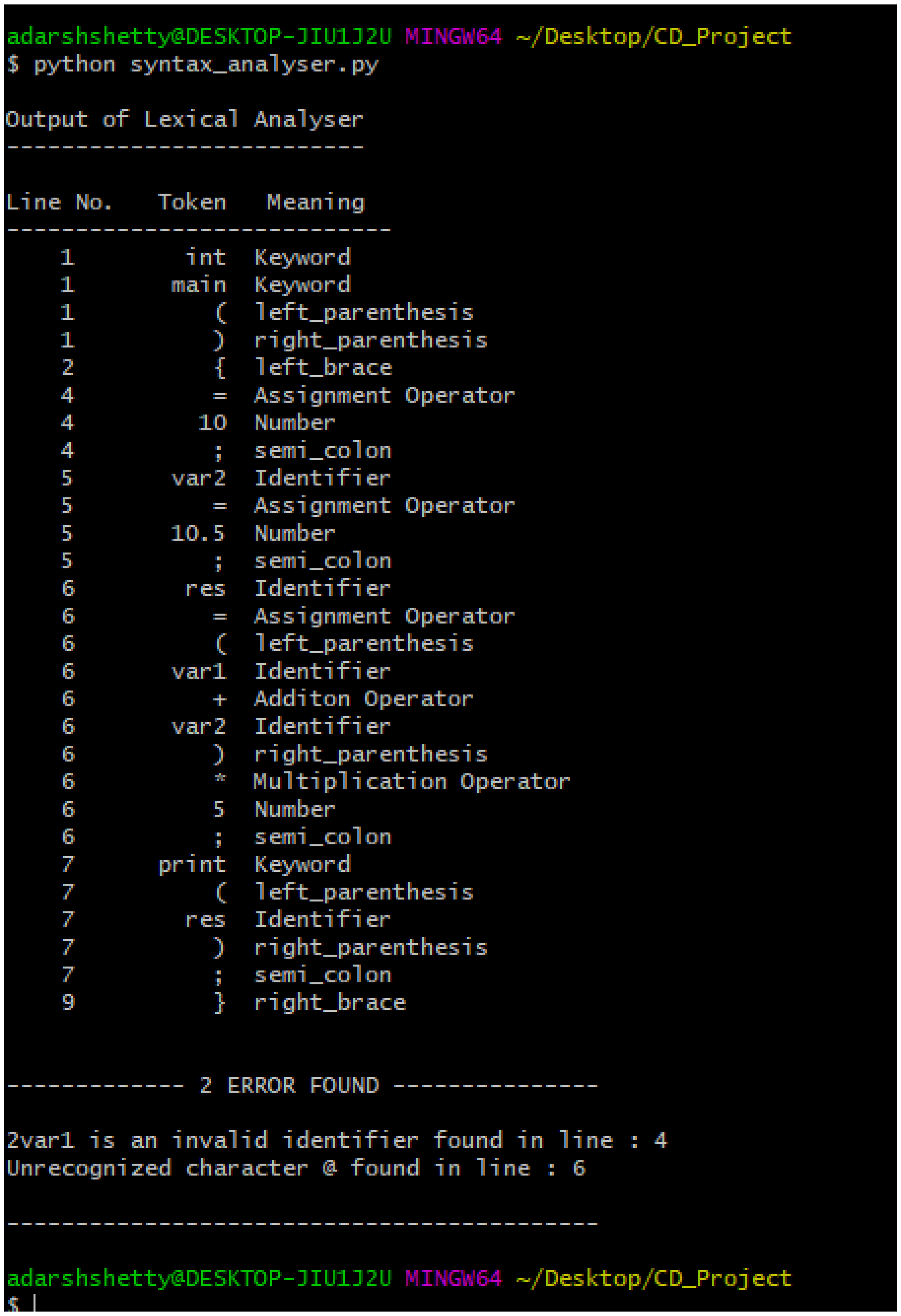
2. Input : Giving invalid tokens and special symbol as input



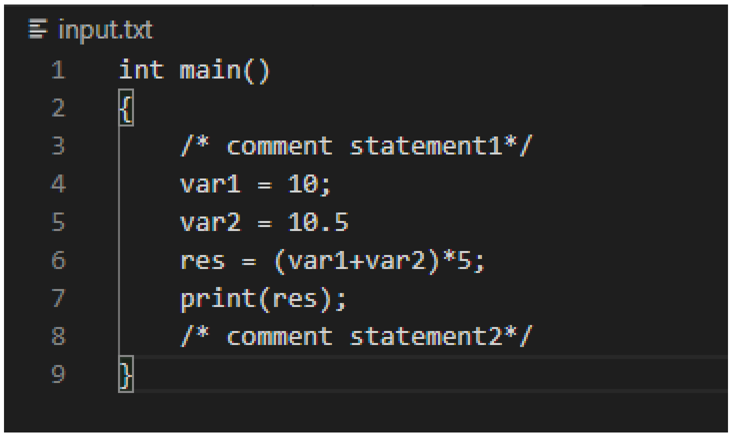
input.txt

```
int main()
{
    /* comment statement1*/
    2var1 = 10;
    var2 = 10.5;
    res@ = (var1+var2)*5;
    print(res);
    /* comment statement2*/
}
```

Output:

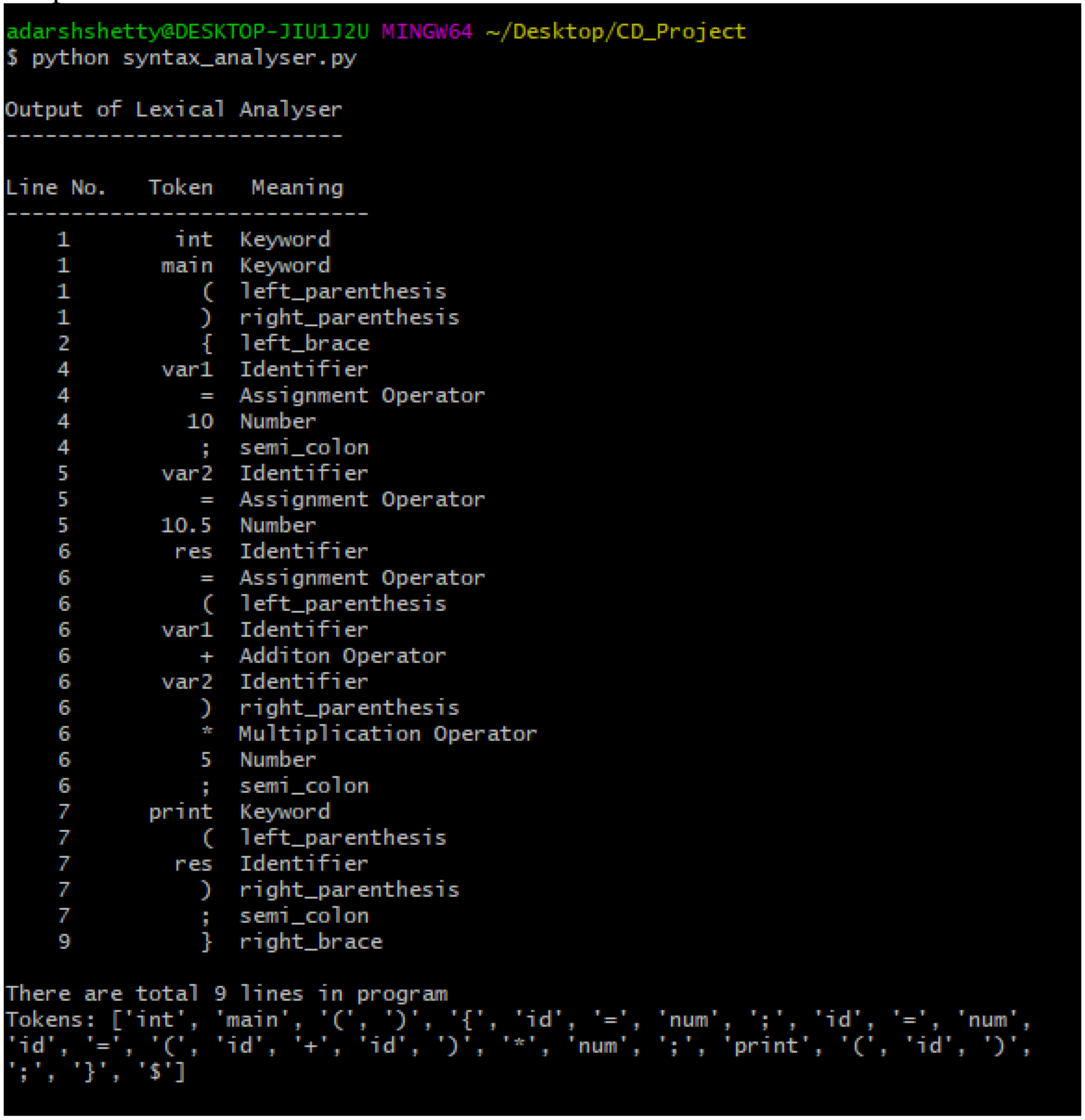

```
adarshshetty@DESKTOP-JIU1J2U MINGW64 ~/Desktop/CD_Project
$ python syntax_analyser.py

Output of Lexical Analyser
--------------------------

Line No.   Token   Meaning
----------------------------
    1        int  Keyword
    1       main  Keyword
    1          (  left_parenthesis
    1          )  right_parenthesis
    2          {  left_brace
    4          =  Assignment Operator
    4         10  Number
    4          ;  semi_colon
    5       var2  Identifier
    5          =  Assignment Operator
    5       10.5  Number
    5          ;  semi_colon
    6        res  Identifier
    6          =  Assignment Operator
    6          (  left_parenthesis
    6       var1  Identifier
    6          +  Additon Operator
    6       var2  Identifier
    6          )  right_parenthesis
    6          *  Multiplication Operator
    6          5  Number
    6          ;  semi_colon
    7      print  Keyword
    7          (  left_parenthesis
    7        res  Identifier
    7          )  right_parenthesis
    7          ;  semi_colon
    9          }  right_brace


------------- 2 ERROR FOUND ---------------

2var1 is an invalid identifier found in line : 4
Unrecognized character @ found in line : 6

-------------------------------------------

adarshshetty@DESKTOP-JIU1J2U MINGW64 ~/Desktop/CD_Project
$ |
```

3. Input : Program with syntax error

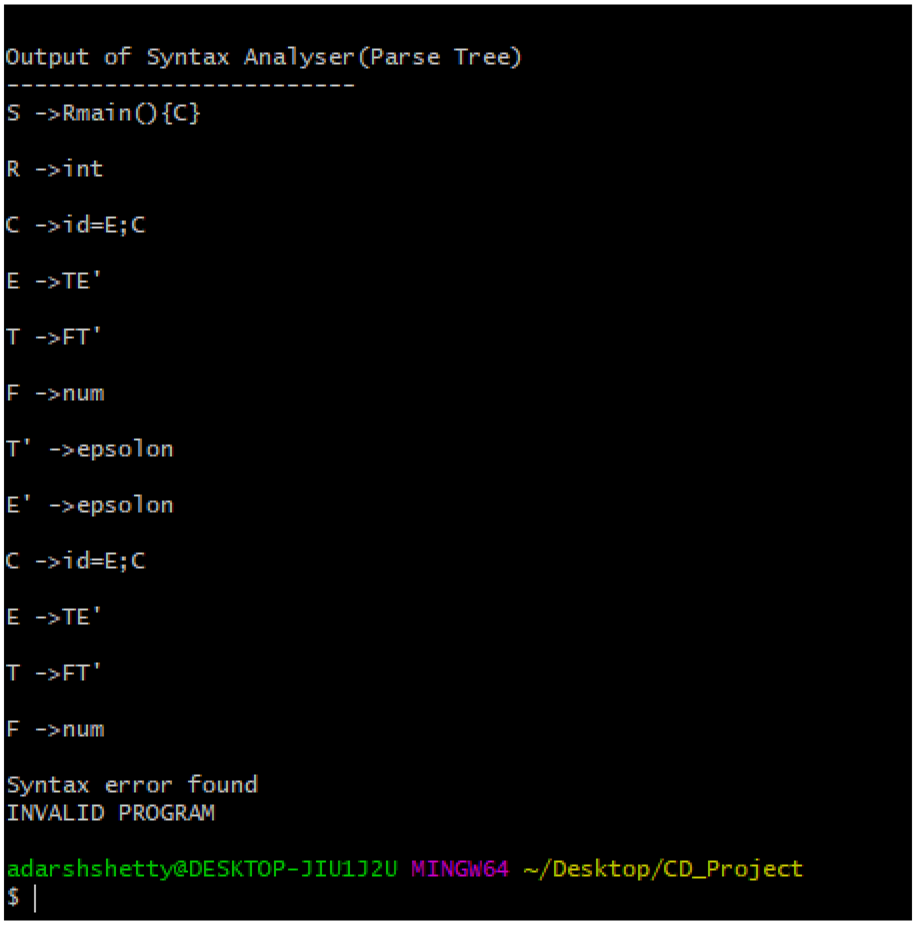


```
input.txt
int main()
{
    /* comment statement1*/
    var1 = 10;
    var2 = 10.5
    res = (var1+var2)*5;
    print(res);
    /* comment statement2*/
}
```

Output:



```
adarshshetty@DESKTOP-JIU1J2U MINGW64 ~/Desktop/CD_Project
$ python syntax_analyser.py

Output of Lexical Analyser
--------------------------

Line No.   Token   Meaning
----------------------------
    1        int  Keyword
    1       main  Keyword
    1          (  left_parenthesis
    1          )  right_parenthesis
    2          {  left_brace
    4       var1  Identifier
    4          =  Assignment Operator
    4         10  Number
    4          ;  semi_colon
    5       var2  Identifier
    5          =  Assignment Operator
    5       10.5  Number
    6        res  Identifier
    6          =  Assignment Operator
    6          (  left_parenthesis
    6       var1  Identifier
    6          +  Additon Operator
    6       var2  Identifier
    6          )  right_parenthesis
    6          *  Multiplication Operator
    6          5  Number
    6          ;  semi_colon
    7      print  Keyword
    7          (  left_parenthesis
    7        res  Identifier
    7          )  right_parenthesis
    7          ;  semi_colon
    9          }  right_brace

There are total 9 lines in program
Tokens: ['int', 'main', '(', ')', '{', 'id', '=', 'num', ';', 'id', '=', 'num',
'id', '=', '(', 'id', '+', 'id', ')', '*', 'num', ';', 'print', '(', 'id', ')',
';', '}', '$']
```



```
Output of Syntax Analyser(Parse Tree)
-------------------------
S ->Rmain(){C}

R ->int

C ->id=E;C

E ->TE'

T ->FT'

F ->num

T' ->epsolon

E' ->epsolon

C ->id=E;C

E ->TE'

T ->FT'

F ->num

Syntax error found
INVALID PROGRAM

adarshshetty@DESKTOP-JIU1J2U MINGW64 ~/Desktop/CD_Project
$ |
```

## <u>**CONCLUSION**</u>

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 top down parser, on giving an input list of tokens, it is parsed as per the grammar and parsing table given in the program. If the string was parsed completely then the leftmost derivation of the string is displayed in the output else a syntax error is displayed.