GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 1

Title of Experiment: Study of Lexical Analysis Tool: Flex

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

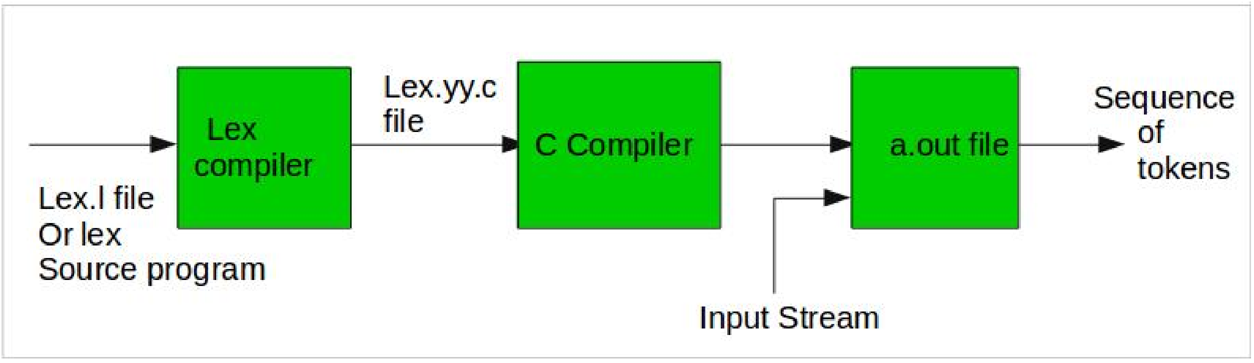
Signature of Subject Teacher

AIM: Study of Lexical Analysis Tool: Flex

Theory:

Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.
Bison produces parser from the input file provided by the user. The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

Given image describes how the Flex is used:



Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.
Step 2: The C complier compile lex.yy.c file into an executable file called a.out.
Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

Program Structure:
In the input file, there are 3 sections:
1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in **"%{ %}"** brackets. Anything written in this brackets is copied directly to the file **lex.yy.c**

Syntax:

```
%{
  // Definitions
%}
```

2. Rules Section: The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in **"%% %%"**.
Syntax:

```
%%
pattern  action
%%
```

Examples: Table below shows some of the pattern matches.

Pattern	It can match with
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘, ‘ or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{4}	exactly 4 a’s i.e, aaaa
a*	0 or more occurrences of a
a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x | y)z	wxz or wyz

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

```
%{
// Definitions
%}

%%
Rules
%%
```

How to run the program:
To run the program, it should be first saved with the extension .l or .lex. Run the below commands on terminal in order to run the program file.
Step 1: lex filename.l or lex filename.lex depending on the extension file is saved with
Step 2: gcc lex.yy.c
Step 3: ./a.out
Step 4: Provide the input to program in case it is required.

Conclusion: We have studied about FLEX which is a lexical analysis tool.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 2

Title of Experiment: Study of Syntax Analysis tool: YACC/BISON

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

AIM: Study of Syntax Analysis tool: YACC/BISON

Theory:

What is a Parser?

Parsing is the process of matching grammar symbols to elements in the input data, according to the rules of the grammar. The parser obtains a sequence of tokens from the lexical analyzer, and recognizes its structure in the form of a parse tree. The parse tree expresses the hierarchical structure of the input data, and is a mapping of grammar symbols to data elements. Tree nodes represent symbols of the grammar (non-terminals or terminals), and tree edges represent derivation steps.

YACC

YACC stands for **Yet Another Compiler Compiler**. YACC provides a tool to produce a parser for a given grammar. YACC is a program designed to compile a LALR (1) grammar. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar and the output is a C program.

These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

The output file "file.output" contains the parsing tables.

The file "file.tab.h" contains declarations.

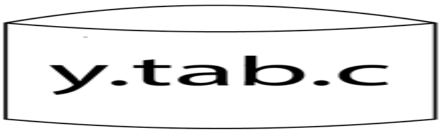
The parser called the yyparse ().

Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:

This file contains the desired grammar in YACC format.

It shows the YACC program.



It is the c source program created by YACC.

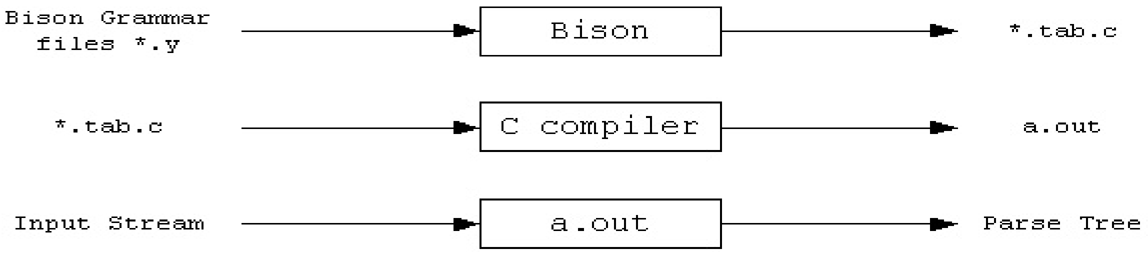
C Compiler

Executable file that will parse grammar given in gram.Y

BISON

BISON is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1)parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.Bison is upward compatible with Yacc: all properly-written Yacc grammars ought towork with Bison with no change.

Bison was written originally by Robert Corbett. Richard Stallman made it Yacc-compatible. Wilfred Hansen of Carnegie Mellon University added multi-character string literals and other features.



Steps to use Bison:

Write a lexical analyzer to process input and pass tokens to the parser (calc.lex).

Write the grammar specification for bison (calc.y), including grammar rules, yyparse() and yyerror().

Run Bison on the grammar to produce the parser. (Makefile)

Compile the code output by Bison, as well as any other source files.

Link the object files to produce the finished product.

Conclusion: We have studied about YACC & BISON parser generator.

GHARDA FOUNDATION’S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL
COMPUTER ENGINEERING DEPARTMENT
A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 3

Title of Experiment: Study of Lexical Analysis Tool: Flex

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

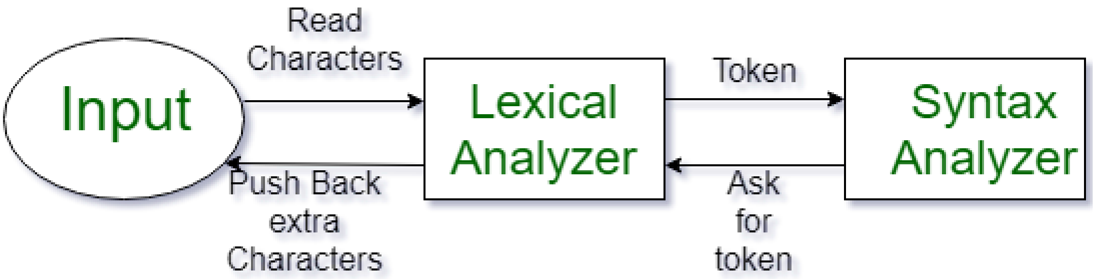
AIM : Implementation of Lexical Analyser.

Theory:

Lexical Analyser:

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis



What is a token?
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Keywords; Examples-for, while, if etc.

Identifier; Examples-Variable name, function name, etc.

Operators; Examples '+', '++', '-' etc.

Separators; Examples ',' ';' etc.

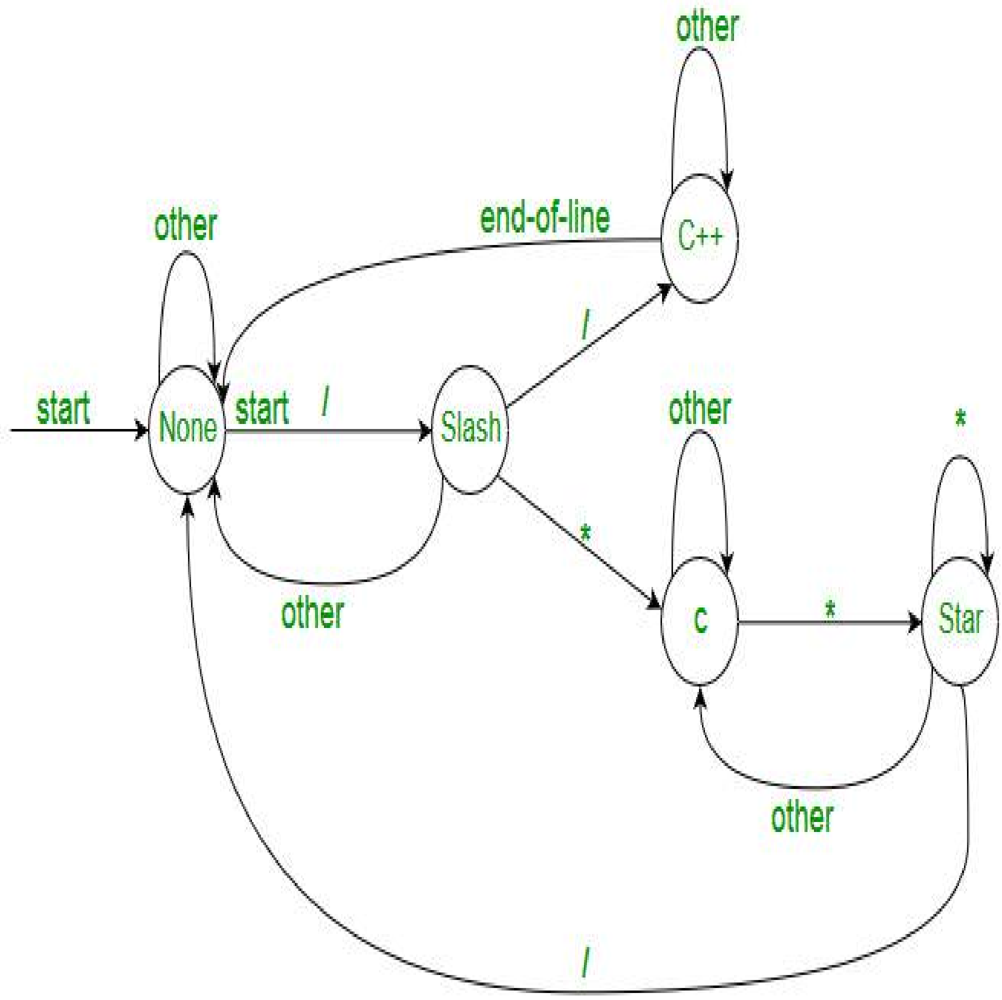
Example of Non-Tokens:

- Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

How Lexical Analyzer functions

1. Tokenization i.e. Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.



The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer –

a = b + c ;   It will generate token sequence like this:
id=id+id;   Where each id refers to it's variable in the symbol table referencing all details

For example, consider the program

```
int main()

{

  // 2 variables

  int a, b;

  a = 10;

 return 0;

}
```

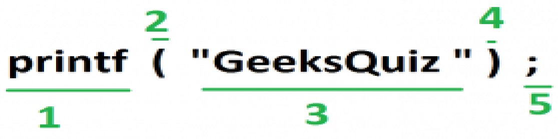
All the valid tokens are:

'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';'

 'a' '=' '10' ';' 'return' '0' ';' '}'

Above are the valid tokens.

As another example, consider below printf statement.



There are 5 valid token in this printf statement.

CODE:

C Program:

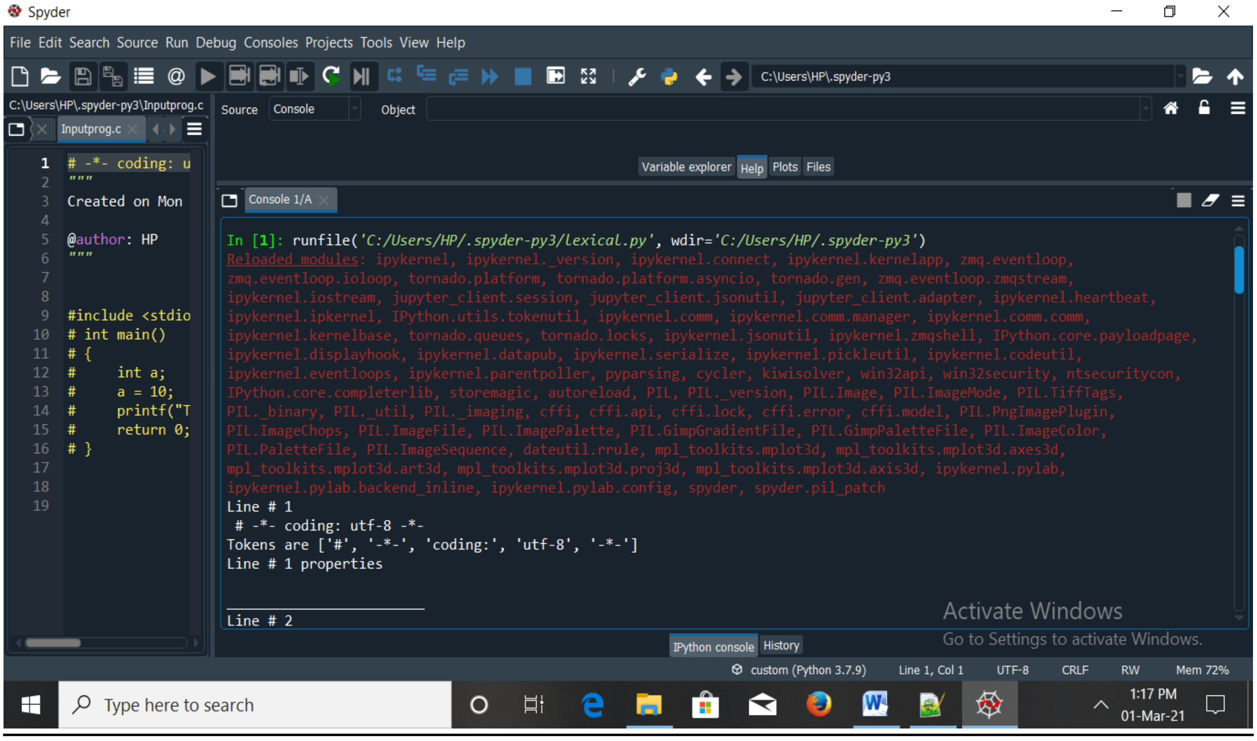
```
include <stdio.h> // This is a header file
int main()
 {
   int a
    a = 10;
  printf("The value of a is %d ",a);
     return 0;    }
```

Python Code:

```
import re
f = open('InputProg.c','r')
operators = { '=': 'Assignment Operator','+': 'Additon Operator', '-' : 'Substraction Operator', '/' : 'Division Operator', '*': 'Multiplication Operator', '++' : 'increment Operator', '--' : 'Decrement Operator'}
optr_keys = operators.keys()
comments = {r'//' : 'Single Line Comment',r'/*' : 'Multiline Comment Start', r'*/' : 'Multiline Comment End', '/**/' : 'Empty Multiline comment'}
comment_keys = comments.keys()
header = {'.h': 'header file'}
header_keys = header.keys()


sp_header_files = {'<stdio.h>':'Standard Input Output Header','<string.h>':'String Manipulation Library'}
macros = {r'#\w+' : 'macro'}
macros_keys = macros.keys()
datatype = {'int': 'Integer','float' : 'Floating Point', 'char': 'Character','long': 'long int'}
datatype_keys = datatype.keys()
keyword = {'return' : 'keyword that returns a value from a block'}
keyword_keys = keyword.keys()
delimiter = {';':'terminator symbol semicolon (;)'}
```

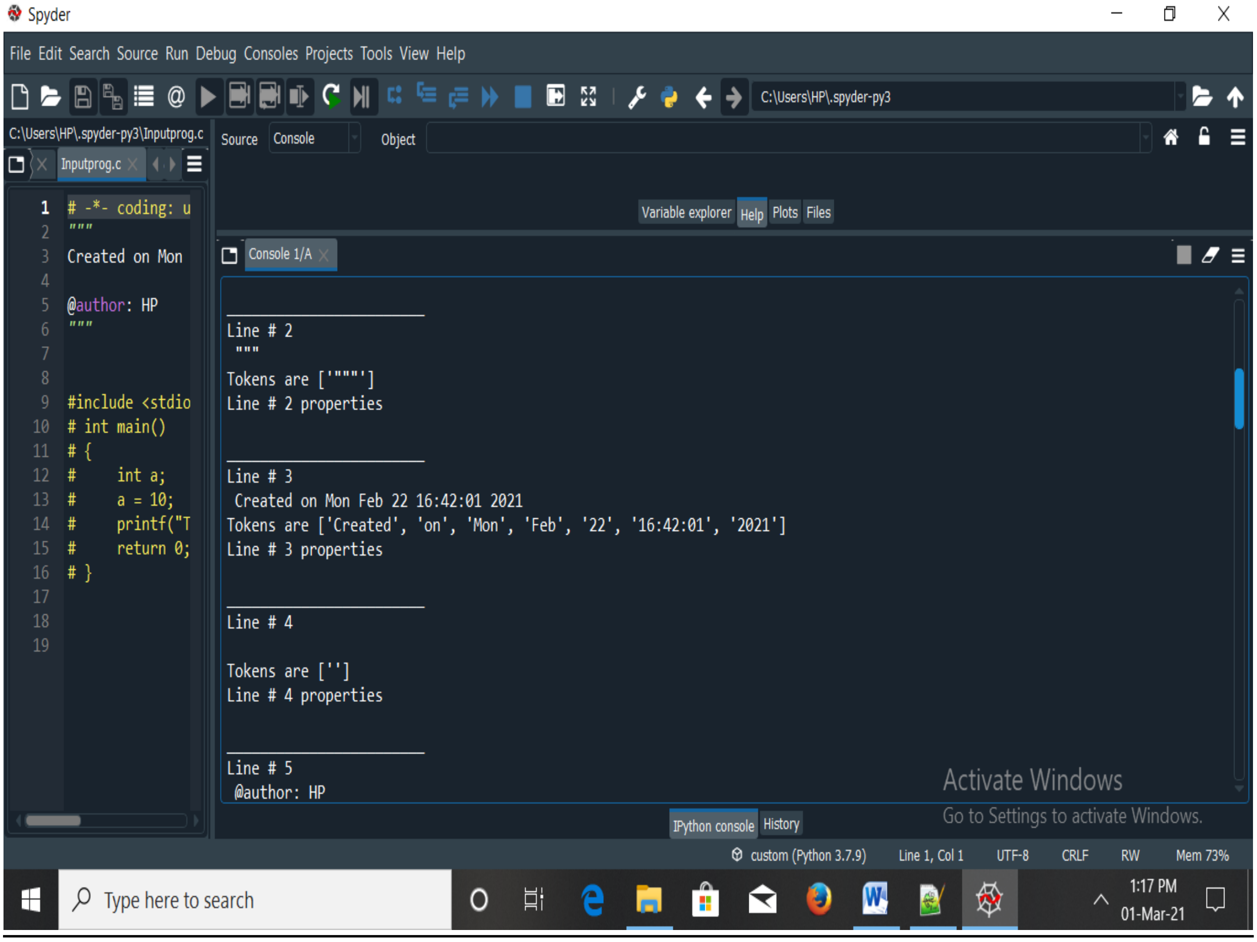
```
delimiter_keys = delimiter.keys()
blocks = {'{' : 'Blocked Statement Body Open', '}':'Blocked Statement Body Closed'}
block_keys = blocks.keys()
builtin_functions = {'printf':'printf prints its argument on the console'}
non_identifiers = ['_','-','+','/','*','`','~','!','@','#','$','%','^','&','*','(',')','=','|','"',':',';','{'
,'}','[',']','<','>','?','/']
numerals = ['0','1','2','3','4','5','6','7','8','9','10']
# Flags
dataFlag = False
i = f.read()
count = 0
program =  i.split('\n')
for line in program:
    count = count+1
    print ("Line #",count,"\n",line)
    tokens = line.split(' ')
    print ("Tokens are",tokens)
    print ("Line #",count,'properties \n')
    for token in tokens:
        if '\r' in token:
            position = token.find('\r')
            token=token[:position]
      # print 1
        if token in block_keys:
            print (blocks[token])
        if token in optr_keys:
            print ("Operator is: ", operators[token])
        if token in comment_keys:
            print ("Comment Type: ", comments[token])
        if token in macros_keys:
            print("Macro is: ", macros[token])
        if '.h' in token:
            print ("Header File is: ",token, sp_header_files[token])
        if '()' in token:
            print ("Function named", token)
        if dataFlag == True and (token not in non_identifiers) and ('()' not in token):
            print ("Identifier: ",token)
        if token in datatype_keys:
            print ("type is: ", datatype[token])
            dataFlag = True
        if token in keyword_keys:
            print (keyword[token])
        if token in delimiter:
            print ("Delimiter" , delimiter[token])
        if '#' in token:
            match = re.search(r'#\w+', token)
            #print ("Header", match.group())
        if token in numerals:
            print (token,type(int(token)))
    dataFlag = False
    print ("______________________")
f.close()
```

Output:


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
Activate Windows
Go to Settings to activate Windows.
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 72%
Type here to search
1:17 PM
01-Mar-21

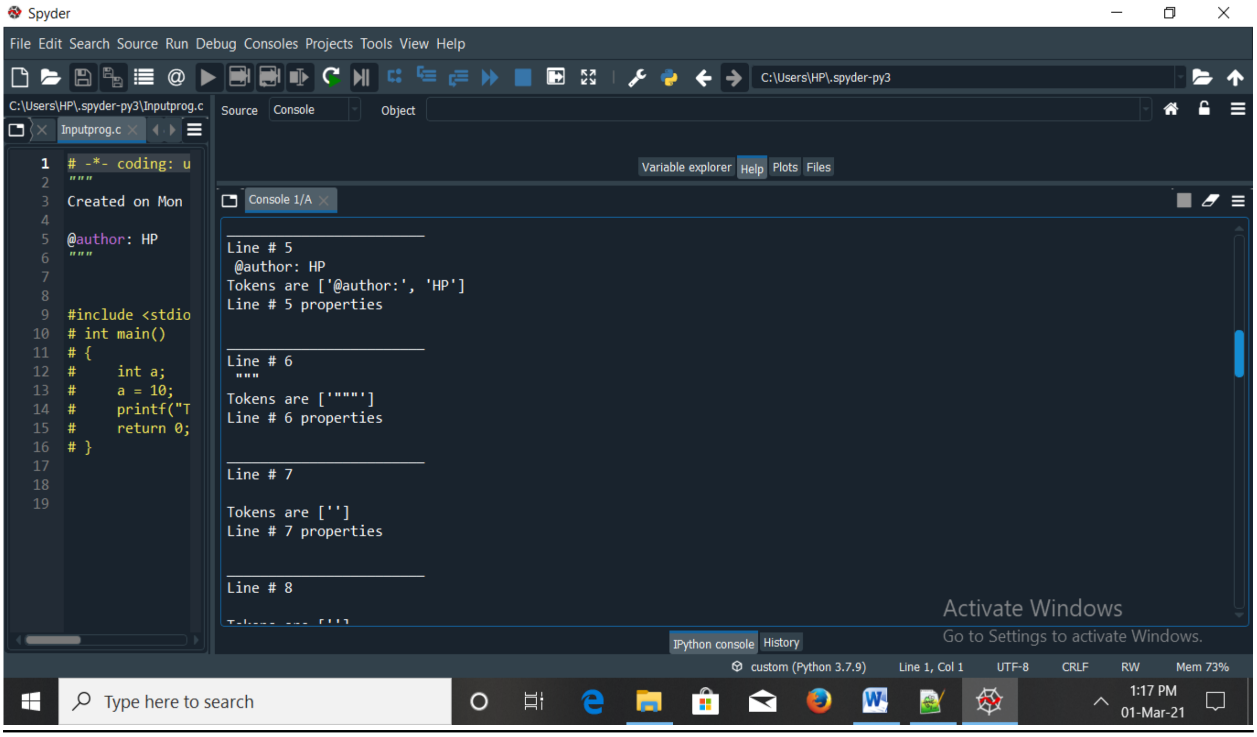
```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
In [1]: runfile('C:/Users/HP/.spyder-py3/lexical.py', wdir='C:/Users/HP/.spyder-py3')
Reloaded modules: ipykernel, ipykernel._version, ipykernel.connect, ipykernel.kernelapp, zmq.eventloop,
zmq.eventloop.ioloop, tornado.platform, tornado.platform.asyncio, tornado.gen, zmq.eventloop.zmqstream,
ipykernel.iostream, jupyter_client.session, jupyter_client.jsonutil, jupyter_client.adapter, ipykernel.heartbeat,
ipykernel.ipkernel, IPython.utils.tokenutil, ipykernel.comm, ipykernel.comm.manager, ipykernel.comm.comm,
ipykernel.kernelbase, tornado.queues, tornado.locks, ipykernel.jsonutil, ipykernel.zmqshell, IPython.core.payloadpage,
ipykernel.displayhook, ipykernel.datapub, ipykernel.serialize, ipykernel.pickleutil, ipykernel.codeutil,
ipykernel.eventloops, ipykernel.parentpoller, pyparsing, cycler, kiwisolver, win32api, win32security, ntsecuritycon,
IPython.core.completerlib, storemagic, autoreload, PIL, PIL._version, PIL.Image, PIL.ImageMode, PIL.TiffTags,
PIL._binary, PIL._util, PIL._imaging, cffi, cffi.api, cffi.lock, cffi.error, cffi.model, PIL.PngImagePlugin,
PIL.ImageChops, PIL.ImageFile, PIL.ImagePalette, PIL.GimpGradientFile, PIL.GimpPaletteFile, PIL.ImageColor,
PIL.PaletteFile, PIL.ImageSequence, dateutil.rrule, mpl_toolkits.mplot3d, mpl_toolkits.mplot3d.axes3d,
mpl_toolkits.mplot3d.art3d, mpl_toolkits.mplot3d.proj3d, mpl_toolkits.mplot3d.axis3d, ipykernel.pylab,
ipykernel.pylab.backend_inline, ipykernel.pylab.config, spyder, spyder.pil_patch
Line # 1
 # -*- coding: utf-8 -*-
Tokens are ['#', '-*-', 'coding:', 'utf-8', '-*-']
Line # 1 properties


______________________
Line # 2
```


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 73%
Type here to search
1:17 PM
01-Mar-21
Activate Windows
Go to Settings to activate Windows.

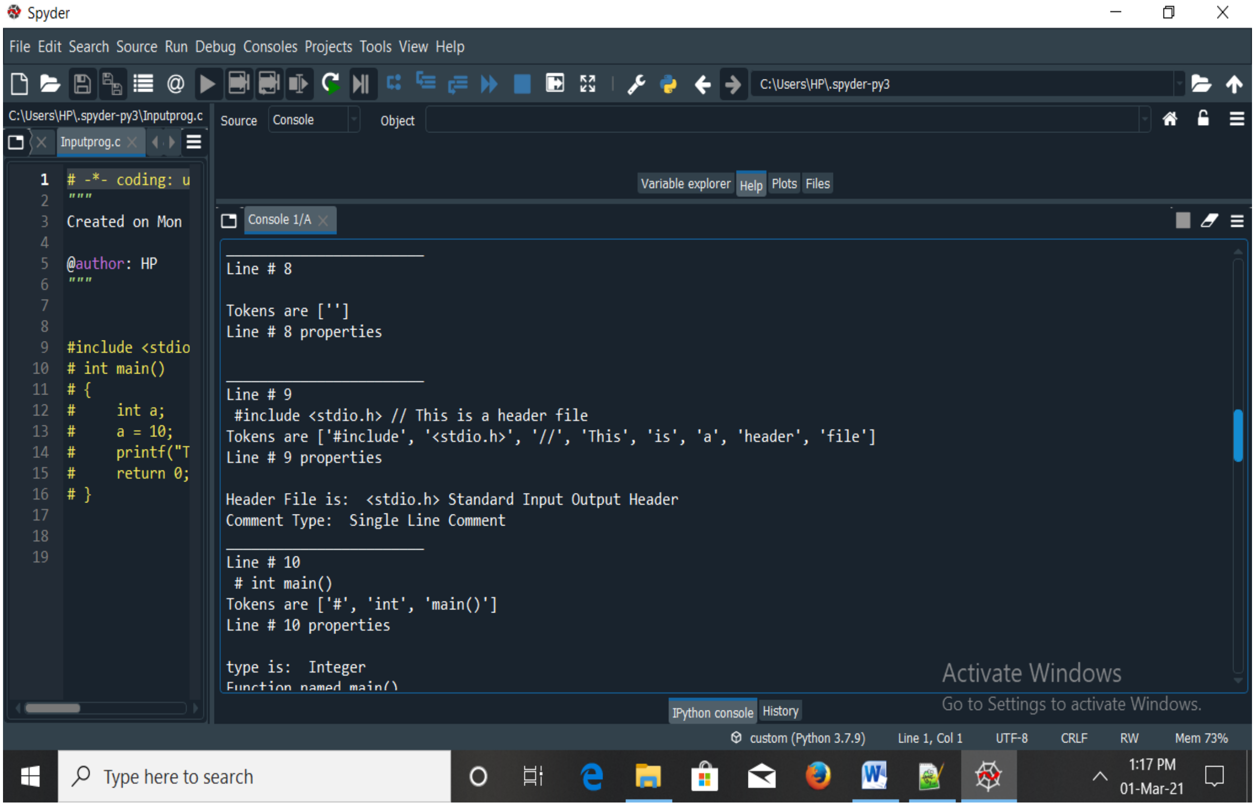
```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
____________________
Line # 2
 """
Tokens are ['"""']
Line # 2 properties

____________________
Line # 3
 Created on Mon Feb 22 16:42:01 2021
Tokens are ['Created', 'on', 'Mon', 'Feb', '22', '16:42:01', '2021']
Line # 3 properties

____________________
Line # 4

Tokens are ['']
Line # 4 properties

____________________
Line # 5
 @author: HP
```


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 73%
Type here to search
1:17 PM
01-Mar-21
Activate Windows
Go to Settings to activate Windows.

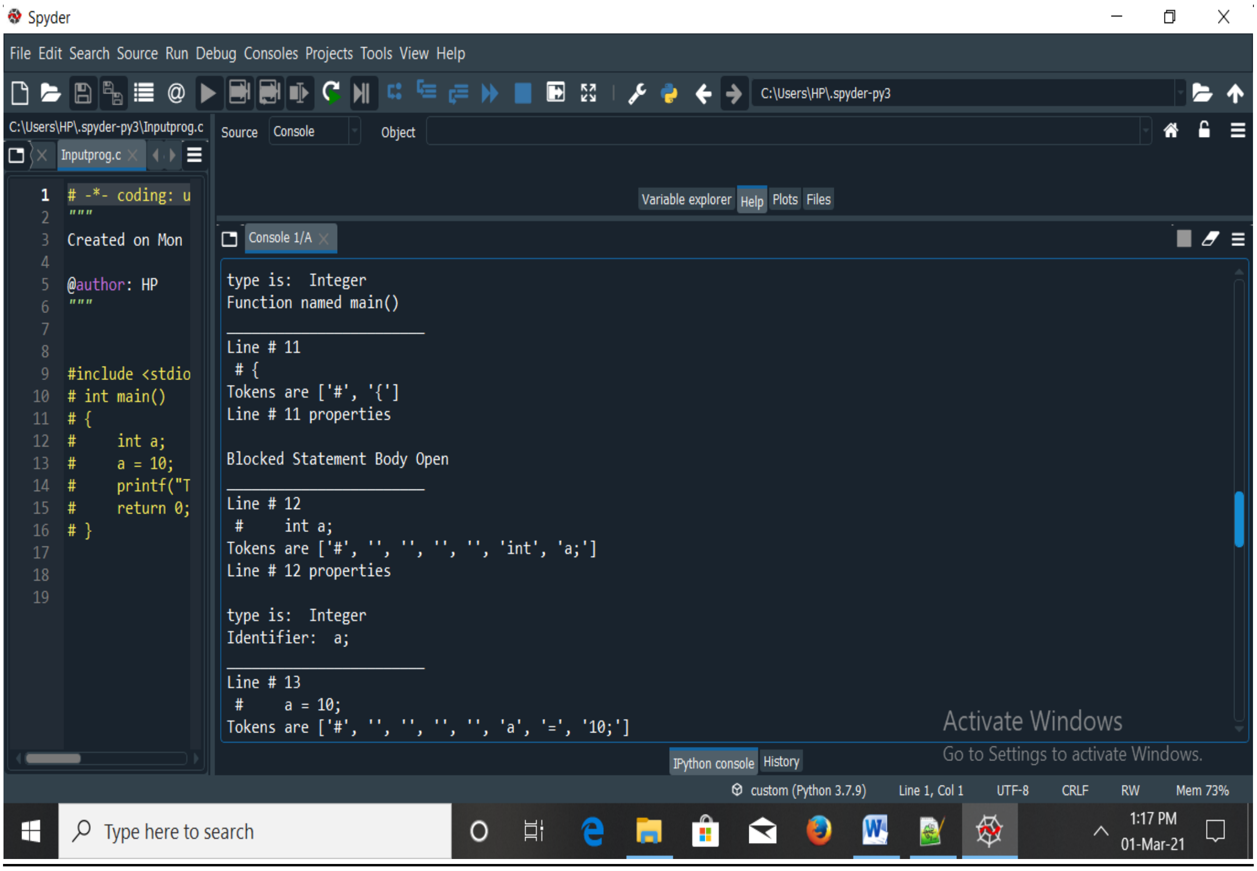
```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
____________________
Line # 5
 @author: HP
Tokens are ['@author:', 'HP']
Line # 5 properties

____________________
Line # 6
 """
Tokens are ['"""']
Line # 6 properties

____________________
Line # 7

Tokens are ['']
Line # 7 properties

____________________
Line # 8
Tokens are ['']
```


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
Activate Windows
Go to Settings to activate Windows.
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 73%
Type here to search
1:17 PM
01-Mar-21

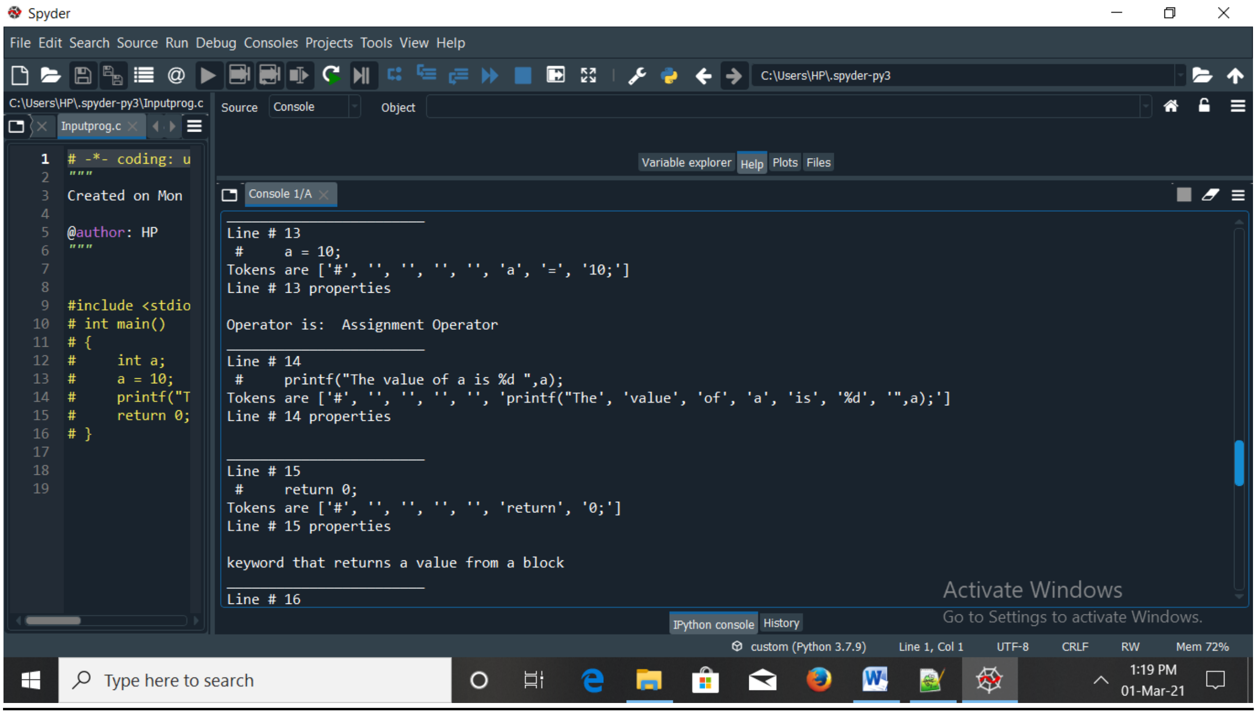
```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
________________
Line # 8

Tokens are ['']
Line # 8 properties

________________
Line # 9
 #include <stdio.h> // This is a header file
Tokens are ['#include', '<stdio.h>', '//', 'This', 'is', 'a', 'header', 'file']
Line # 9 properties

Header File is:  <stdio.h> Standard Input Output Header
Comment Type:  Single Line Comment

________________
Line # 10
 # int main()
Tokens are ['#', 'int', 'main()']
Line # 10 properties

type is:  Integer
Function named main()
```


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
Activate Windows
Go to Settings to activate Windows.
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 73%
Type here to search
1:17 PM
01-Mar-21

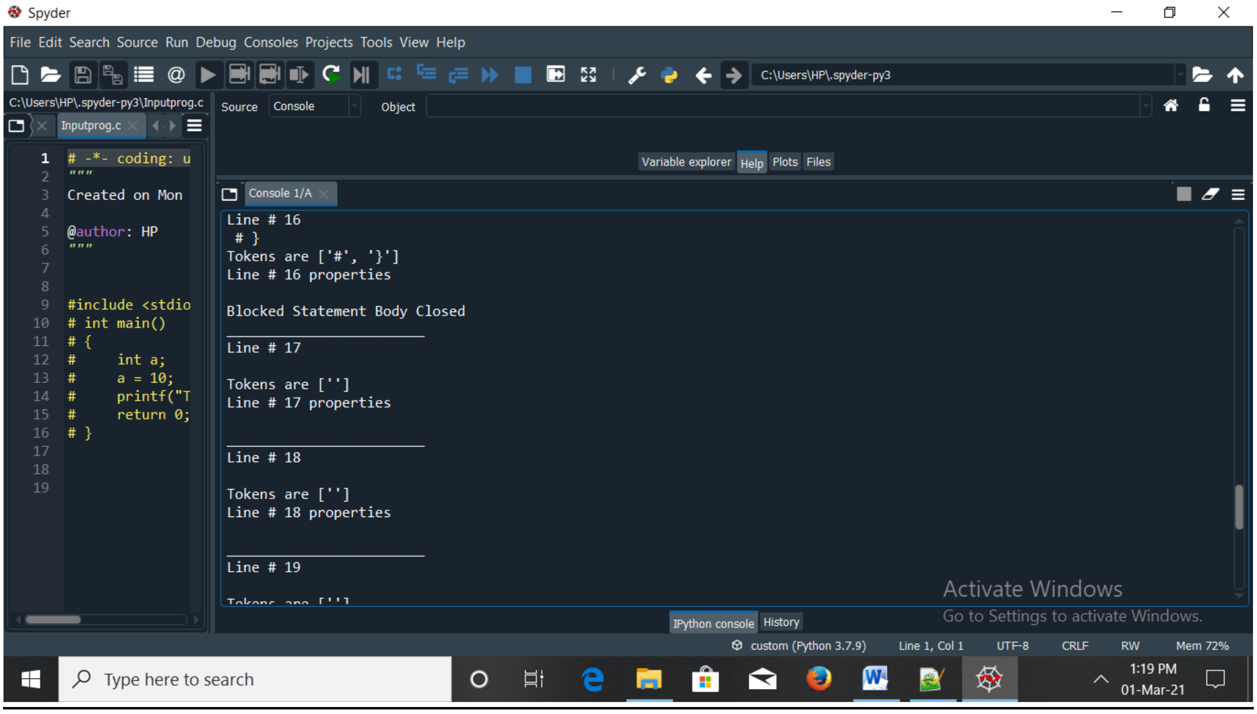
```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
type is:  Integer
Function named main()

________________
Line # 11
 # {
Tokens are ['#', '{']
Line # 11 properties

Blocked Statement Body Open

________________
Line # 12
 #     int a;
Tokens are ['#', '', '', '', '', 'int', 'a;']
Line # 12 properties

type is:  Integer
Identifier:  a;

________________
Line # 13
 #     a = 10;
Tokens are ['#', '', '', '', '', 'a', '=', '10;']
```


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
Activate Windows
Go to Settings to activate Windows.
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 72%
Type here to search
1:19 PM
01-Mar-21

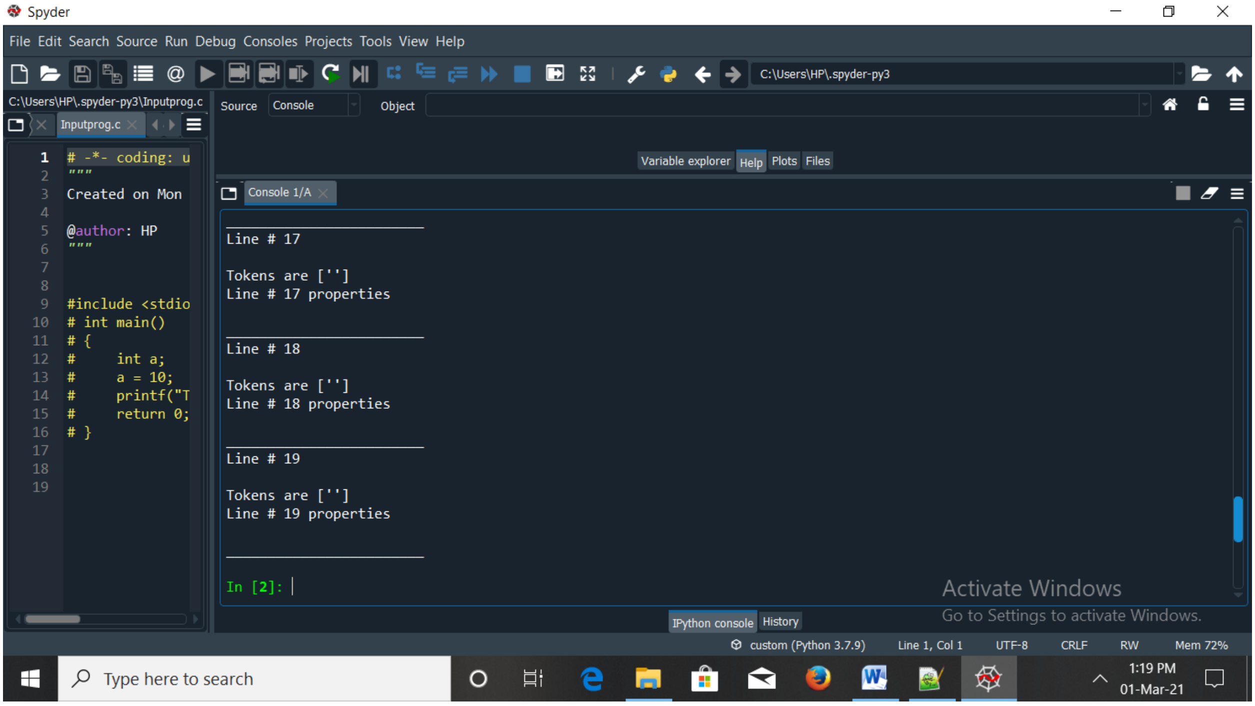
```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
________________________
Line # 13
 #     a = 10;
Tokens are ['#', '', '', '', '', 'a', '=', '10;']
Line # 13 properties

Operator is:  Assignment Operator
________________________
Line # 14
 #     printf("The value of a is %d ",a);
Tokens are ['#', '', '', '', '', 'printf("The', 'value', 'of', 'a', 'is', '%d', '",a);']
Line # 14 properties


________________________
Line # 15
 #     return 0;
Tokens are ['#', '', '', '', '', 'return', '0;']
Line # 15 properties

keyword that returns a value from a block
________________________
Line # 16
```


Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\Inputprog.c
Inputprog.c
Source Console Object
Variable explorer Help Plots Files
Console 1/A
IPython console History
Activate Windows
Go to Settings to activate Windows.
custom (Python 3.7.9) Line 1, Col 1 UTF-8 CRLF RW Mem 72%
Type here to search
1:19 PM
01-Mar-21

```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
Line # 16
 # }
Tokens are ['#', '}']
Line # 16 properties

Blocked Statement Body Closed
________________________
Line # 17

Tokens are ['']
Line # 17 properties


________________________
Line # 18

Tokens are ['']
Line # 18 properties


________________________
Line # 19

Tokens are ['']
```



```
# -*- coding: u
"""
Created on Mon

@author: HP
"""


#include <stdio
# int main()
# {
#     int a;
#     a = 10;
#     printf("T
#     return 0;
# }
```

```
_______________
Line # 17

Tokens are ['']
Line # 17 properties

_______________
Line # 18

Tokens are ['']
Line # 18 properties

_______________
Line # 19

Tokens are ['']
Line # 19 properties

_______________

In [2]:
```

Conclusion:

Thus, we understood and successfully implement Lexical Analyser.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL
COMPUTER ENGINEERING DEPARTMENT
A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 4

Title of Experiment: Implementation of finding First and Follow of given grammer.

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

AIM : Implementation of finding First and Follow of given grammer.

Theory:

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 non-terminal. 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 → 1 2 3 … n and any FIRST( ) contains t then t is in FIRST(α).

First set can be seen as:

$$FIRST(\alpha) = \{ t \mid \alpha \xrightarrow{*} t\,\beta \} \cup \{ \varepsilon \mid \alpha \xrightarrow{*} \varepsilon \}$$

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:

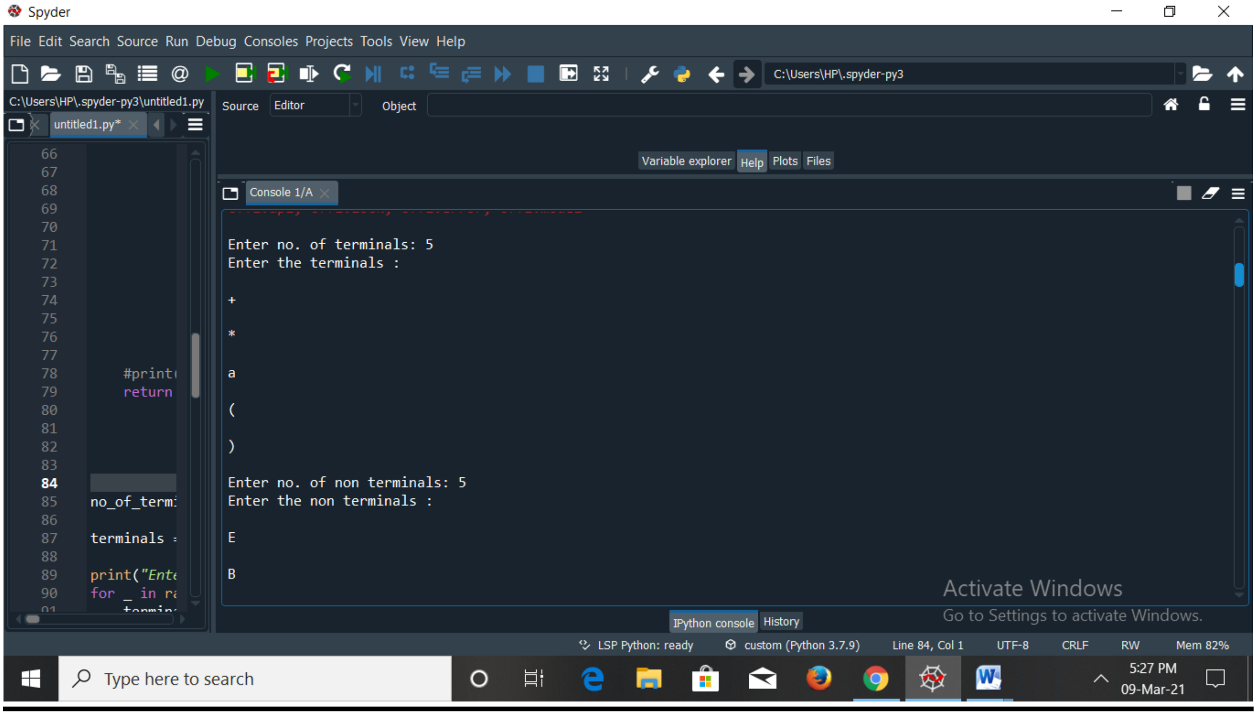
- if α is a start symbol, then FOLLOW() = $
- if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except Ɛ.
- if α is a non-terminal and has a production α → AB, where B Ɛ, then FOLLOW(A) is in FOLLOW(α).

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

Code:

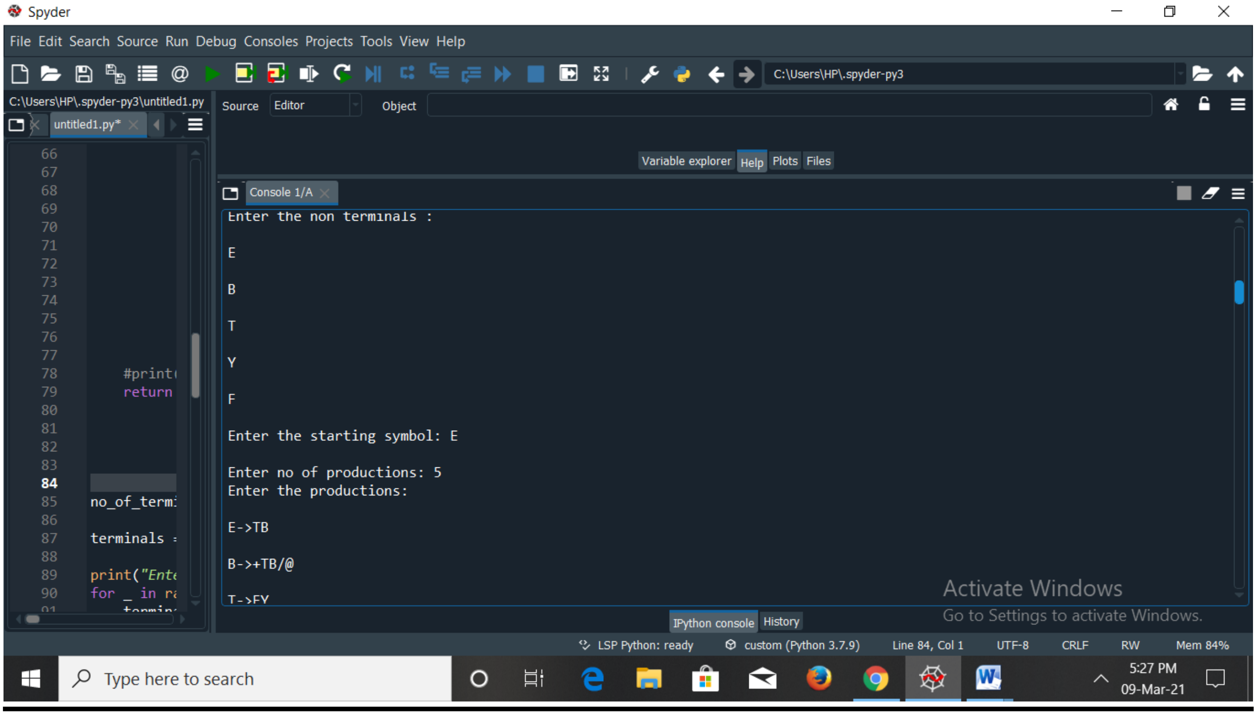
```
import sys
sys.setrecursionlimit(60)
def first(string):
    #print("first({})".format(string))
    first_ = set()
    if string in non_terminals:
        alternatives = productions_dict[string]
        for alternative in alternatives:
            first_2 = first(alternative)
            first_ = first_ |first_2
    elif string in terminals:
        first_ = {string}
    elif string=='' or string=='@':
        first_ = {'@'}
    else:
        first_2 = first(string[0])
        if '@' in first_2:
            i = 1
            while '@' in first_2:
                #print("inside while")
                first_ = first_ | (first_2 - {'@'})
                #print('string[i:]=', string[i:])
                if string[i:] in terminals:
                    first_ = first_ | {string[i:]}
                    break
                elif string[i:] == '':
                    first_ = first_ | {'@'}
                    break
                first_2 = first(string[i:])
                first_ = first_ | first_2 - {'@'}
                i += 1
        else:
            first_ = first_ | first_2
    #print("returning for first({})".format(string),first_)
    return  first_
def follow(nT):
    #print("inside follow({})".format(nT))
    follow_ = set()
    #print("FOLLOW", FOLLOW)
    prods = productions_dict.items()
    if nT==starting_symbol:
        follow_ = follow_ | {'$'}
    for nt,rhs in prods:
        #print("nt to rhs", nt,rhs)
        for alt in rhs:
            for char in alt:
                if char==nT:
                    following_str = alt[alt.index(char) + 1:]
                    if following_str=='':
                        if nt==nT:
                            continue
                        else:
                            follow_ = follow_ | follow(nt)
                    else:
                        follow_2 = first(following_str)
                        if '@' in follow_2:
                            follow_ = follow_ | follow_2-{'@'}
                            follow_ = follow_ | follow(nt)
                        else:
                            follow_ = follow_ | follow_2
    #print("returning for follow({})".format(nT),follow_)
    return follow_
no_of_terminals=int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals :")
for _ in range(no_of_terminals):
    terminals.append(input())
no_of_non_terminals=int(input("Enter no. of non terminals: "))
non_terminals = []
print("Enter the non terminals :")
for _ in range(no_of_non_terminals):
    non_terminals.append(input())
starting_symbol = input("Enter the starting symbol: ")
no_of_productions = int(input("Enter no of productions: "))
productions = []
print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())
#print("terminals", terminals)
#print("non terminals", non_terminals)
#print("productions",productions)
productions_dict = {}
for nT in non_terminals:
    productions_dict[nT] = []
#print("productions_dict",productions_dict)
for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("/")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)
#print("productions_dict",productions_dict)
#print("nonterm_to_prod",nonterm_to_prod)
#print("alternatives",alternatives)
FIRST = {}
FOLLOW = {}
for non_terminal in non_terminals:
    FIRST[non_terminal] = set()
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = set()
#print("FIRST",FIRST)
for non_terminal in non_terminals:
    FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)
#print("FIRST",FIRST)
FOLLOW[starting_symbol] = FOLLOW[starting_symbol] | {'$'}
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)
#print("FOLLOW", FOLLOW)
print("{: ^20}{: ^20}{: ^20}".format('Non Terminals','First','Follow'))
for non_terminal in non_terminals:
    print("{: ^20}{: ^20}{:
^20}".format(non_terminal,str(FIRST[non_terminal]),str(FOLLOW[non_terminal])))
```

OUTPUT:



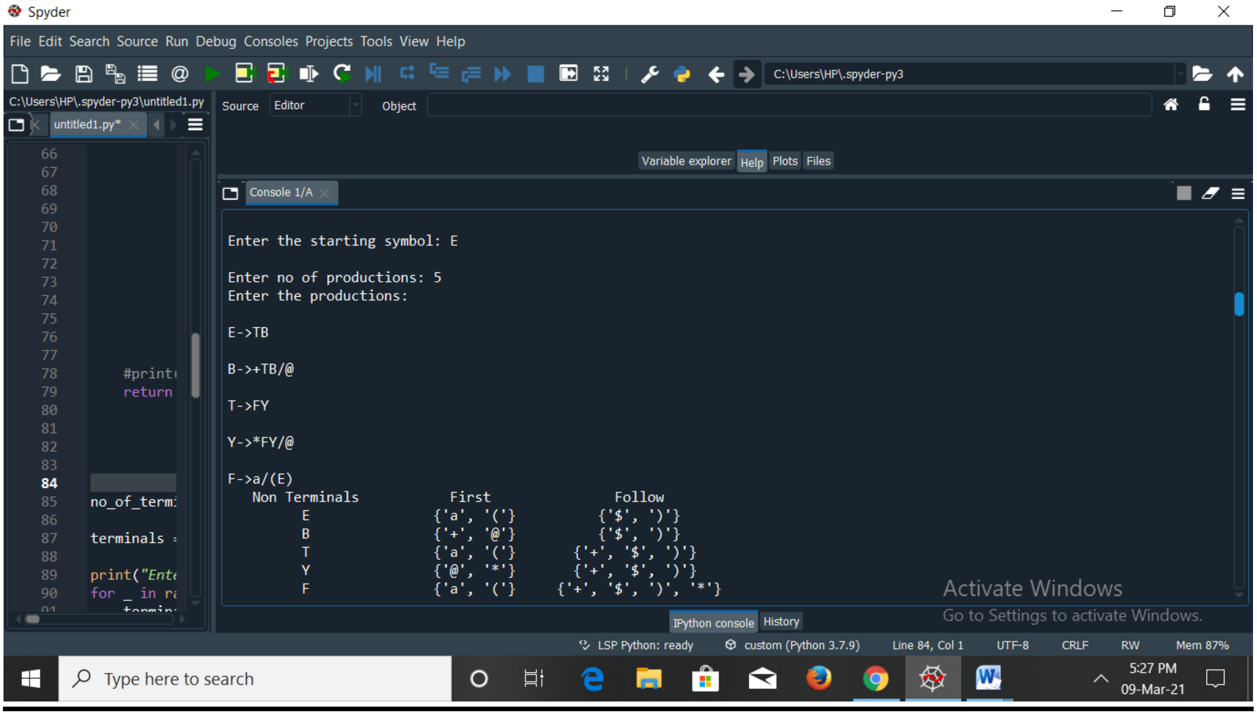
Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\untitled1.py
untitled1.py*
Source Editor Object
Variable explorer Help Plots Files
Console 1/A
#print(
return
no_of_term
terminals =
print("Ente
for _ in ra
IPython console History
Activate Windows
Go to Settings to activate Windows.
LSP Python: ready custom (Python 3.7.9) Line 84, Col 1 UTF-8 CRLF RW Mem 82%
Type here to search
5:27 PM
09-Mar-21

```
Enter no. of terminals: 5
Enter the terminals :

+

*

a

(

)

Enter no. of non terminals: 5
Enter the non terminals :

E

B
```



Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\HP\.spyder-py3
C:\Users\HP\.spyder-py3\untitled1.py
untitled1.py*
Source Editor Object
Variable explorer Help Plots Files
Console 1/A
#print(
return
no_of_term
terminals =
print("Ente
for _ in ra
IPython console History
Activate Windows
Go to Settings to activate Windows.
LSP Python: ready custom (Python 3.7.9) Line 84, Col 1 UTF-8 CRLF RW Mem 84%
Type here to search
5:27 PM
09-Mar-21

```
Enter the non terminals :

E

B

T

Y

F

Enter the starting symbol: E

Enter no of productions: 5
Enter the productions:

E->TB

B->+TB/@

T->FY
```



```
        #print
        return


no_of_term
terminals =
print("Ente
for _ in ra
        termin
```

```
Enter the starting symbol: E

Enter no of productions: 5
Enter the productions:

E->TB

B->+TB/@

T->FY

Y->*FY/@

F->a/(E)
   Non Terminals           First              Follow
         E              {'a', '('}          {'$', ')'}
         B              {'+', '@'}          {'$', ')'}
         T              {'a', '('}       {'+', '$', ')'}
         Y              {'@', '*'}       {'+', '$', ')'}
         F              {'a', '('}     {'+', '$', ')', '*'}
```

Conclusion:

Thus, we understood and successfully implementation of finding First and Follow of given grammar.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 5

Title of Experiment: Study of Parser – LL(1)

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

AIM: Study of Parser – LL(1)

Theory:

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table- driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).

LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

Algorithm:

```
Input:
  string ω
  parsing table M for grammar G

Output:
  If ω is in L(G) then left-most derivation
  of ω, error otherwise.

Initial State : $S on stack (with S being start
  symbol) ω$ in the input buffer

SET ip to point the first symbol of

ω$. repeat
  let X be the top stack symbol and a the symbol pointed by ip.

  if X∈ Vt or
    $ if X = a
      POP X and advance
    ip. else
      error()
    endif

  else /* X is non-terminal */
    if M[X,a] = X → Y1, Y2,... Yk
```

```
         POP X
        PUSH Yk, Yk-1,... Y1 /* Y1 on top */
        Output the production X → Y1, Y2,...
        Yk
      else

          error()
      endif
    endif
until X = $     /* empty stack */
```

A grammar G is LL(1) if A → α | β are two distinct productions of G:

- for no terminal, both α and β derive strings beginning with a.
- at most one of α and β can derive empty string.
- if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

SOURCE CODE:

```
import re
r = int(input('Enter no. of non-terminals:
')) c = int(input('Enter no. of terminals:
')) + 1 non = []
ter = []
table = []
stack = ['$', 'S']
ptr = 0
print('Enter non-
terminals') for i in
range(r):
   non.append(input
())          print('Enter
terminals')  for  i  in
range(c-1):
   ter.append(input())
ter.append('$')
print('Enter productions for:
') temp_list = []
for i in range(r):
   for j in
   range(c):
      print(non[i], '&', ter[j],':' )
      temp_list.append(input
      ())
   table.append(temp_list)
inp_str = input('Enter string to parse: ') + '$'
while(True):
   if(re.match('[A-z]', stack[-1]) and inp_str[ptr] != '$'):
      row = non.index(stack.pop())
      col =
      ter.index(inp_str[ptr])
      if(table[row][col] != '*'):
         temp = table[row][col][::-1]
         while(temp != ''):
            stack.append(temp[0])
```

```
     temp = temp.replace(temp[0], '')
  if(stack[-1] == inp_str[ptr] and inp_str[ptr] !=
    '$'): ptr = ptr + 1
    stack.pop()
  if(stack[-1] == '$'):
    if(inp_str[ptr] == '$'):
      print('String is
      accepted') break
    else:
      print('String is not
      accepted') break

  elif(inp_str[ptr] == '$'):
    print('String is not
    accepted') break
```

OUTPUT:

```
Enter no. of non-terminals:
3 Enter no. of terminals: 3
Enter non-terminals
S
A
B
Enter
terminals c
b
a
Enter productions
for: S & c :
-
S & b :
-
S & a :
aABb
S & $ :
-
A & c :
c
A & b :
-
A & a :
aAc
A & $ :
-
B & c :
c
B & b :
bB
B & a :
```

```
-
 B & $ :
 -
 Enter string to parse:
 accb String is
 accepted
```

Conclusion: We have studied about LL(1) Parser.

GHARDA FOUNDATION’S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 6

Title of Experiment: Implementation of Recursive Desent Parser.

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

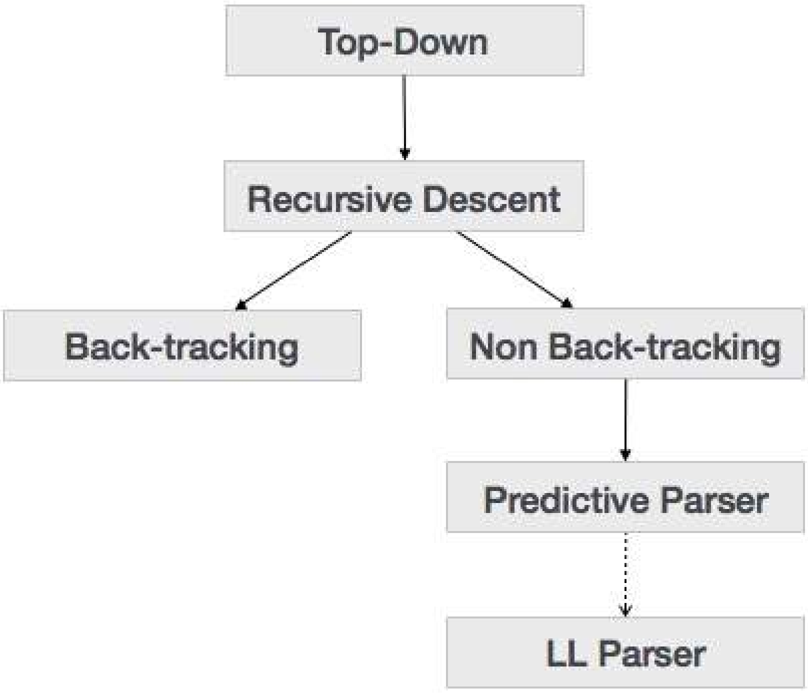
AIM : Implementation of Recursive Desent Parser.

DATE: 22/02/2021

Theory:

Recursive Desent Parser:

The Top-Down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



Recursive Descent Parsing

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**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

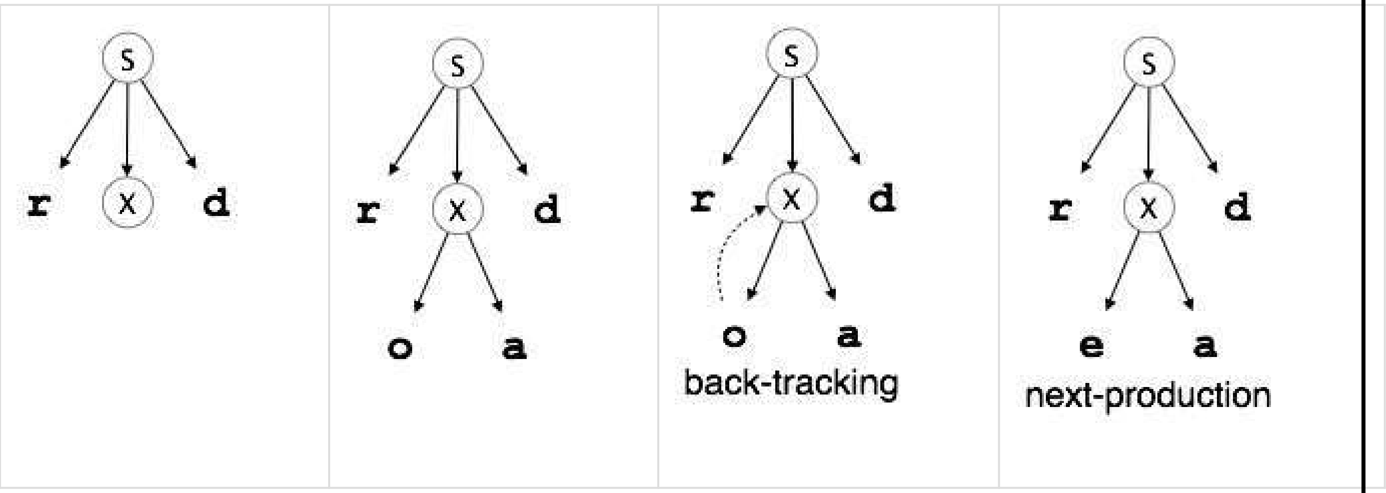
```
S → rXd | rZd
X → oa | ea
Z → ai
```

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal

'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

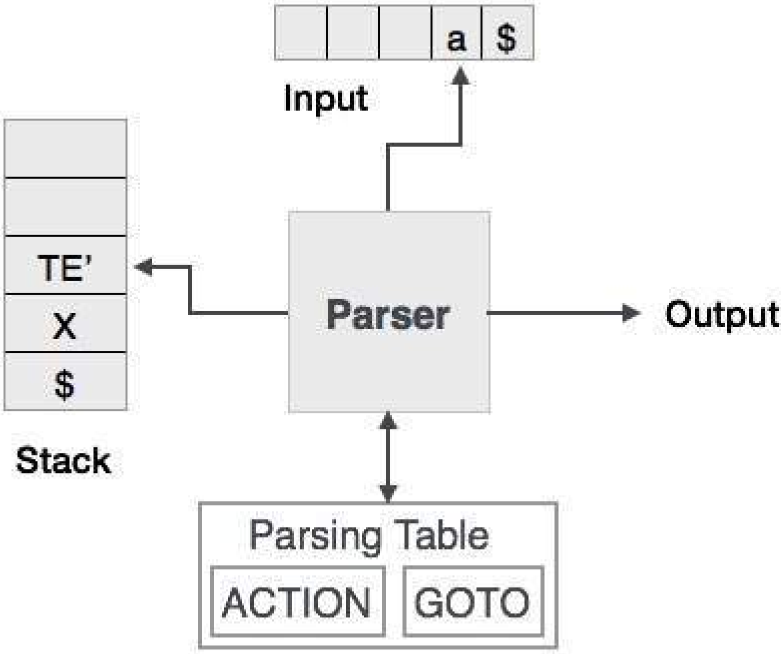
Now the parser matches all the input letters in an ordered manner. The string is accepted.



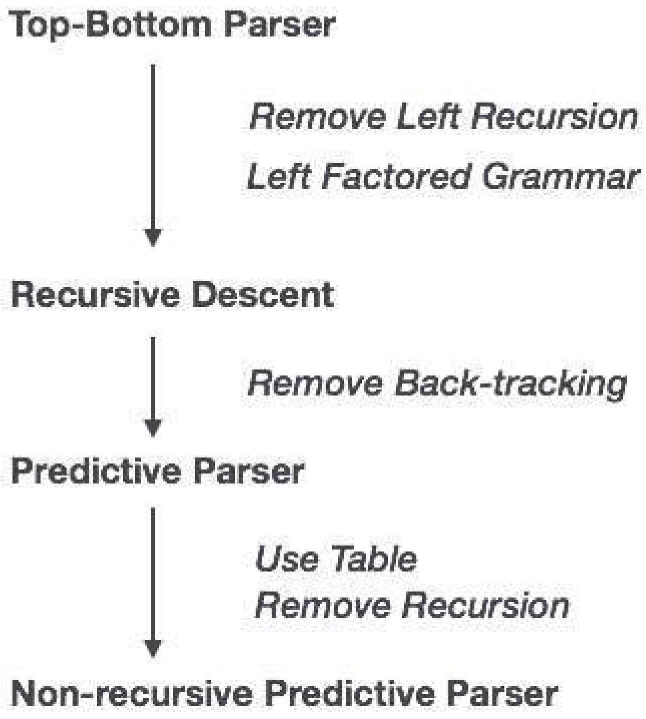
Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

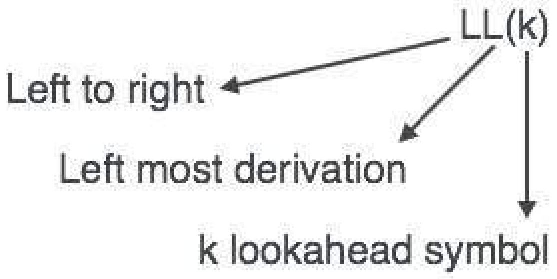


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

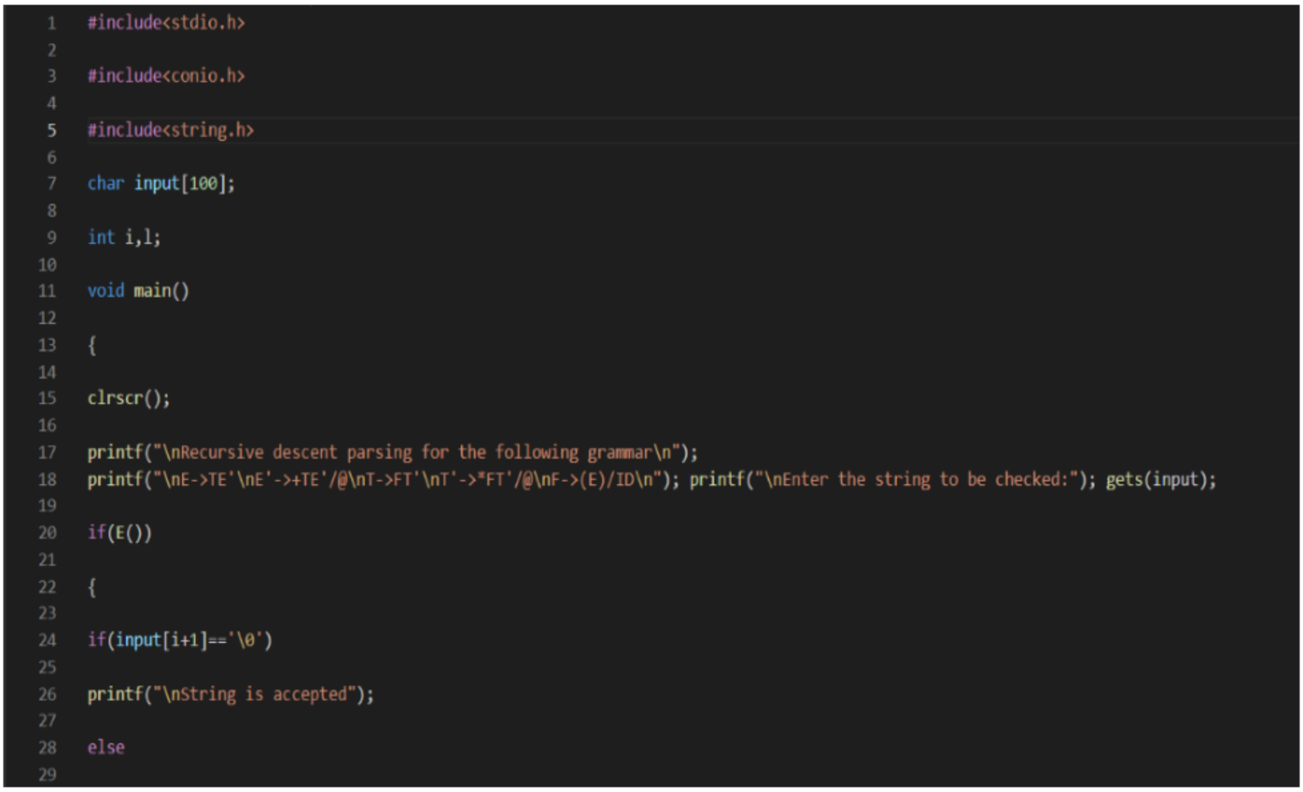
```
Input:
  string ω
  parsing table M for grammar G

Output:
  If ω is in L(G) then left-most derivation of ω,
  error otherwise.

Initial State : $S on stack (with S being start symbol)
  ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat
  let X be the top stack symbol and a the symbol pointed by ip.

  if X∈ V_t or $
    if X = a
      POP X and advance ip.
    else
      error()
    endif

  else     /* X is non-terminal */
    if M[X,a] = X → Y1, Y2,... Yk
      POP X
      PUSH Yk, Yk-1,... Y1 /* Y1 on top */
      Output the production X → Y1, Y2,... Yk
    else
      error()
    endif
  endif
until X = $           /* empty stack */
```

A grammar G is LL(1) if A → α | β are two distinct productions of G:

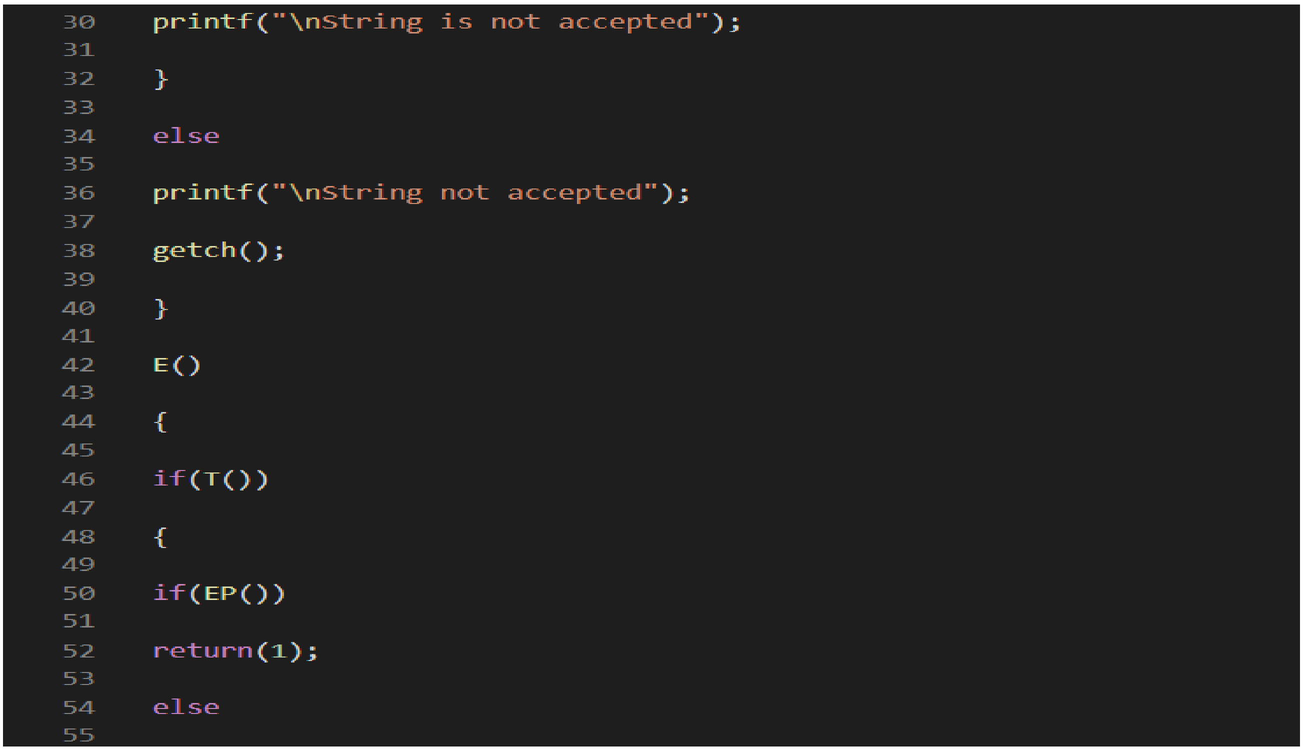
- for no terminal, both α and β derive strings beginning with a.
- at most one of α and β can derive empty string.
- if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

Code:

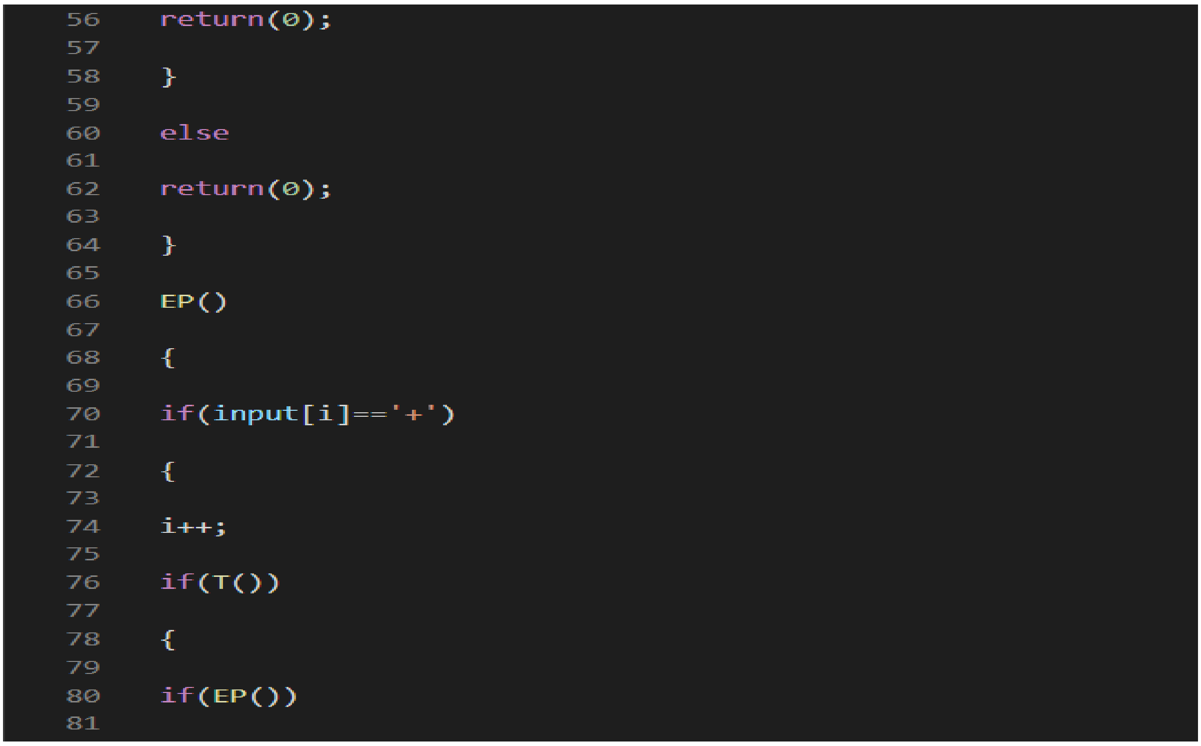
Source Code:



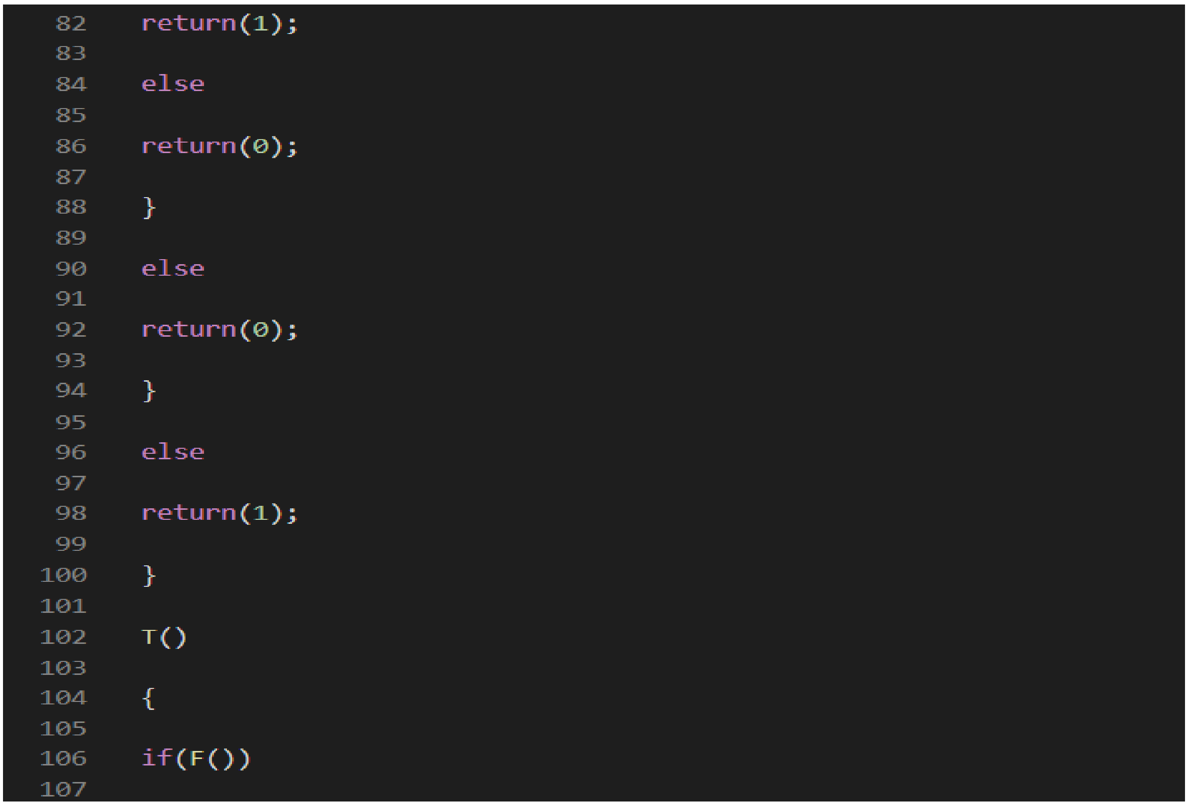
```
#include<stdio.h>

#include<conio.h>

#include<string.h>

char input[100];

int i,l;

void main()

{

clrscr();

printf("\nRecursive descent parsing for the following grammar\n");
printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n"); printf("\nEnter the string to be checked:"); gets(input);

if(E())

{

if(input[i+1]=='\0')

printf("\nString is accepted");

else

```



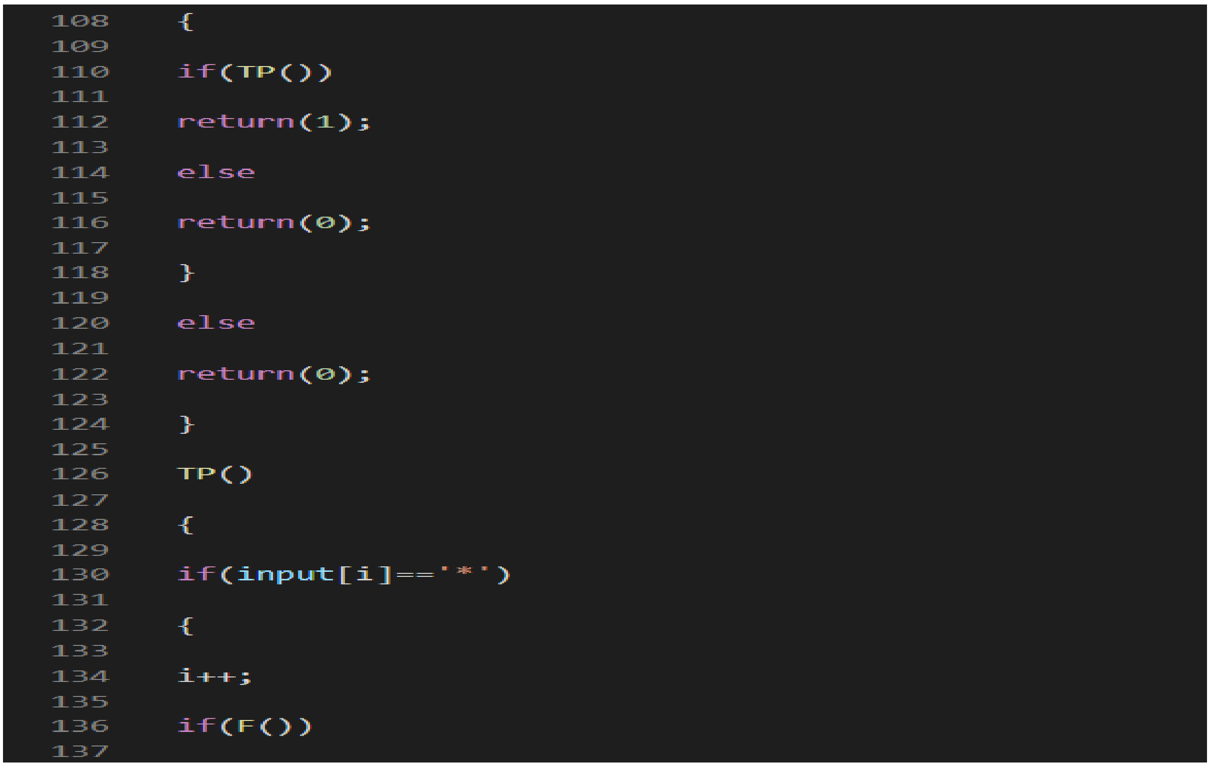
```
printf("\nString is not accepted");

}

else

printf("\nString not accepted");

getch();

}

E()

{

if(T())

{

if(EP())

return(1);

else

```



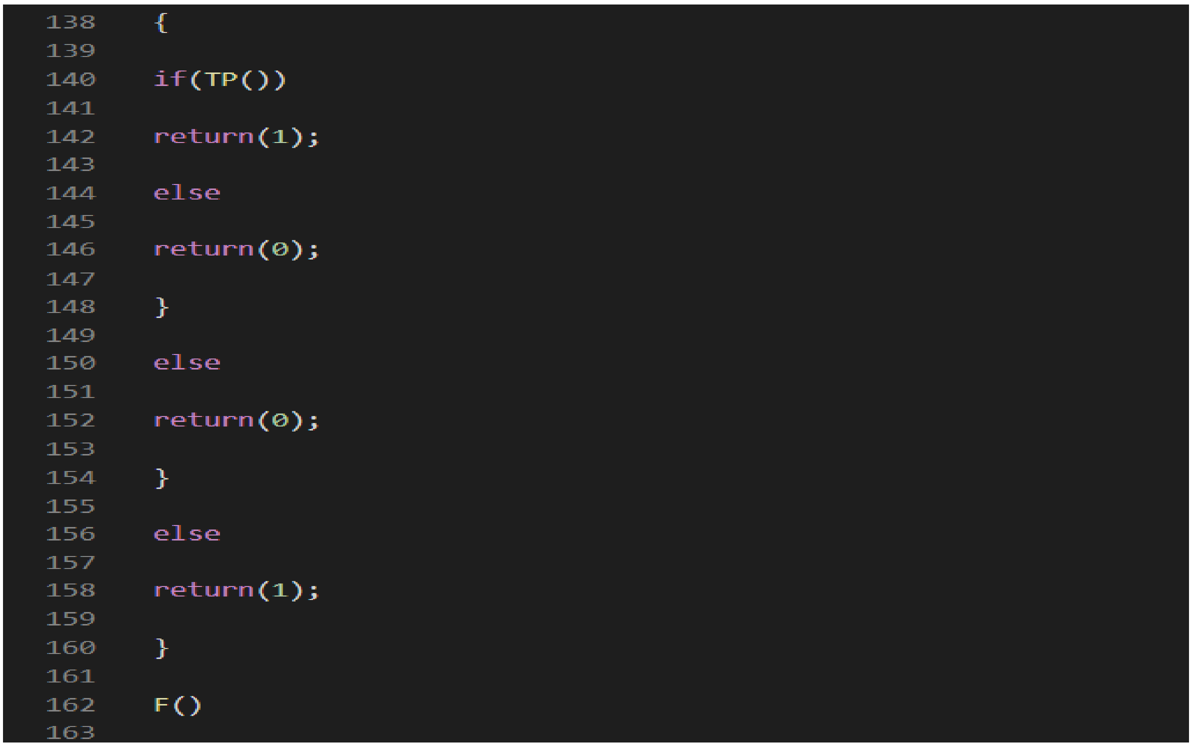
```
return(0);

}

else

return(0);

}

EP()

{

if(input[i]=='+')

{

i++;

if(T())

{

if(EP())

```



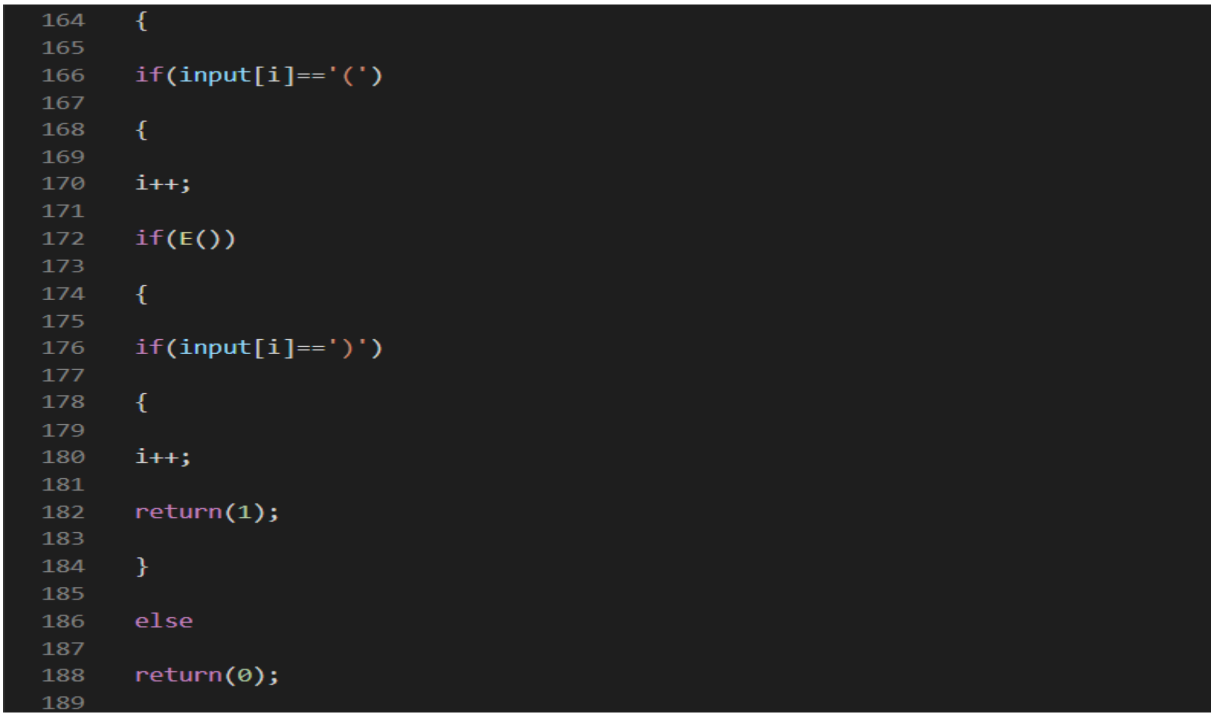
```
return(1);

else

return(0);

}

else

return(0);

}

else

return(1);

}

T()

{

if(F())

```



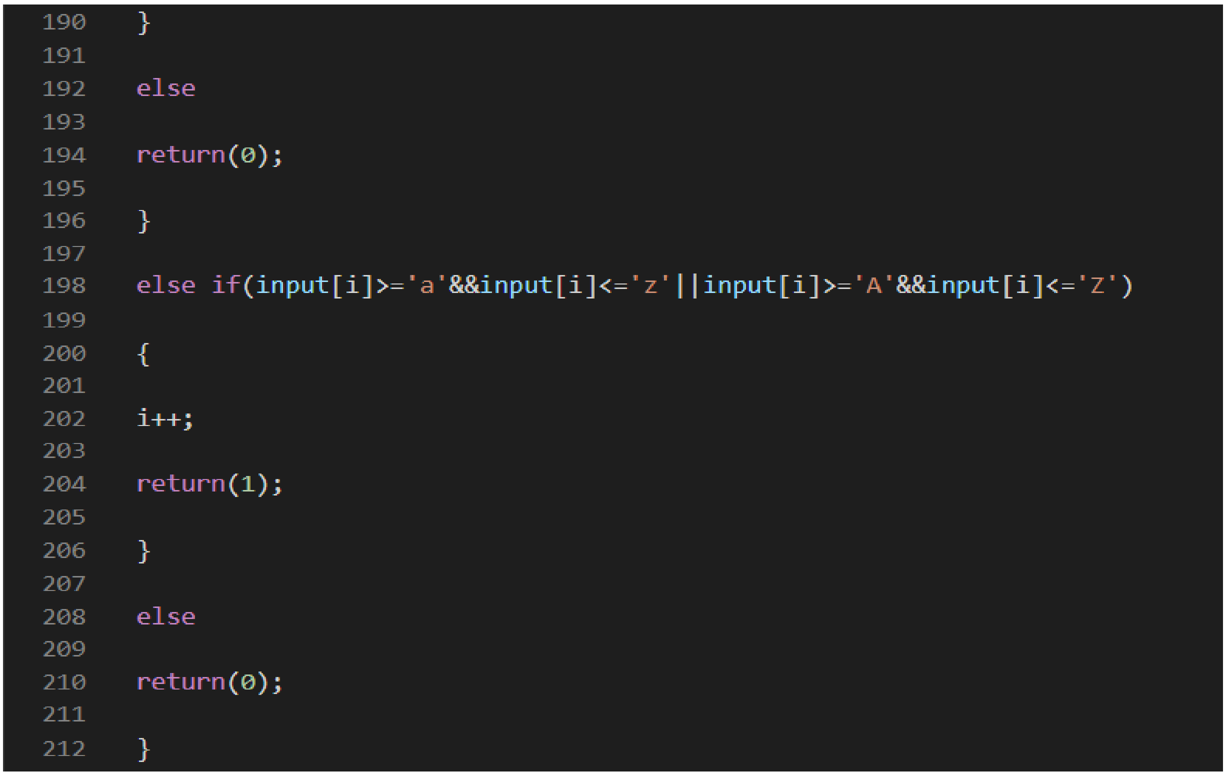
```
{

if(TP())

return(1);

else

return(0);

}

else

return(0);

}

TP()

{

if(input[i]=='*')

{

i++;

if(F())

```



```
{

if(TP())

return(1);

else

return(0);

}

else

return(0);

}

else

return(1);

}

F()

```

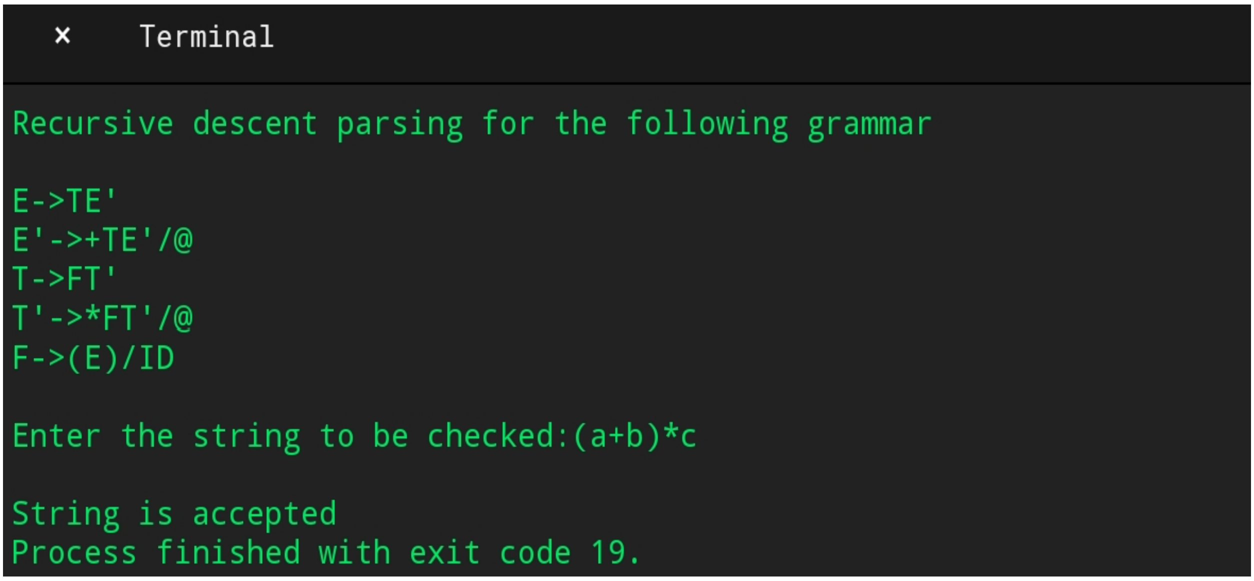


```
{

if(input[i]=='(')

{

i++;

if(E())

{

if(input[i]==')')

{

i++;

return(1);

}

else

return(0);

```

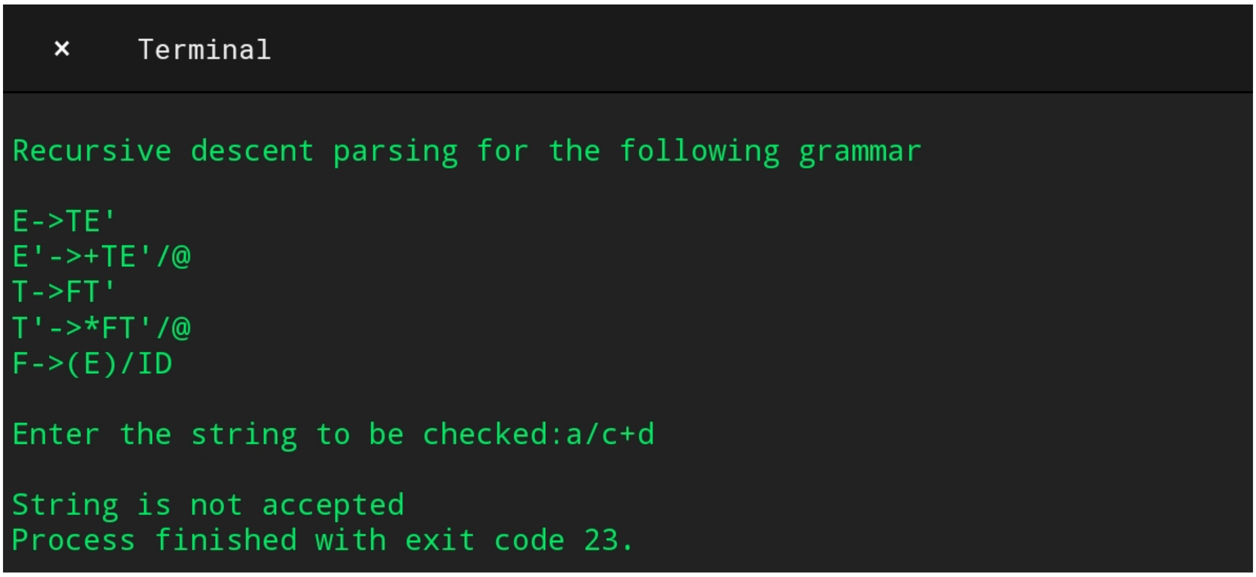


```
}

else

return(0);

}

else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')

{

i++;

return(1);

}

else

return(0);

}
```

Output:



```
Recursive descent parsing for the following grammar

E->TE'
E'->+TE'/@
T->FT'
T'->*FT'/@
F->(E)/ID

Enter the string to be checked:(a+b)*c

String is accepted
Process finished with exit code 19.
```



```
Recursive descent parsing for the following grammar

E->TE'
E'->+TE'/@
T->FT'
T'->*FT'/@
F->(E)/ID

Enter the string to be checked:a/c+d

String is not accepted
Process finished with exit code 23.
```

Conclusion:

Thus, we understood and successfully implementation of Recursive Desent Parser.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 7

Title of Experiment: Implementation of code generation phase of compiler.

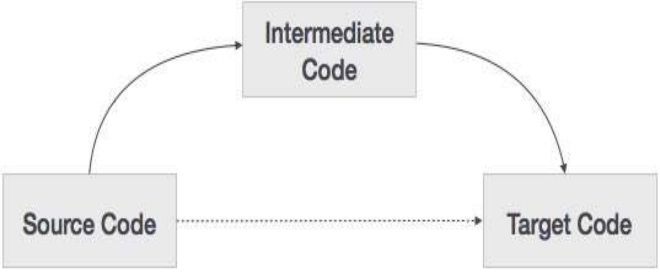
Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

Aim: Implementation of code generation phase of compiler.

Theory:

Intermediate Code



If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.

The second part of compiler, synthesis, is changed according to the target machine.

It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

a = b + c * d;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

r1 = c * d;

r2 = b + r1;

a = r2

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms: quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg1	arg2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

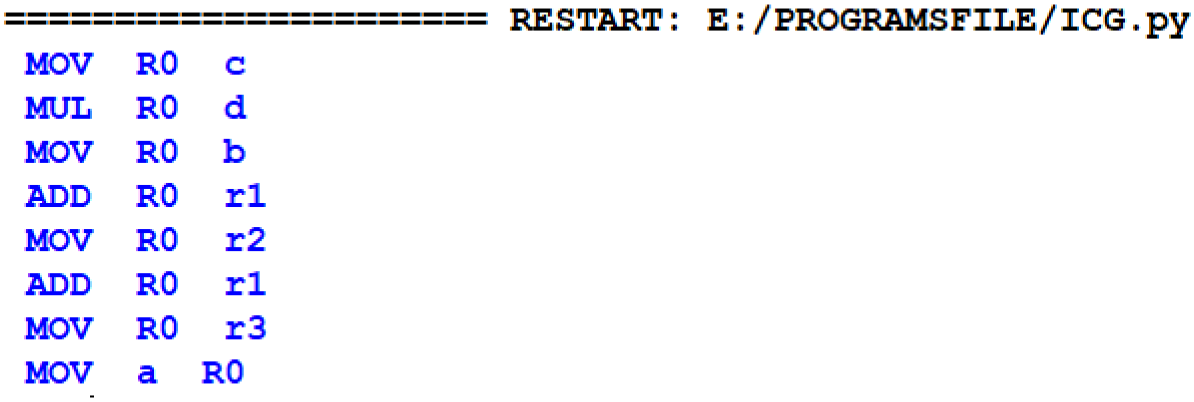
Input File:

```
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3	?	a
```

SOURCE CODE:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 10 21:27:24 2019

@author: Gaurav
"""



file = open('quad.txt','r')
content = file.readlines()
for line in content:
    splitCode = line.split()
    print(" MOV  R0 ",splitCode[1])
    if(splitCode[0]=='+'):
        print(" ADD  R0 ",splitCode[2])
    elif(splitCode[0]=='-'):
        print(" SUB  R0 ",splitCode[2])
    elif(splitCode[0]=='*'):
        print(" MUL  R0 ",splitCode[2])
    elif(splitCode[0]=='='):
        print(" MOV ",splitCode[3],"  R0 ")
```

OUTPUT:



```
====================== RESTART: E:/PROGRAMSFILE/ICG.py
 MOV   R0   c
 MUL   R0   d
 MOV   R0   b
 ADD   R0   r1
 MOV   R0   r2
 ADD   R0   r1
 MOV   R0   r3
 MOV   a   R0
```

Conclusion: We have studied Intermediate Code Generation and implemented same in the form of Quadruples representation.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 8

Title of Experiment: Study of Lexical Analysis Tool: Flex

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

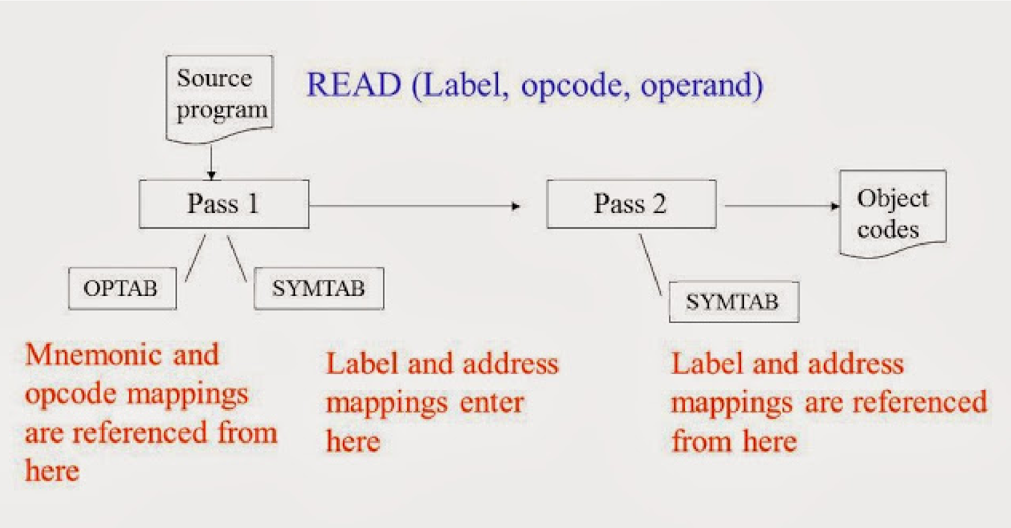
Signature of Subject Teacher

Aim: Implementation of two pass Assembler.

Theory:

Two Pass Assembler

The two pass assembler performs two passes over the source program
In the first pass, it reads the entire source program, looking only for label definitions. All the labels are collected, assigned address, and placed in the symbol table in this pass, no instructions as assembled and at the end the symbol table should contain all the labels defined in the program. To assign address to labels, the assembles maintains a Location Counter (LC).
In the second pass the instructions are again read and are assembled using the symbol table. Basically, the assembler goes through the program one line at a time and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are.


Source program
READ (Label, opcode, operand)
Pass 1
Pass 2
Object codes
OPTAB
SYMTAB
SYMTAB
Mnemonic and opcode mappings are referenced from here
Label and address mappings enter here
Label and address mappings are referenced from here

Difference between One Pass and Two Pass Assemblers

The difference between one pass and two pass assemblers are:
A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references (the problem of forward referencing) and assemble code in one pass. The one pass assembler prepares an intermediate file, which is used as input by the two pass assembler.
A two pass assembler does two passes over the source file (the second pass can be over an intermediate file generated in the first pass of the assembler). In the first pass all it does is looks for label definitions and introduces them in the symbol table (a dynamic table which includes the label name and address for each label in the source program). In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations into machine codes and so on.

Input:

```
START 200
MOVER AREG,FIRST
ADD AREG,SECOND
MOVEM AREG,RESULT
LABEL MULT AREG,THREE
BC LE,LABEL
PRINT RESULT
RESULT DS '1'
```

```
FIRST DC '5'
SECOND DC '7'
THREE DC '1'
END
```

Source Code:

```
from collections import OrderedDict
from prettytable import PrettyTable
REG = {'AREG':'01','BREG':'02','CREG':'03','DREG':'04'}
INS_CODE = {'STOP':'00','ADD':'01','SUB':'02','MULT':'03','MOVER':'04','MOVEM':'05',
         'COMP':'06','BC':'07','DIV':'08','READ':'09','PRINT':'10','LE':'11'}
DS = {'DC':'01','DS':'02'}
AD = {'START':'01','END':'02','ORIGIN':'03'}
Literals = OrderedDict()
Symbols = OrderedDict()
a = []
represent = []
table = []
Sym_table = []
Literal_table = []
code = []
c = 0
lit = 0
j = 0
pt = 0
sym_count = 1
def assembler(line):
  global c,sym_count,represent,lit,Symbols,Literal,a
  words = line.split(" ")
  if(words[0] in INS_CODE):
    represent.append(c)
    represent.append("(IS,"+INS_CODE[words[0]]+")")
    spli = words[1].split(",")
    if(spli[0] in REG):
      if('=' in list(spli[1])):
        lit = lit + 1
        represent.append(REG[spli[0]].replace("0",""))
        represent.append("("+"L"+","+"0"+str(lit)+")")
        literal = spli[1].replace("'","").replace("\n","").replace("=","")
        Literals[literal] = c
      else:
        sym_count = sym_count + 1
        represent.append(REG[spli[0]].replace("0",""))
        represent.append("(s,"+"0"+str(sym_count)+")")
        Symbols[spli[1].replace("\n","")] = c
      table.append(represent)
      represent = []
    else:
      sym_count = sym_count + 1
      Symbols[spli[0].replace("\n","")] = c
```

```
            represent.append("-")
            if words[1].replace("\n","") in Symbols.keys():
                for k in Symbols.keys():
                    a.append(k)
                #(Symbols.keys()).index("FIRST")
                represent.append("(S,"+"0"+ str(a.index(spli[0].replace("\n","")) + 1) + ")")
                a = []
            else:
                represent.append("(S,"+"0"+str(sym_count)+")")
            table.append(represent)
            represent = []
    if(words[1] in DS):
        represent.append(c)
        represent.append("(DL,"+ DS[words[1]] +")")
        represent.append("-")
        represent.append("(c,"+ words[2].replace("'","").replace("\n","")+")")
        if words[0] in Symbols.keys():
            Symbols[words[0]] = c
        table.append(represent)
        represent = []
    c = c + 1
fp = open("two_pass.txt","r")
line = fp.readline()
while(line):
    if(line == "\n"):
        line = fp.readline()
        continue
    code.append(line)
    ls = line.split(" ")
    if('START' in line.split(" ")):
        sym_count = int(c)
        lit = int(c)
        represent.append("-")
        word = line.split(" ")
        represent.append("(AD,"+AD[word[0].replace("\n","")]+ ")")
        represent.append("-")
        represent.append("(c,"+word[1].replace("\n","")+")")
        c = int(word[1].replace("\n",""))
        table.append(represent)
        represent=[]
        line = fp.readline()
        continue
    if('BC' in ls):
        represent.append(c)
        c = c + 1
        represent.append("(IS,"+INS_CODE['BC']+")")
        represent.append("-")
        represent.append("(S,0" +str(pt) +")")
        table.append(represent)
        represent = []
        line = fp.readline()
        continue
    if(len(ls) == 3 and "" not in list(ls[2])):
        Symbols[ls[0]] = c
        sym_count = sym_count + 1
        pt  = sym_count
        line = line.replace(ls[0],"")
        line = line[1:]
    if(line.replace("\n","") == 'END'):
        represent.append(c)
        represent.append("(AD,"+AD['END']+")")
        represent.append("-")
        represent.append("-")
        table.append(represent)
        represent = []
        line = fp.readline()
        continue
    assembler(line)
    line = fp.readline()
fp.close()
x = PrettyTable()
x = PrettyTable(["Symbol", "Address"])
y = PrettyTable()
y = PrettyTable(["Literal", "Address"])
z = PrettyTable()
z = PrettyTable(["Code","Address", "IC1","IC2","IC3"])
w = PrettyTable()
w = PrettyTable(["Address","MC1","MC2","MC3"])
b =  PrettyTable()
b = PrettyTable(["Assembly instruction","Code","Class"])
for k,v in Symbols.items():
    x.add_row([k,v])
for k,v in Literals.items():
    y.add_row([k,v])
for row in table:
    z.add_row([code[j],row[0],row[1],row[2],row[3]])
    j = j + 1
for k,v in INS_CODE.items():
    b.add_row([k,v,"IS"])
for k,v in DS.items():
    b.add_row([k,v,"DS"])
for k,v in AD.items():
    b.add_row([k,v,"AD"])
print("Mnemonic")
print(b)
#pass1
print("\r")
print("pass1")
print(z)
#Symbol table
print("\r")
print("Symbol Table")
print(x)
print("\r")
#print(y)
print('pass 2')
for i in range(1,len(table)-1):
   temp = table[i][1].split(",")
   temp_1 = code[i].split(" ")
   addr_1 = table[i][3].split(",")
   if(len(temp_1) == 2):
      temp_2 = temp_1[1].split(",")
      if(len(temp_2) == 2):
         if(temp_2[1].replace("\n","") in Symbols.keys()):
            addr = Symbols[temp_2[1].replace("\n","")]
         else:
            adde = addr_1[1].replace(")","")
   else:
      if(temp_1[1].replace("\n","") in Symbols.keys()):
         addr = Symbols[temp_1[1].replace("\n","")]
      else:
         addr = "0"+addr_1[1].replace(")","")
   w.add_row([table[i][0] , temp[1].replace(")","").replace("0",""),
table[i][2],str(addr).replace("00","0")])
print(w)
```

Output:

```
Mnemonic
+----------------------+------+-------+
| Assembly instruction | Code | Class |
+----------------------+------+-------+
|        STOP          |  00  |   IS  |
|        ADD           |  01  |   IS  |
|        SUB           |  02  |   IS  |
|        MULT          |  03  |   IS  |
|       MOVER          |  04  |   IS  |
|       MOVEM          |  05  |   IS  |
|        COMP          |  06  |   IS  |
|         BC           |  07  |   IS  |
|        DIV           |  08  |   IS  |
|        READ          |  09  |   IS  |
|       PRINT          |  10  |   IS  |
|         LE           |  11  |   IS  |
|         DC           |  01  |   DS  |
|         DS           |  02  |   DS  |
|       START          |  01  |   AD  |
|        END           |  02  |   AD  |
|       ORIGIN         |  03  |   AD  |
+----------------------+------+-------+
pass1
+----------------------+---------+---------+-----+---------+
|         Code         | Address |   IC1   | IC2 |   IC3   |
+----------------------+---------+---------+-----+---------+
|      START 200       |    -    | (AD,01) |  -  | (c,200) |
|                      |         |         |     |         |
|   MOVER AREG,FIRST   |   200   | (IS,04) |  1  |  (s,01) |
|                      |         |         |     |         |
|   ADD AREG,SECOND    |   201   | (IS,01) |  1  |  (s,02) |
|                      |         |         |     |         |
|  MOVEM AREG,RESULT   |   202   | (IS,05) |  1  |  (s,03) |
|                      |         |         |     |         |
| LABEL MULT AREG,THREE |   203   | (IS,03) |  1  |  (s,05) |
|                      |         |         |     |         |
|     BC LE,LABEL      |   204   | (IS,07) |  -  |  (S,04) |
|                      |         |         |     |         |
|     PRINT RESULT     |   205   | (IS,10) |  -  |  (S,03) |
|                      |         |         |     |         |
|    RESULT DS '1'     |   206   | (DL,02) |  -  |  (c,1)  |
|                      |         |         |     |         |
|     FIRST DC '5'     |   207   | (DL,01) |  -  |  (c,5)  |
|                      |         |         |     |         |
|    SECOND DC '7'     |   208   | (DL,01) |  -  |  (c,7)  |
|                      |         |         |     |         |
|     THREE DC '1'     |   209   | (DL,01) |  -  |  (c,1)  |
|                      |         |         |     |         |
|         END          |   210   | (AD,02) |  -  |    -    |
+----------------------+---------+---------+-----+---------+
Symbol Table
+--------+---------+
| Symbol | Address |
+--------+---------+
| FIRST  |   207   |
| SECOND |   208   |
| RESULT |   206   |
| LABEL  |   203   |
| THREE  |   209   |
+--------+---------+
pass 2
+---------+-----+-----+-----+
| Address | MC1 | MC2 | MC3 |
+---------+-----+-----+-----+
|   200   |  4  |  1  | 207 |
|   201   |  1  |  1  | 208 |
|   202   |  5  |  1  | 206 |
|   203   |  3  |  1  |  05 |
|   204   |  7  |  -  | 203 |
|   205   |  1  |  -  | 203 |
|   206   |  2  |  -  |  01 |
|   207   |  1  |  -  |  05 |
|   208   |  1  |  -  |  07 |
|   209   |  1  |  -  |  01 |
+---------+-----+-----+-----+
```

Conclusion: Hence, we studied and implemented two pass assembler.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: **System Programming & Compiler Construction**

Experiment No: 9

Title of Experiment: Implementation of single pass Macro Processor

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

Aim: Implementation of single pass Macro Processor

Theory:

A macro in computer science is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output sequence according to a defined procedure. The mapping process that instantiates a macro use into a specific sequence is known as macro expansion. Macros are pre-processed which means that all the macros would be processed before your program compiles. In macros, no type checking(incompatible operand, etc.) is done and thus use of micros can lead to errors/side-effects in some cases. However, this is not the case with functions. Also, macros do not check for compilation error (if any).

Writing a macro is another way of ensuring modular programming in assembly language.A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.

In NASM, macros are defined with %macro and %endmacro directives.The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition –

%macro macro_name number_of_params

<macro body>

%endmacro

Where, number_of_params specifies the number parameters, macro_name specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

Advantages:

The speed of the execution of the program is the major advantage of using a macro.

It saves a lot of time that is spent by the compiler for invoking / calling the functions.

It reduces the length of the program.

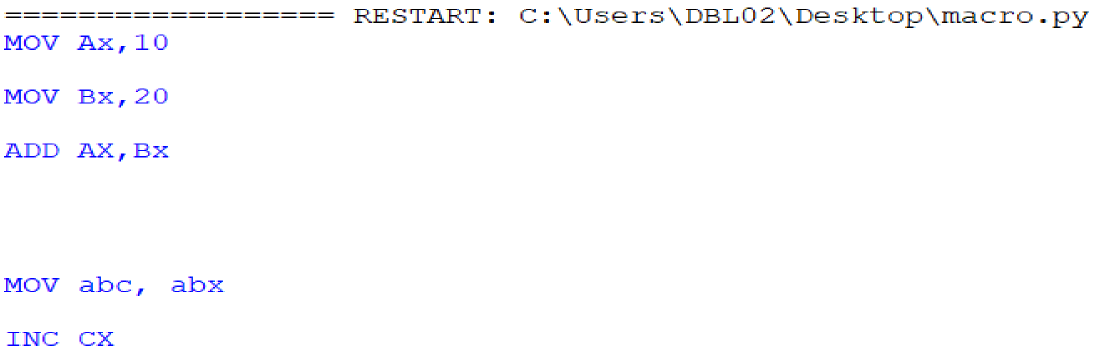
SOURCE CODE:

```
fp =  open("macro.txt","r")

flg = 0

str = ""

str1 = ""

arg = ""

args = ""
```

```
i = 0
c = 0
name = ""
macargs = ""
saved = ""
dict = {}
def assembly(line):
    global flg,str,str1,name,arg,args,macargs,saved
    words = line.split()
    for word in words:
        if(word == "MACRO"):
            flg = 1
            saved = line
            name = words[1]
            for i in range(2,len(words)):
                arg = arg + words[i]
            continue
        if(word == name and flg == 1):
            for i in range(2,len(words)):
                macargs = macargs + words[i]+" "
            continue
        if(flg == 1 and word == "ENDMACRO"):
            flg = 0
            continue
        if(word == name and flg == 0):
            for i in range(1,len(words)):
                args = args + words[i]+" "
                line = line.replace(words[i],"")
        str1 = str1.replace("@","\n")
        if(words[len(words)-1] == word ):
            if(flg == 1):
                str1 = str1 + line + "@"
            else:
                str = str + line + "\n"
```

```
line = fp.readline()

while(line):
   assembly(line)
   line = fp.readline()
print(str.replace(name,""))
str1 = str1.replace(saved,"")
#print(arg)
#print(args)
temp = macargs.split(" ")
temp.pop()
args = args.split(" ")
for a in temp:
   dict[a] = args[i]
   i =i + 1
#print(dict)
for k,v in dict.items():
   str1 = str1.replace(k,v)
print(str1)
#print(dict)
```

OUTPUT:



```
================== RESTART: C:\Users\DBL02\Desktop\macro.py
MOV Ax,10

MOV Bx,20

ADD AX,Bx



MOV abc, abx

INC CX
```

Conclusion: We have studied and implemented Single Pass Macro Processor.

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL
COMPUTER ENGINEERING DEPARTMENT
A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: System Programming & Compiler Construction

Experiment No: 10

Title of Experiment: Creating own library in python

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	8	
2	Oral	3	
3	Timely Submission	2	
4	Neatness	2	
	Total	15	

Signature of Subject Teacher

Aim: Creating own library in python

Theory:

Packages are a way of structuring many packages and modules which help in a well organized hierarchy of data set, making the directories and modules easy to access. Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.

Creating and Exploring Packages

To tell Python that a particular directory is a package, we create a file named __init_.py inside it and then it is considered as a package and we may create other modules and sub packages within it. This __init_.py file can be left blank or can be coded with the initialization code for the package.

To create a package in Python, we need to follow these three simple steps:

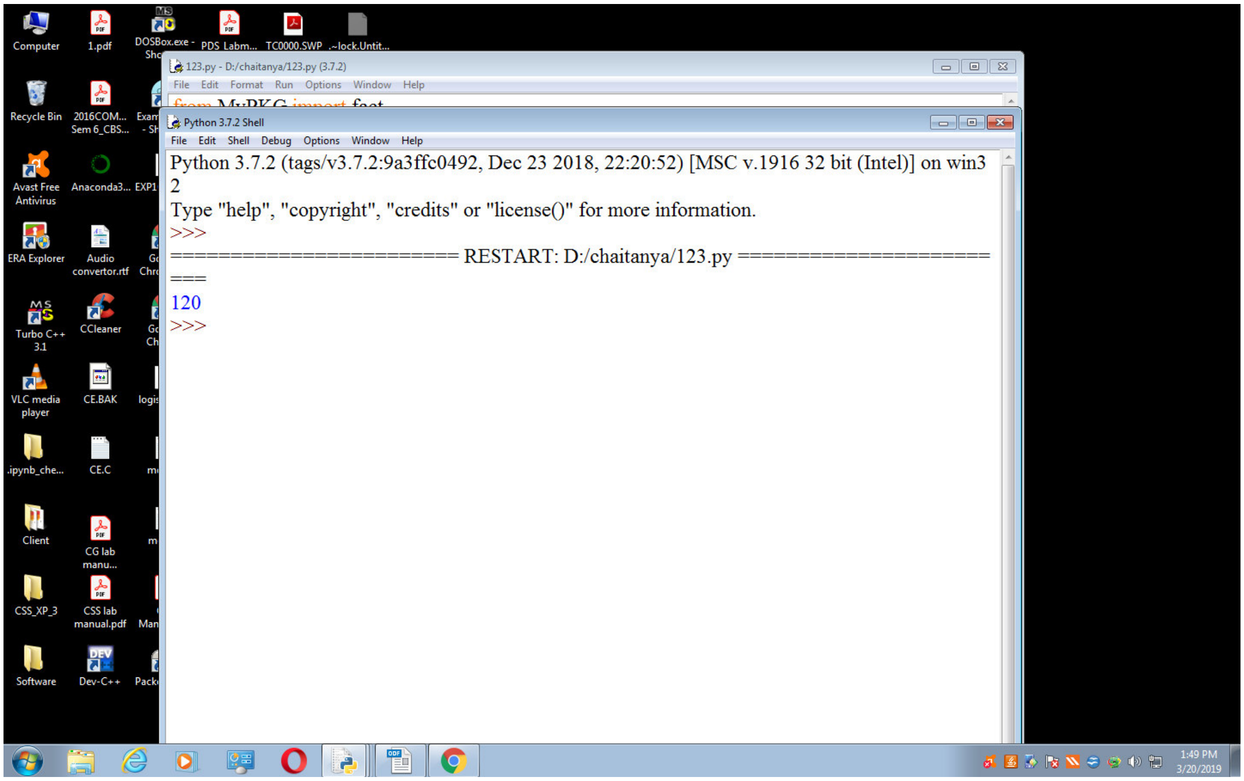
1. First, we create a directory and give it a package name, preferably related to its Operation.
2. Then we put the classes and the required functions in it.
3. Finally we create an __init_.py file inside the directory, to let Python know that the Directory is a package.

SOURCE CODE:

```
#Fact.py
def fact
(n): fact=1
while(n>0):
fact = fact *
n n -= 1
return fact
```

```
#123.py
fromMyPKG import
fact print
(fact.fact(5))
```

OUTPUT:



```
from MyPKG import fact
```

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win3
2
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: D:/chaitanya/123.py ======================
===
120
>>>
```

Conclusion: Hence we created our own library.