



B.S. Abdur Rahman

# Crescent

Institute of Science & Technology

Deemed to be University u/s 3 of the UGC Act, 1956

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**CSC 4104  
COMPILER LAB**

**NAME: M MOHAMED IMRAN**

**RRN: 180071601108**

**SEMESTER: VII SEMESTER**

**BRANCH & SECTION: B.TECH – CSE & B**



B.S. Abdur Rahman

# Crescent

Institute of Science & Technology

Deemed to be University u/s 3 of the UGC Act, 1956

## **BONAFIDE CERTIFICATE**

Certified that this is the Bonafide record of the work done by  
**M Mohamed Imran** Register No. **180071601108** of VI Semester  
**B. TECH-COMPUTER SCIENCE AND ENGINEERING** in the **CSC4104**  
**COMPILER LAB** during the year 2020 - 2021.

Course Faculty

Head of the Department

Date: 01.10.2021

Submitted for the Practical Examination held on 01.10.2021

Examiner



B.S. Abdur Rahman

**Crescent**

Institute of Science & Technology

Deemed to be University u/s 3 of the UGC Act, 1956

**Name of the Student** : MOHAMED IMRAN M

**RRN** : 180071601108

**Department** : Computer Science & Engineering.

**Semester from** : July 2021 – December 2021

**Class &Section** : IV year – VII Semester – B

**Course Code** : CSC4104

**Course Name** : Compiler Lab

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**INDEX**

S.NO	DATE	TITLE	PAGE NO.
1	01.09.21	Design a lexical analyzer for a given High Level Language. Ignore redundant	5 – 7
2	07.09.21	Study of compiler construction tools	8 - 15
3	08.09.21	Implement a lexical analyzer	16 – 21
4		Implementation using LEX program	
4.1	14.09.21	Write a LEX program to count the number of identifiers and constants in a given file.	22 – 24
4.2	15.09.21	Write a LEX program to validate the mobile number allotted to India.	25 – 27
4.3		Write a LEX program to check valid email id and date	
4.3.1	15.09.21	LEX program to check valid email id	28 – 30
4.3.2	15.09.21	LEX program to check valid date	31 – 33
4.4	15.09.21	Write a LEX program to count of words starting with letter 'a'.	34 – 36
4.5	15.09.21	Write a LEX program to separate the tokens from a given file.	37 – 39
5	17.09.21	Implement a scientific calculator	40 - 44
6		Implementing parser	
6.1	22.09.21	Design and implement Top-Down parsing techniques and	45 – 52
6.2	24.09.21	Design and implement Bottom-up parsing techniques.	53 – 57
7	24.09.21	Generate abstract syntax tree and intermediate code for the given language.	58 – 61
8	28.09.21	Implement code optimization phase of the compiler.	62 – 69

**Exp. No: 1**

**Date: 01.09.21**

**Design a Lexical Analyzer for a given high level language.  
Ignore redundant Space, tabs and new lines.**

**AIM:**

To Design a Lexical Analyzer for a given high level language. Ignore redundant Space, tabs and new lines.

**ALGORITHM:**

**STEP-1:** Identify the valid tokens i.e., identifiers, datatypes, integers, endstatement, operators etc., in the given language according to entered expression.

**STEP-2:** Write a program for lexical analyzer to recognize all the valid tokens in the input program.

**STEP-3:** Import necessary packages.

**STEP-4:** Take the input from user and split it using split() function and save it to a list.

**STEP-5:** Define list of operators, data types, integers and endstatement.

**STEP-6:** Print the tokens.

**STEP-7:** Finally output is displayed in the screen.

## PROGRAM:

```
import re

li=[]

value = input("Enter your value: ")

message = value.split()

print('VALUE ENTERED:\n',value)


datatypes=['int','float','complex','list','tuple','range','dict','set',
'frozenset','bool','bytes','bytearray','memoryview','str']

operators=['+','-','*','/','//','%','**','>','<','==','!=','>=','<=']

endstatement=[';']

integer=[0,1,2,3,4,5,6,7,8,9]

for word in message:
    if word.lower() in datatypes:
        li.append(['DATATYPE', word])
    elif word.lower() in operators:
        li.append(['OPERATOR', word])
    elif word.lower() in endstatement:
        li.append(['ENDSTATEMENT', word])
    elif word.lower() in integer:
        li.append(['INTEGER', word])
    elif re.match("[a-z]", word.lower()):
        li.append(['IDENTIFIER', word])
    else:
        li.append(['INTEGER', word])

print(li)
```

## OUTPUT:



```
[1] import re

[2] ll=[]
value = input("Enter your value: ")
message = value.split()
print("VALUE ENTERED:\n",value)

Enter your value: ( x * y ) / 5 = 10 ;
VALUE ENTERED:
( x * y ) / 5 = 10 ;

[3] datatypes=['int','float','complex','list','tuple','range','dict','set',
'frozenset','bool','bytes','bytearray','memoryview','str']
operators=['+','-','**','/','//','%','**','>','<','==','!=','>=','<=']
endstatement=[';']
integer=[0,1,2,3,4,5,6,7,8,9]

[4] for word in message:
    if word.lower() in datatypes:
        ll.append(['DATATYPE', word])
    elif word.lower() in operators:
        ll.append(['OPERATOR', word])
    elif word.lower() in endstatement:
        ll.append(['ENDSTATEMENT', word])
    elif word.lower() in integer:
        ll.append(['INTEGER', word])
    elif re.match("[a-z]", word.lower()):
        ll.append(['IDENTIFIER', word])
    else:
        ll.append(['INTEGER', word])
print(ll)

[[['INTEGER', '('], ['IDENTIFIER', 'x'], ['OPERATOR', '*'], ['IDENTIFIER', 'y'], ['INTEGER', ')'], ['OPERATOR', '/'], ['INTEGER', '5'], ['INTEGER', '='], ['INTEGER', '10'], ['ENDSTATEMENT', ';']]]
```

## RESULT:

Hence a lexical analyser has been successfully created, executed and output displayed in google colaboratory using Python Programming Language.

**Exp. No: 2**

**Date: 07.09.21**

## **Study of Flex Compiler Construction Tool**

### **1. Flex (Fast Lexical Analyzer Generator):**

FLEX 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.

In order for Flex to recognize patterns in text, the pattern must be described by a regular expression. The input to Flex is a machine readable

set of regular expressions. The input is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, `lex.yy.c`, which defines a routine `yylex()`. This file is compiled and linked with the `-lfl` library to produce an executable.

When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

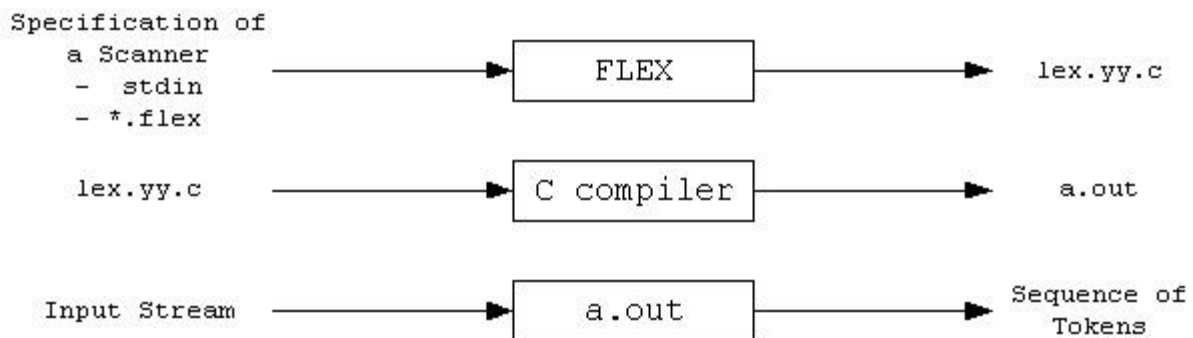
flex is a tool for generating scanners: programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, `'lex.yy.c'`, which defines a routine `'yylex()'`. This file is compiled and



linked with the `-lfl` library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

### How to use FLEX?

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. In stead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language (e.g. Simple), write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



First, FLEX reads a specification of a scanner either from an input file `*.lex`, or from standard input, and it generates as output a C source file `lex.yy.c`. Then, `lex.yy.c` is compiled and linked with the `-lfl` library to produce an executable `a.out`. Finally, `a.out` analyzes its input stream and transforms it into a sequence of tokens.

`*.lex` is in the form of pairs of regular expressions and C code.  
(`sample1.lex`, `sample2.lex`)

`lex.yy.c` defines a routine `yylex()` that uses the specification to recognize tokens.

`a.out` is actually the scanner!

## Flex Examples:

The following Flex input specifies a scanner which whenever it encounters the string “username” will replace it with the user’s login name:

```
%%  
username printf( "%s", getlogin() );
```

By default, any text not matched by a Flex scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of “username” expanded. In this input, there is just one rule. “username” is the pattern and the “printf” is the action. The “%%” marks the beginning of the rules.

Here’s another simple example:

```
int num_lines = 0, num_chars = 0;  
%%  
\n ++num_lines; ++num_chars;  
. ++num_chars;  
%%  
main()  
{  
  yylex();  
  printf( "# of lines = %d, # of chars = %d\n",  
    num_lines, num_chars );  
}
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, num lines and num chars, which are accessible both inside yylex() and in the main() routine declared after the second “%%”. There are two rules, one which matches a newline (“\n”) and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the “.” regular expression).

## The Generated Scanner

The output of Lex/Flex is the file `lex.yy.c`, which contains the scanning routine `yylex()`, a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, `yylex()` is declared as follows:

```
int yylex()
{
... various definitions and the actions in here ...
}
```

(If your environment supports function prototypes, then it will be “`int yylex(void )`”.) This definition may be changed by redefining the “YY DECL” macro. For example, you could use:

```
#undef YY_DECL
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give the scanning routine the name `lexscan`, returning a float, and taking two floats as arguments. Note that if you give arguments to the scanning routine using a K&R-style/non-prototyped function declaration, you must terminate the definition with a semi-colon (;).

Whenever `yylex()` is called, it scans tokens from the global input file `yypin` (which defaults to `stdin`). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a return statement. In the former case, when called again the scanner will immediately return unless `yyrestart()` is called to point `yypin` at the new input file. ( `yyrestart()` takes one argument, a `FILE *` pointer.) In the latter case (i.e., when an action executes a return), the scanner may then be called again and it will resume scanning where it left off.

## 2. YACC (Yet Another Compiler Compiler):

In order for Yacc/Bison to parse a language, the language must be described by a context-free grammar. The most common formal system for presenting such rules for humans to read is Backus-Naur Form or “BNF”, which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Yacc/Bison is essentially machine-readable BNF.

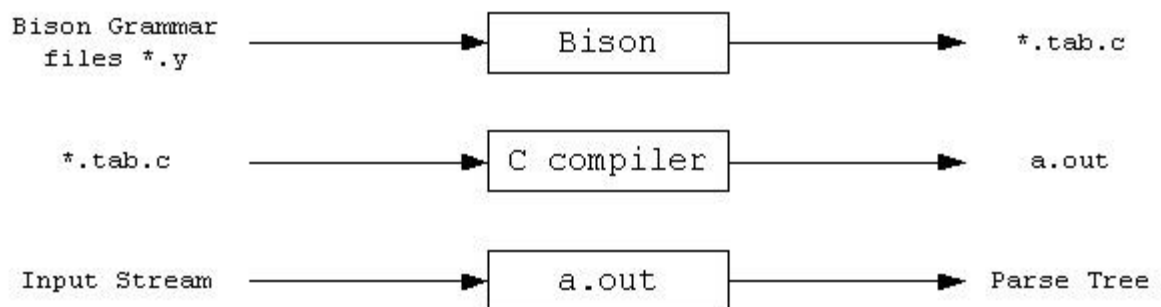
Not all context-free languages can be handled by Yacc/Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1).

Yacc is a parser generator, specifically a tool to generate LALR parsers. Essentially a parser groups tokens (like the ones generated by Lex) into logical structures. Again, all you need to have is a grammar that describes the rules to follow. For example, you can generate a parser that can take a IDENTIFIER (e.g., a variable), a EQUAL token, and a INT token and understand that together they form an assignment. A parser deals with much more complicated problems than that of a lexer.

Since Lex is used to generate lexers and Yacc to generate parsers, they were complementary and often used together. They were simply the best software available in their respective niches.

## How to use YACC?

Bison is a general-purpose parser generator that converts a grammar description (Bison Grammar Files) for an LALR(1) context-free grammar into a C program to parse that grammar. The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.



## Steps to use Yacc:

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.

## Yacc Examples:

The following is a Yacc/Bison input file which defines a reverse Polish notation calculator. The file created by Yacc/Bison simulates the calculator. The details of the example are explained in later sections.

```
/* Reverse Polish notation calculator. */
% {
#define YYSTYPE double
#include <math.h>
% }
%token NUM
%% /* Grammar rules and actions follow */
input : /* empty */
| input line
;
line : '\n'
| exp '\n' { printf ("\t%.10g\n", $1); }
;
exp : NUM { $$ = $1; }
| exp exp '+' { $$ = $1 + $2; }
| exp exp '-' { $$ = $1 - $2; }
| exp exp '*' { $$ = $1 * $2; }
| exp exp '/' { $$ = $1 / $2; }
/* Exponentiation */
| exp exp '^' { $$ = pow ($1, $2); }
/* Unary minus */
| exp 'n' { $$ = -$1; }
;
```

## Parser Generator:

It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.

The Antlr parser generator tool converts a grammar source file like Sum.g4 into Java classes that implement a parser. To do that, you need to go to a command prompt (Terminal or Command Prompt) and run a command like this:

```
java -jar antlr.jar Sum.g4
```

You'll need to make sure you're in the right folder (where Sum.g4 is) and that antlr.jar is in there too (or refer to it where it is in your project folder structure – it may be a relative path like ../../lib/antlr.jar).

Assuming you don't have any syntax errors in your grammar file, the parser generator will produce new Java source files in the current folder. The generated code is divided into several cooperating modules:

the lexer takes a stream of characters as input, and produces a stream of terminals as output, like NUMBER, +, and (. For Sum.g4, the generated lexer is called SumLexer.java.

the parser takes the stream of terminals produced by the lexer and produces a parse tree. The generated parser is called SumParser.java.

the tree walker lets you write code that walks over the parse tree produced by the parser, as explained below. The generated tree walker files are the interface SumListener.java, and an empty implementation of the interface, SumBaseListener.java.

Antlr also generates two text files, Sum.tokens and SumLexer.tokens, that list the terminals that Antlr found in your grammar. These aren't needed for a simple parser, but they're needed when you include grammars inside other grammars.

**Exp. No: 3**

**Date: 08.09.21**

## **Implement a lexical analyzer**

### **AIM:**

To implement a lexical analyzer that contains the following features

1. Identify and display the number of characters, words and lines in the input string.
2. Identify the tokens and display them.
3. Identify the total number of tokens.
4. Find the identifiers and validate them (display the correctly and wrongly written Identifiers).
5. Identify the keywords (for any three languages).
6. Identify the comment line and display it (for three languages).
7. Identify and display the numbers.
8. Identify and display the number of white spaces.
9. Identify the operators and display the corresponding operations.
10. Identify the programming language of the input (use identifiers to check).

### **ALGORITHM:**

**STEP-1:** Identify the valid tokens i.e., identifiers, datatypes, integers, endstatement, operators etc., in the given language according to entered expression.

**STEP-2:** Write a program for lexical analyzer to recognize all the valid tokens in the input program.

**STEP-3:** Import necessary packages.

**STEP-4:** Take the input from C program file.

**STEP-5:** Define list of operators, data types, integers, endstatement, keywords and identifiers.

**STEP-6:** Print the total number of tokens, white space comment line language, keyword language and programming language of input.

**STEP-7:** Finally output is displayed in the screen.



## PROGRAM:

### Code for input file:

```
import java.io // header file

def name: /* function name to uniquely identify the function */

int main()

{

    int D ;

    char b ;

    c = 100 ;

    c += 1 ;

    printf ("The value of c is %d ",c);

    print ("Hello")

    cout << ("Welcome");

    return 0 ;

}
```

### Code for Lexical analyzer:

```
import re

f = open('Input.c','r')

token_length = 0

operators = { '=': 'Assignment Operator', '+': 'Addition Operator', '+=': 'Addition assignment', '-': 'Subtraction assignment', '-=': 'Subtraction assignment', '/' : 'Division Operator', '*': 'Multiplication Operator', '++': 'increment Operator', '--': 'Decrement Operator' }

optr_keys = operators.keys()

comments = { 'r//' : 'Single Line Comment \n comment line language : Java', 'r/*' : 'Multiline Comment Start', 'r*/' : 'Multiline Comment End \n comment line language: C', 'r/**' : 'Empty Multiline comment', 'r--' : 'Proceed with the text of the comment \n comment line language: SQL' }

comment_keys = comments.keys()

header = { '.h': 'header file' }

header_keys = header.keys()

sp_header_files = { '<stdio.h>': 'Standard Input Output Header \n Language: C', '<string.h>': 'String Manipulation Library \n Language: C', '<iostream.h>': 'standard input-output stream \n Language: C++', 'java.io.*': 'Language: java' }

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 \n Keyword language: C','native': 'Keyword language: Java', 'def' : 'Keyword language: Python'}
```

```
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 \n Language: C', 'print':'Print the statement \n Language: Python', 'cout':'Print the statement \n Language: C++'}
```

```
builtin_functions_keys = builtin_functions.keys()
```

```
non_identifiers = ['_', '-', '+', '/', '*', '^', '~', '!', '@', '#', '$', '%', '&', '*', '(', ')', '=', '|', '"', "'", ':', ';', '{', '}', '[', ']', '<', '>', '?', '/', '/*', '*/']
```

```
numerals = ['0','1','2','3','4','5','6','7','8','9','10']  
loats = ['0.0','1.1','2.2','3.3','4.4','5.5','6.6','7.7','8.8','9.9']
```

```
dataFlag = False
```

```
i = f.read()
```

```
space = 0
```

```
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 token == ":
```

```
            space +=1
```

```
        if '\\r' in token:
```

```
            position = token.find('\\r')
```

```
            token=token[:position]
```

```
        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 token in builtin_functions_keys:
    print("Built in function is: ", builtin_functions[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):
    if(token.isupper()):
        print(token, "is not a valid Identifier instead use ->",token.lower())
    else:
        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())

elif 'import' in token:
    match = re.search(r'import\w+', token)
    print("Header", token)
    print('Language: Java')

if token in floats:
    print(token, "It is not a valid Identifier")

if token in numerals:
    print (token,type(int(token)))

dataFlag = False

token_length += len(token)

print("_____")

f.close()

print("Total number of Line: ",count)

print("Total number of tokens: ",token_length)

print("Total number of White space: ",space)

```

# OUTPUT:

```
f.close()
print("Total number of Lines: ",count)
print("Total number of tokens: ",token_length)
print("Total number of white space: ",space)

Line # 1
import java.io // header file
Tokens are ['import', 'java.io', '/', '/', 'header', 'file']
Line # 1 properties

Header: import.java.io
Language: Java
Comment Type: Single Line Comment
comment line language: Java

Line # 2
def name: /* function name to uniquely identify the function */
Tokens are ['def', 'name:', '/', '/', 'function', 'name', 'to', 'uniquely', 'identify', 'the', 'function', '/']
Line # 2 properties

Keyword language: Python
Comment Type: Multiline Comment Start
Comment Type: Multiline Comment End
comment line language: C

Line # 3
int main()
Tokens are ['int', 'main()']
Line # 3 properties

type is: Integer
Function named main()

Line # 4
{
Tokens are ['{']
Line # 4 properties

Blocked Statement Body Open

Line # 5
int D ;
Tokens are ['int', 'D', ';']
Line # 5 properties
```

```
Line # 5
int D ;
Tokens are ['int', 'D', ';']
Line # 5 properties

type is: Integer
D is not a valid identifier instead use -> d
Delimiter terminator symbol semicolon (;)

Line # 6
char b ;
Tokens are ['char', 'b', ';']
Line # 6 properties

type is: Character
Identifier: b
Delimiter terminator symbol semicolon (;)

Line # 7
c = 100 ;
Tokens are ['c', '=', '100', ';']
Line # 7 properties

Operator is: Assignment Operator
Delimiter terminator symbol semicolon (;)

Line # 8
c += 1 ;
Tokens are ['c', '+', '=', '1', ';']
Line # 8 properties

Operator is: Addition assignment
1 <class 'int'>
Delimiter terminator symbol semicolon (;)

Line # 9
printf("The value of c is %d ",c);
Tokens are ['printf', '(', 'The', 'value', 'of', 'c', 'is', '%d', ' ', 'c', ')', ';']
Line # 9 properties
```



**Exp. No: 4**

## **Implement a lexical analyser using Lex programs**

**Exp. No: 4.1**

**Date: 14.09.21**

### **LEX program to count the number of identifiers and constants**

#### **AIM:**

To write a LEX program to count the number of identifiers and constants in a given file.

#### **ALGORITHM:**

**Step 1:** Start

**Step 2:** Import required variables and functions

**Step 3:** Create a file “iden.l” and “input.txt”

**Step 4:** If the given buffer is “constant”, how many number of times constants exists and show the count as number of constants

**Step 5:** Else if the given buffer is not a keyword, take it as an identifier and count the number of identifiers similarly and show it as number of identifiers

**Step 6:** Finally output will display in the screen.

**Step 7:** Stop

## PROGRAM:

```
% {  
int count=0;  
int ccount=0;  
% }  
op [+-*/]  
letter [a-zA-Z]  
digitt [0-9]  
id {letter}*(({letter}{digitt})+  
notid ({digitt}|{digitt}{letter})+  
%%  
[\t\n])+  
(('int')|('float')|('char')|('case')|('default')|('if')|('for')|('printf')|('scanf')) {printf("%s is a  
keyword\n", yytext);}   
{id} {printf("%s is an identifier\n", yytext); count++;}  
{notid} {printf("%s is an constant\n", yytext); ccount++;}  
%%  
int yywrap(void){ }  
int main()  
{  
FILE *fp;  
char file[10];  
printf("\nEnter the filename: ");  
scanf("%s", file);  
fp=fopen(file,"r");  
yyin=fp;  
yylex();  
printf("Total identifiers are: %d\n", count);  
printf("Total constants are: %d\n", ccount);  
return 0;  
}
```

The screenshot shows a Windows File Explorer window with the address bar displaying the path: `C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\Iden1 - E08Plus`. The left sidebar shows the 'Compiler lab' folder expanded, with subfolders: `exp`, `date`, `Iden`, `mail`, `mobile`, and `re`. The 'Iden' folder is selected. The main pane shows the contents of the 'Iden' folder, which includes a file named `input.txt`. A Command Prompt window is open in the foreground, showing the execution of a C program. The program's output is as follows:

```

Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>cd onedrive\documents\text\compiler lab\exp\Iden
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\Iden>lex iden.1
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\Iden>gcc lex.yy.c
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\Iden>a.exe

Enter the filenames: input.txt
int is an identifier
float is an identifier
78g is a constant
90H is a constant
p is an identifier
a is an identifier
d is an identifier
are is an identifier
ase is an identifier
default is an identifier
printf is an identifier
scanf is an identifier
Total identifiers are: 10
Total constants are: 2

C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\Iden>
  
```

Hence the code for program to count the number of identifiers and constants in a given file is implemented using lex program.



**Exp. No: 4.2**

**Date: 15.09.21**

**LEX program to validate the mobile  
number allotted to India**

**AIM:**

To Write a LEX program to validate the mobile number allotted to India.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to check mobile number validity which has only numbers for Indian region

**Step 4:** in main(), input the mobile number and call yylex().

**Step 5:** Display the output if number is valid or not.

**Step 6:** Stop.

## PROGRAM:

```
%%
```

```
[\n] {printf("\nEnter a valid mobile Number: ");}
```

```
[9][1][6-9][0-9]{9} {printf("\nThe mobile number entered is an Indian mobile number and valid\n");}
```

```
.* {printf("\nThe mobile number entered is invalid\n");}
```

```
%%
```

```
int yywrap(void){}
```

```
int main()
```

```
{
```

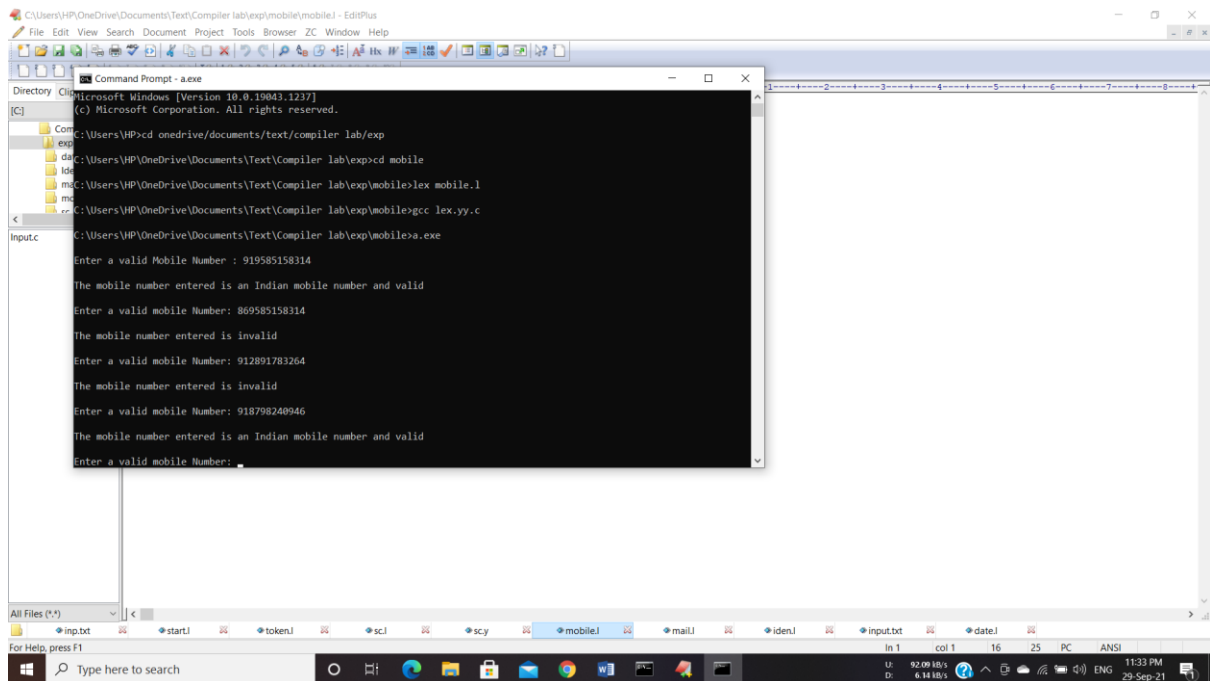
```
    printf("\nEnter a valid Mobile Number : ");
```

```
    yylex();
```

```
    return 0;
```

```
}
```

## OUTPUT:



```
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mobile - EditPlus
File Edit View Search Document Project Tools Browser ZC Window Help

Command Prompt - a.exe
Directory: C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>cd onedrive\documents\text\compiler lab\exp
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp>cd mobile
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mobile>lex mobile.l
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mobile>gcc lex.yy.c
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mobile>a.exe

Enter a valid Mobile Number : 919585158314
The mobile number entered is an Indian mobile number and valid
Enter a valid mobile Number: 869585158314
The mobile number entered is invalid
Enter a valid mobile Number: 912891783264
The mobile number entered is invalid
Enter a valid mobile Number: 918798240946
The mobile number entered is an Indian mobile number and valid
Enter a valid mobile Number: .
```

## RESULT:

Hence the code for program to validate the mobile number in India is implemented using lex program.

**Exp. No: 4.3**

## **LEX program to check valid email id and date**

**Exp. No: 4.3.1**

**Date: 15.09.21**

### **LEX program to check valid email id**

**AIM:**

To Write a LEX program to check valid email id.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to check valid date validity which has alphanumeric@email.domain\_name.

**Step 4:** in main(), input the email ID and call yylex().

**Step 5:** Display the output if Check Valid email ID is valid or not.

**Step 6:** Stop.

## PROGRAM:

%%

```
[\n] {printf("\nEnter a valid Email ID: ");}
```

```
[a-z0-9+]*(@[a-z]+)(.[a-z]+)+ {printf("\nThe Email ID entered is valid\n");}
```

```
.* {printf("\nThe Email ID entered is invalid\n");}
```

%%

```
int yywrap(void){ }
```

```
int main()
```

```
{
```

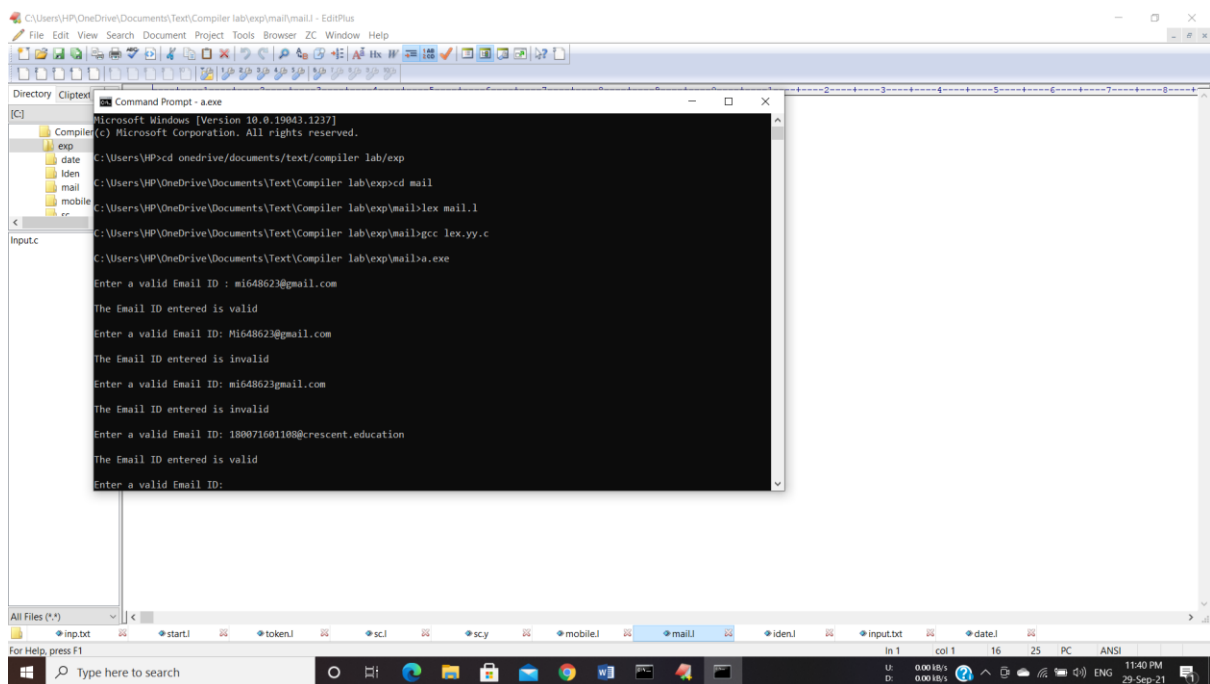
```
    printf("\nEnter a valid Email ID : ");
```

```
    yylex();
```

```
    return 0;
```

```
}
```

## OUTPUT:



```
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mail\mail1 - EditPlus
File Edit View Search Document Project Tools Browser ZC Window Help
Directory ClipText
[C:]
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.
C:\Users\HP>cd onedrive\documents\text\compiler lab\exp
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp>cd mail
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mail>lex mail.1
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mail>gcc lex.yy.c
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\mail>a.exe
Enter a valid Email ID : mi648623@gmail.com
The Email ID entered is valid
Enter a valid Email ID: M1648623@gmail.com
The Email ID entered is invalid
Enter a valid Email ID: mi648623@gmail.com
The Email ID entered is invalid
Enter a valid Email ID: 180071601100@crescent.education
The Email ID entered is valid
Enter a valid Email ID:
```

## RESULT:

Hence the code for program to check valid EMAIL ID is implemented using lex program.

**Exp. No: 4.3.2**

**Date: 15.09.21**

**LEX program to check valid date**

**AIM:**

To Write a LEX program to check valid date.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to check valid date validity which has

**Step 4:** in main(), input the date and call yylex().

**Step 5:** Display the output if Check Valid Date is valid or not.

**Step 6:** Stop.

## PROGRAM:

```
%%
```

```
[\n] {printf("\nEnter a valid Date(DD/MM/YYYY): ");}
```

```
([0-2][0-9]|3[0-1])\/(0[1-9]|1[0-2])\(/[0-9][0-9][0-9][0-9]) {printf("\nThe Date entered is valid\n");}
```

```
.* {printf("\nThe Date entered is invalid\n");}
```

```
%%
```

```
int yywrap(void){}
```

```
int main()
```

```
{
```

```
    printf("\nEnter a valid Date(DD/MM/YYYY): ");
```

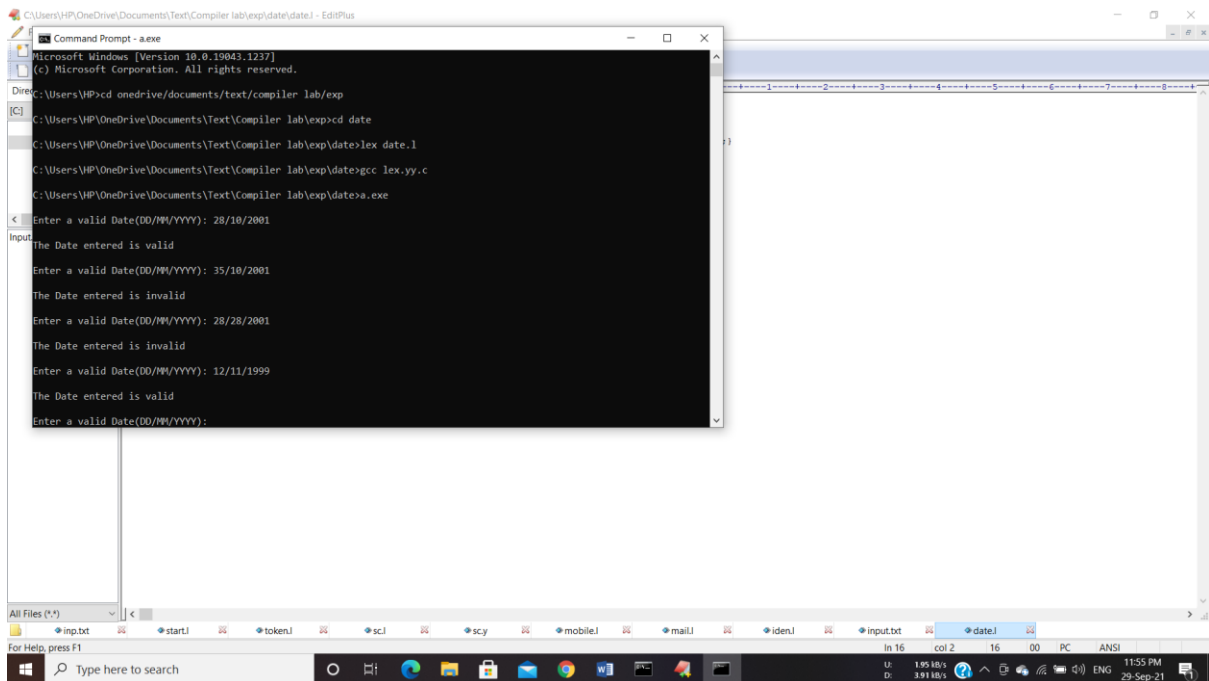
```
    yylex();
```

```
    return 0;
```

```
}
```



## OUTPUT:



```
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\date\ - EditPlus
Command Prompt - a.exe
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>cd onedrive\documents\text\compiler lab\exp
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp>cd date
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\date>lex date.l
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\date>gcc lex.yy.c
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\date>a.exe
Enter a valid Date(DD/MM/YYYY): 28/10/2001
The Date entered is valid
Enter a valid Date(DD/MM/YYYY): 35/10/2001
The Date entered is invalid
Enter a valid Date(DD/MM/YYYY): 28/28/2001
The Date entered is invalid
Enter a valid Date(DD/MM/YYYY): 12/11/1999
The Date entered is valid
Enter a valid Date(DD/MM/YYYY):
```

## RESULT:

Hence the code for program to check valid date is implemented using lex program.

**Exp. No: 4.4**

**Date: 15.09.21**

**LEX program to count the number  
of words starting with 'a'**

**AIM:**

To Write a LEX program to count the number of words starting with 'a'.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to count the word starting with 'a' letter a in the in given file

**Step 4:** in main(), input the characters and call yylex().

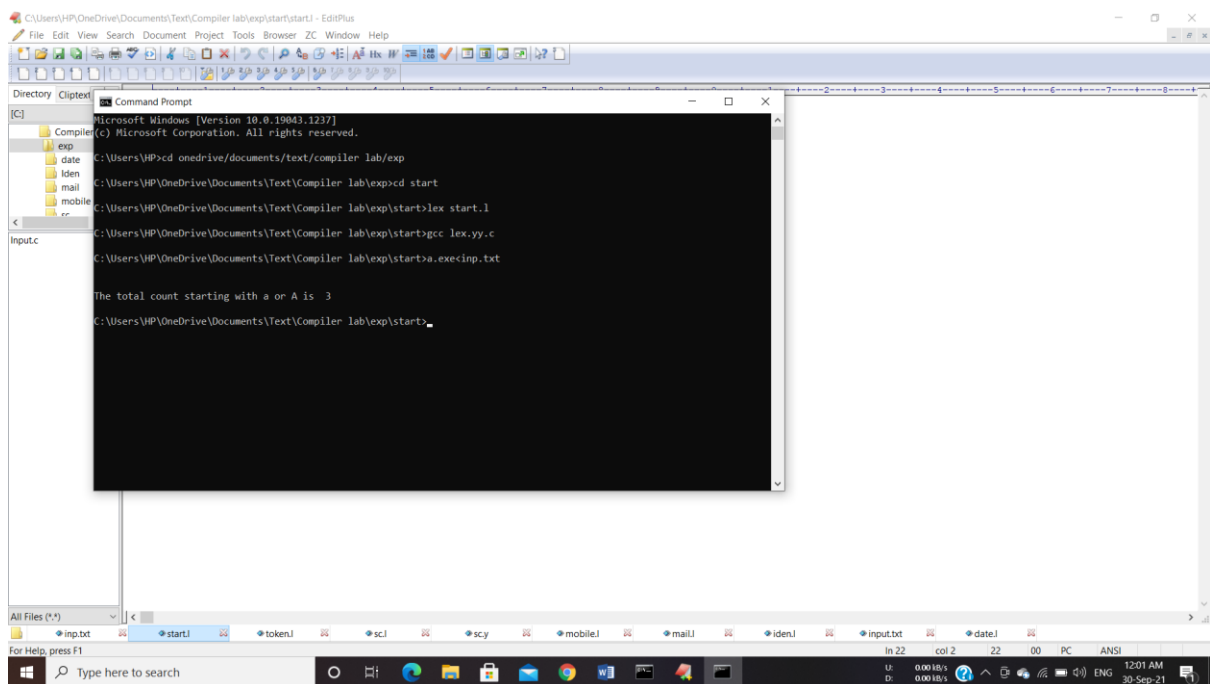
**Step 5:** Display the count of the words starting with letter a in the file

**Step 6:** Stop.

## PROGRAM:

```
% {  
    int count=0;  
% }  
  
alpha [a-zA-Z]  
digit [0-9]  
start ^aA  
  
%%  
  
{start} {count++;}  
(a|A)({alpha}|{digit})* {count++;}  
  
%%  
  
int yywrap(){  
main()  
{  
    yylex();  
    printf("The total count starting with a or A is %d\n",count);  
    return 0;  
}
```

## OUTPUT:



```
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\start1 - EditPlus
File Edit View Search Document Project Tools Browser ZC Window Help
Directory Cliptext Command Prompt
[C:] Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.
C:\Users\HP>cd onedrive\documents\text\compiler lab\exp
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp>cd start
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\start>lex start.l
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\start>gcc lex.yy.c
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\start>a.execinp.txt

The total count starting with a or A is 3
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\start>
```

## RESULT:

Hence the code for program to count of words starting with letter 'a' is implemented using lex program.

**Exp. No: 4.5**

**Date: 15.09.21**

**LEX program to separate the tokens  
from a given file**

**AIM:**

To Write a LEX program to separate the tokens from a given file.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to separate the tokens from a given file.

**Step 4:** in main(), input the token and call yylex().

**Step 5:** Display the output.

**Step 6:** Stop.

## PROGRAM:

```
% {  
int n = 0 ;  
% }  
%%  
"while"|"if"|"else" {n++;printf("\n token : %s", yytext);}  
  
"int"|"float" {n++;printf("\n token : %s", yytext);}  
  
[a-zA-Z_][a-zA-Z0-9_]* {n++;printf("\n token : %s", yytext);}  
  
"<="|"=="|"="|"++"|"-"|"*"|"+" {n++;printf("\n token : %s", yytext);}  
  
[(){}|,;] {n++;printf("\n token : %s", yytext);}  
  
[0-9]*"."[0-9]+ {n++;printf("\n token : %s", yytext);}  
  
[0-9]+ {n++;printf("\n token : %s", yytext);}  
  
. ;  
%%  
  
int yywrap(void){}  
int main()  
{  
  
    yylex();  
  
    printf("\n total no. of token = %d\n", n);  
}
```

## OUTPUT:

```
1 int n = 0 ;
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 token : int
24
25
26 token : float
27
28
29 token : 78
30
31 token : 90
32
33 token : h
34
35 token : a
36
37 token : d
38
39 token : are
40
41 token :
```

```
1 int n = 0 ;
2
3
4
5
6
7
8
9
10 token : 90
11
12 token : h
13
14 token : a
15
16 token : d
17
18 token : are
19
20 token :
21
22 token : case
23
24 token : default
25
26 token : printf
27
28 token : scanf
29
30 total no. of token = 13
31
32
33 C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\token>
34
```

## RESULT:

Hence the code for program to separate the tokens from a given file is implemented using lex program.

**Exp. No: 5**

**Date: 17.09.21**

## **YACC program to implement a scientific calculator**

**AIM:**

To Write a YACC program to implement a scientific calculator.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to design a scientific calculator by using lex as well as using yacc in the program.

**Step 4:** in main (), input the expression from the user and call yylex().

**Step 5:** Display the output.

**Step 6:** Stop.



## PROGRAM:

### Lex Program:

```
% {  
#include "y.tab.h"  
#include <math.h>  
% }  
%%  
([0-9]+|([0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?) { yylval.dval=atof(yytext);return  
NUMBER;}  
log |  
LOG {return LOG;}  
ln {return nLOG;}  
sin |  
SIN {return SINE;}  
cos |  
COS {return COS;}  
tan |  
TAN {return TAN;}  
mem {return MEM;}  
[\t];  
\$ return 0;  
\n|. return yytext[0];  
%%
```

### Yacc Program:

```
% {  
double memvar;  
% }  
%union  
{  
double dval;
```

```

}

%token<dval>NUMBER

%token<dval>MEM

%token LOG SINE nLOG COS TAN

%left '-' '+'

%left '*' '/'

%right '^'

%left LOG SINE nLOG COS TAN

%nonassoc UMINUS

%type<dval>expression

%%

start:statement'\n'

|start statement'\n'

;

statement:MEM='expression { memvar=$3;}

| expression{printf("Answer=%g\n",$1);}

;

expression:expression'+expression {$$=$1+$3;}

| expression '-' expression {$$=$1-$3;}

| expression '*' expression {$$=$1*$3;}

| expression '/' expression

{ if($3==0)

yyerror("divide by zero");

else

$$=$1/$3;

}

|expression'^'expression {$$=pow($1,$3);}

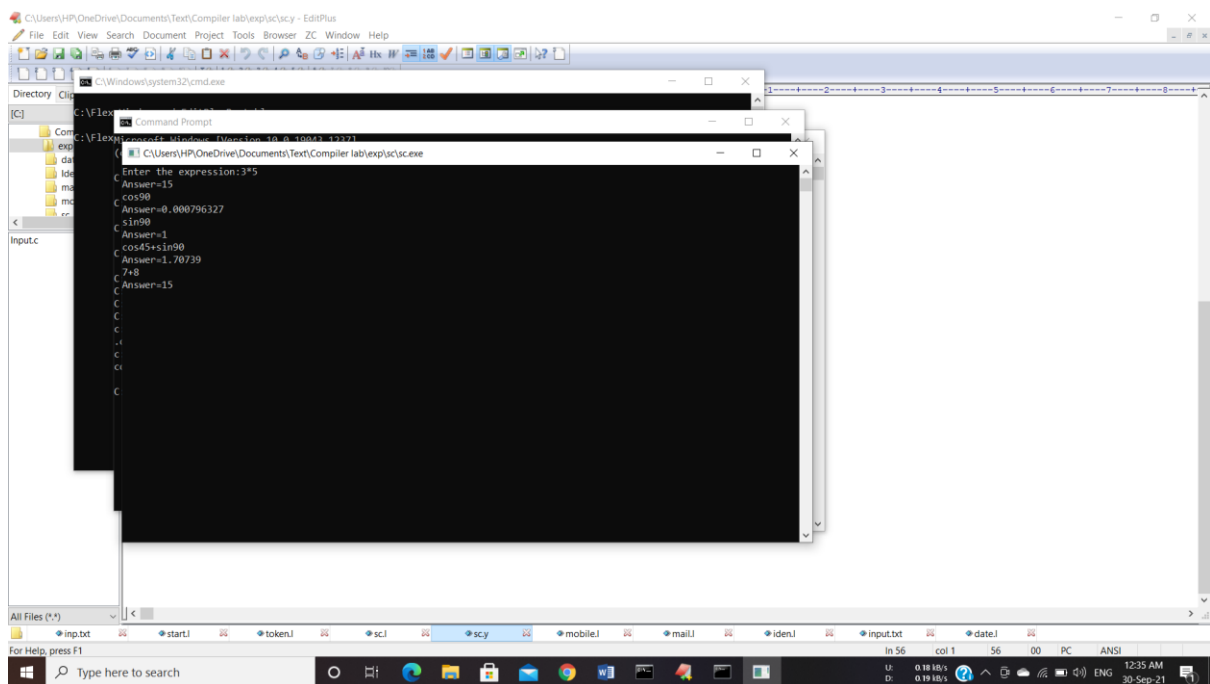
;

expression:'-'expression %prec UMINUS {$$=-$2;}

```

```
|('expression')'{$$=$2;}
|LOG expression {$$=log($2)/log(10);}
|nLOG expression {$$=log($2);}
|SINE expression {$$=sin($2*3.14/180);}
|COS expression {$$=cos($2*3.14/180);}
|TAN expression {$$=tan($2*3.14/180);}
|NUMBER {$$=$1;}
|MEM {$$=memvar;}
;
%%
int yywrap(void){}
main()
{
printf("Enter the expression:");
yyparse();}
int yyerror(char *error)
{
printf("%s\n",error);
}
```

## OUTPUT:



The screenshot shows a Windows desktop environment. In the background, there is a window titled 'C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\sc\scy - EditPlus'. In the foreground, there is a 'Command Prompt' window. The command prompt shows the following text:

```
C:\Users\HP\OneDrive\Documents\Text\Compiler lab\exp\sc\scy.exe
Enter the expression: 3*5
Answer=15
cos90
Answer=0.000796327
sin90
Answer=1
cos45+sin90
Answer=1.707107
7+8
Answer=15
```

The taskbar at the bottom shows several open applications: 'inp.bat', 'start1', 'token1', 'sci', 'scy', 'mobile1', 'mail1', 'iden1', 'input.txt', 'date1', and 'ANSI'. The system clock in the bottom right corner indicates the time is 12:35 AM on 30-Sep-21.

## RESULT:

Hence the given program to implement a scientific calculator is executed and compiled successfully by using lex and yacc program.

**Exp. No: 6**

## **Implementing parser**

**Exp. No: 6.1**

**Date: 22.09.21**

### **Design and implement Top-Down parsing techniques**

**AIM:**

To Design and implement Top-Down parsing techniques.

**ALGORITHM:**

**STEP-1:** Start

**STEP-2:** Input: string  $\omega$

**STEP-3:** Whenever a Non-terminal spend first time then go with the first alternative and compare with the given I/P String.

**STEP-4:** If matching doesn't occur then go with the second alternative and compare with the given I/P String.

**STEP-5:** If matching again not found then go with the alternative and so on

**STEP-6:** Moreover, If matching occurs for at least one alternative, then the I/P string is parsed successfully..

**STEP-7:** Stop.

## PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
char ip_sym[15],ip_ptr=0,op[50],tmp[50];
void e_prime();
void e();
void t_prime();
void t();
void f();
void advance();
int n=0;
void e()
{
strcpy(op,"TE");
printf("E=%-25s",op);
printf("E->TE\n");
t();
e_prime();
}
void e_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
if(op[i]!='e')
tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='E';n++);
if(ip_sym[ip_ptr]=='+')

```

```

{
i=n+2;
do
{
op[i+2]=op[i];
i++;
}while(i<=l);
op[n++]='+';
op[n++]='T';
op[n++]='E';
op[n++]=39;
printf("E=%-25s",op);
printf("E'->+TE\n");
advance();
t();
e_prime();
}
else
{
op[n]='e';
for(i=n+1;i<=strlen(op);i++)
op[i]=op[i+1];
printf("E=%-25s",op);
printf("E'->e");
}
}

void t()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
if(op[i]!='e')
tmp[n++]=op[i];

```

```

strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
i=n+1;
do
{
op[i+2]=op[i];
i++;
}while(i < l);
op[n++]='F';
op[n++]='T';
op[n++]=39;
printf("E=%-25s",op);
printf("T->FT\n");
f();
t_prime();
}
void t_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
if(op[i]!='e')
tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
if(ip_sym[ip_ptr]=='*')
{
i=n+2;
do
{
op[i+2]=op[i];

```



```

i++;
}while(i < l);
op[n++]='*';
op[n++]='F';
op[n++]='T';
op[n++]=39;
printf("E=%-25s",op);
printf("T'->*FT\n");
advance();
f();
t_prime();
}
else
{
op[n]='e';
for(i=n+1;i<=strlen(op);i++)
op[i]=op[i+1];
printf("E=%-25s",op);
printf("T'->e\n");
}
}
void f()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
if(op[i]!='e')
tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='F';n++);
if((ip_sym[ip_ptr]=='i')||(ip_sym[ip_ptr]=='T'))
{

```

```
op[n]='i';
printf("E=%-25s",op);
printf("F->i\n");
advance();
}
else
{
if(ip_sym[ip_ptr]=='(')
{
advance();
e();
if(ip_sym[ip_ptr]==')')
{
advance();
i=n+2;
do
{
op[i+2]=op[i];
i++;
}while(i<=l);
op[n++]='(';
op[n++]='E';
op[n++]=')';
printf("E=%-25s",op);
printf("F->(E)\n");
}
}
else
{
printf("\n\t syntax error");
getch();
exit(1);
```

```

}
}
}
void advance()
{
ip_ptr++;
}
void main()
{
int i;
printf("\nGrammar without left recursion"); printf("\n\t\t E->TE' \n\t\t E'->+TE'e \n\t\t T- >FT' ");
printf("\n\t\t T'->*FT'e \n\t\t F->(E)i"); printf("\n Enter the input expression:");
gets(ip_sym);
printf("Expressions");
printf("\t Sequence of production rules\n");
e();
for(i=0;i < strlen(ip_sym);i++)
{
if(ip_sym[i]!='+'&&ip_sym[i]!='*'&&ip_sym[i]!='('&&
ip_sym[i]!=')'&&ip_sym[i]!='i'&&ip_sym[i]!='T')
{
printf("\nSyntax error");
break;
}
for(i=0;i<=strlen(op);i++)
if(op[i]!='e')
tmp[n++]=op[i];
strcpy(op,tmp);
printf("\nE=%-25s",op);
}
getch();
}

```

## OUTPUT:

```
C:\TURBOC3\BIN>TC

Grammar without left recursion
      E->TE'
      E'->+TE' ie
      T->FT'
      T'->*FT' ie
      F->(E){i

Enter the input expression:i*i+i
Expressions      Sequence of production rules
E=TE'            E->TE'
E=FT'E'          T->FT'
E=iT'E'          F->i
E=i*FT'E'        T'->*FT'
E=i*iT'E'        F->i
E=i*ieE'         T'->e
E=i*i+TE'        E'->+TE'
E=i*i+FT'E'      T->FT'
E=i*i+iT'E'      F->i
E=i*i+ieE'       T'->e
E=i*i+ie         E'->e
E=i*i+i
```

## RESULT:

Hence, the top-down parser are implemented and executed successfully got desired output.

**Exp. No: 6.2**

**Date: 24.09.21**

**Design and implement Bottom-up parsing  
techniques**

**AIM:**

To Design and implement Bottom-up parsing techniques.

**ALGORITHM:**

1. Start
2. token = next\_token()
3. repeat forever
4. s = top of stack
5. if action[s, token] = —shift si then
6. PUSH token
7. PUSH si
8. token = next\_token()
9. else if action[s, token] = —reduce  $A ::= \beta$ — then
10. POP 2 \*  $|\beta|$  symbols
11. s = top of stack
12. PUSH A
13. PUSH goto[s, A]
14. else if action[s, token] = —accept then
15. return
16. else error()
17. Stop

## PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <string.h>

char ip_sym[15], stack[15];

int ip_ptr = 0, st_ptr = 0, len, i;

char temp[2], temp2[2];

char act[15];

void check();

void main()

{

printf("\n\t\t SHIFT REDUCE PARSER\n");

printf("\n GRAMMER\n");

printf("\n E->E+E\n E->E/E");

printf("\n E->E*E\n E->a/b");

printf("\n enter the input symbol:\t");

gets(ip_sym);

printf("\n\t stack implementation table");

printf("\n stack\t\t input symbol\t\t action");

printf("\n____\t\t ____\t\t ____\n");

printf("\n $\t\t %s$\t\t t--", ip_sym);

strcpy(act, "shift ");

temp[0] = ip_sym[ip_ptr];

temp[1] = '\0';

strcat(act, temp);

len = strlen(ip_sym);

for (i = 0; i <= len - 1; i++)

{

stack[st_ptr] = ip_sym[ip_ptr];

stack[st_ptr + 1] = '\0';

ip_sym[ip_ptr] = ' ';
```

```

ip_ptr++;
printf("\n $%s\t\t%s$\t\t\t%s", stack, ip_sym, act);
strcpy(act, "shift ");
temp[0] = ip_sym[ip_ptr];
temp[1] = '\0';
strcat(act, temp);
check();
st_ptr++;
}
st_ptr++;
check();
}
void check()
{
int flag = 0;
temp2[0] = stack[st_ptr];
temp2[1] = '\0';
if ((!strcmpi(temp2, "a")) || (!strcmpi(temp2, "b")))
{
stack[st_ptr] = 'E';
if (!strcmpi(temp2, "a"))
printf("\n $%s\t\t%s$\t\t\tE->a", stack, ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->b", stack, ip_sym);
flag = 1;
}
if ((!strcmpi(temp2, "+")) || (strcmpi(temp2, "*")) || (!strcmpi(temp2, "/")))
{
flag = 1;
}
if ((!strcmpi(stack, "E+E")) || (!strcmpi(stack, "E\E")) || (!strcmpi(stack, "E*E")))
{

```

```
strcpy(stack, "E");
st_ptr = 0;
if (!strcmpi(stack, "E+E"))
printf("\n $%s\t\t%s$\t\tE->E+E", stack, ip_sym);
else if (!strcmpi(stack, "E\E"))
printf("\n $%s\t\t %s$\t\tE->E\E", stack, ip_sym);
else
printf("\n $%s\t\t%s$\t\tE->E*E", stack, ip_sym);
flag = 1;
}
if (!strcmpi(stack, "E") && ip_ptr == len)
{
printf("\n $%s\t\t%s$\t\tACCEPT", stack, ip_sym);
getch();
exit(0);
}
if (flag == 0)
{
printf("\n%s\t\t\t%s\t\t reject", stack, ip_sym);
exit(0);
}
return;
}
```



## OUTPUT:

```
C:\TURBOC3\BIN>TC

                        SHIFT REDUCE PARSER

GRAMMER
E->E+E
E->E/E
E->E*E
E->a/b
enter the input symbol:      a+b

stack      stack implementation table      action
-----
$          a+b$      --
$a         +b$      shift a
$E         +b$      E->a
$E+        b$      shift +
$E+b       $       shift b
$E+E       $       E->b
$E         $       E->E*E
$E         $       ACCEPT
```

## RESULT:

Hence, the bottom up parser are implemented and executed successfully got desired output.

**Exp. No: 7**

**Date: 24.09.21**

**GENERATE ABSTRACT SYNTAX TREE AND  
INTERMEDIATE CODE FOR THE GIVEN LANGUAGE**

**AIM:**

To generate abstract syntax tree and intermediate code for the given language.

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to implement code to generate abstract syntax tree and intermediate code for the given language

**Step 4:** Give the necessary condition

**Step 5:** Display the output.

**Step 6:** Stop.

**PROGRAM:**

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

char op[2],arg1[5],arg2[5],result[5];

void main(){

FILE *fp1,*fp2;

fp1=fopen("input.txt","r");

fp2=fopen("output.txt","w");

while(!feof(fp1))

{

fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);

if(strcmp(op,"+")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1);

fprintf(fp2,"\nADD R0,%s",arg2);

fprintf(fp2,"\nMOV %s,R0",result);

}

if(strcmp(op,"*")==0)

{

fprintf(fp2,"\nMOV R0,%s",arg1);

fprintf(fp2,"\nMUL R0,%s",arg2);

fprintf(fp2,"\nMOV %s,R0",result);

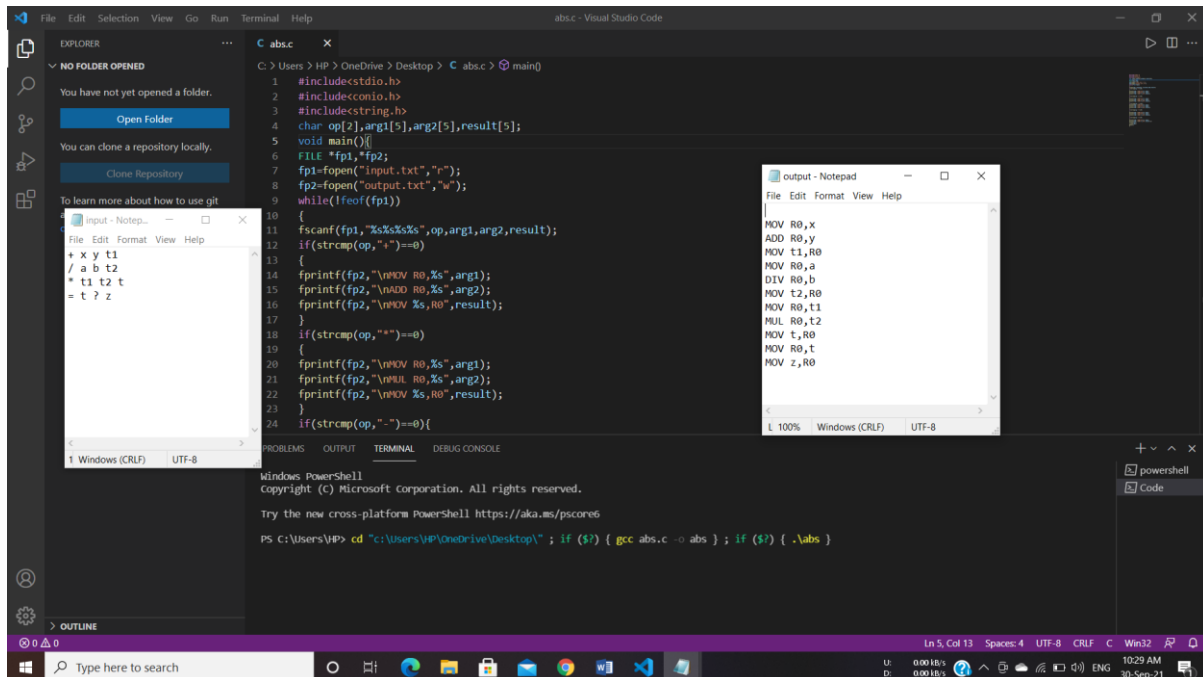
}

if(strcmp(op,"-")==0){

fprintf(fp2,"\nMOV R0,%s",arg1);
```

```
fprintf(fp2, "\nSUB R0,%s",arg2);
fprintf(fp2, "\nMOV %s,R0",result);
}
if(strcmp(op,"/")==0)
{
fprintf(fp2, "\nMOV R0,%s",arg1);
fprintf(fp2, "\nDIV R0,%s",arg2);
fprintf(fp2, "\nMOV %s,R0",result);
}
if(strcmp(op,"=")==0)
{
fprintf(fp2, "\nMOV R0,%s",arg1);
fprintf(fp2, "\nMOV %s,R0",result);}}
fclose(fp1);
fclose(fp2);
getch();
}
```

## OUTPUT:



```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
void main()
{
FILE *fp1,*fp2;
fp1=fopen("input.txt","r");
fp2=fopen("output.txt","w");
while(!feof(fp1))
{
fscanf(fp1,"%s%s%s",op,arg1,arg2,result);
if(strcmp(op,"+")==0)
{
printf(fp2,"%s\n",result);
}
if(strcmp(op,"-")==0)
{
printf(fp2,"%s\n",result);
}
if(strcmp(op,"*")==0)
{
printf(fp2,"%s\n",result);
}
if(strcmp(op,"/")==0)
{
printf(fp2,"%s\n",result);
}
}
}
```

```
MOV R0,x
ADD R0,y
MOV t1,R0
MOV R0,a
DIV R0,b
MOV t2,R0
MOV R0,t1
MUL R0,t2
MOV t,R0
MOV R0,t
MOV z,R0
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\VP> cd "C:\Users\VP\OneDrive\Desktop\" ; if ($?) { gcc abs.c -o abs } ; if ($?) { .\abs }
```

## RESULT:

Thus, the given program to generate abstract syntax tree and intermediate code for the given input successfully.

**Exp. No: 8**

**Date: 28.09.21**

## **IMPLEMENT CODE OPTIMIZATION PHASE OF THE COMPILER**

### **AIM:**

To write a C program to implement simple code optimization technique.

### **ALGORITHM:**

**Step 1:** Start.

**Step 2:** Define the function and rule sections.

**Step 3:** Write condition to implement code optimization phase of the compiler

**Step 4:** Give the necessary condition

**Step 5:** Display the output.

**Step 6:** Stop

## PROGRAM:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

char s[20],o[20];

void main()

{

int i=0,j=0,k,f1=1,f=1,k1=0;

void part();

//clrscr();

printf("\n Enter the input string\n");

scanf("%s",o);

strlen(o);

while(o[k1]!='\0')

{

if((o[k1]=='')==1)

{

break;

}

k1++;

}

for(j=k1+1;j<strlen(o);j++)

{

s[i]=o[j];

i++;

}

s[i]='\0';

i=strlen(s);

j=0;

printf("\n Three address code is\n");

if(i>3)

{
```

```
while(s[j]!='\0')
{
if((s[j]=='*')==1||(s[j]=='/')==1)
{
k=j;
if(f1!=0)
{
printf("t1=%c\n",s[k+1]);
printf("t2=%c%c\t",s[k-1],s[k]);
}
else
{
if(k>3)
{
printf("t2=t1%c%c\n",s[k],s[k+1]);
}
else
{
printf("\t2=t1%c%c\n",s[k],s[k-1]);
}
}
f=0;
break;
}
j++;
}
j=0;
while(s[j]!='\0')
{
if((s[j]=='+')==1||(s[j]=='-')==1)
{
k=j;
```



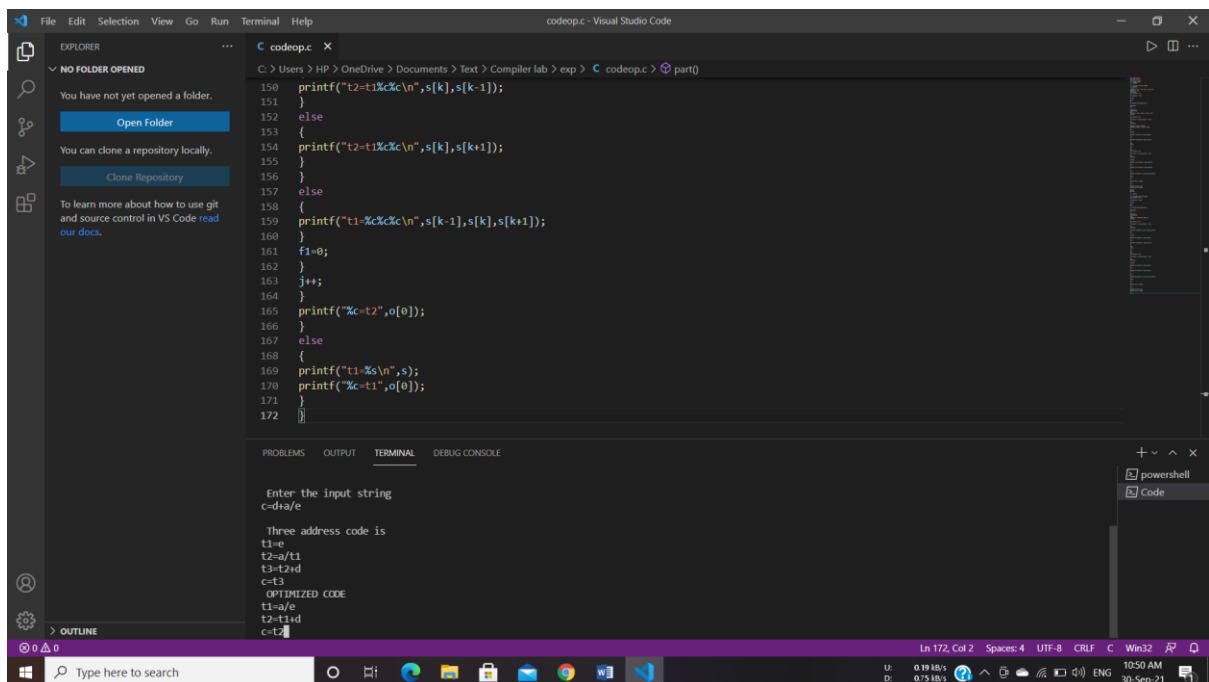
```
if(f==0)
{
if(k<3)
{
printf("\nt3=t2%c%c\n",s[k],s[k-1]);
}
else
{
printf("\nt3=t2%c%c\n",s[k],s[k+1]);
}
}
else
{
printf("t1=%c%c%c\n",s[k-1],s[k],s[k+1]);
}
f1=0;
}
j++;
}
printf("%c=t3",o[0]);
}
else
{
printf("t1=%s\n",s);
printf("%c=t1",o[0]);
}
part();
getch();
}
void part()
{
int i=0,j=0,k,f1=1,f=1,k1=0;
```

```
while(o[k1]!='\0')
{
if((o[k1]=='')==1)
{
break;
}
k1++;
}
for(j=k1+1;j<strlen(o);j++)
{
s[i]=o[j];
i++;
}
s[i]='\0';
i=strlen(s);
j=0;
printf("\n OPTIMIZED CODE\n");
if(i>3)
{
while(s[j]!='\0')
{
if((s[j]=='*')==1||(s[j]=='/')==1)
{
k=j;
if(f1!=0)
{
printf("t1=%c%c%c\n",s[k-1],s[k],s[k+1]);
}
else
{
if(k>3)
{
```

```
printf("t2=t1%c%c\n",s[k],s[k+1]);
}
else
{
printf("\t2=t1%c%c\n",s[k],s[k-1]);
}
}
f=0;
break;
}
j++;
}
j=0;
while(s[j]!='\0')
{
if((s[j]=='+')==1||(s[j]=='-')==1)
{
k=j;
if(f==0)
{
if(k<3)
{
printf("t2=t1%c%c\n",s[k],s[k-1]);
}
else
{
printf("t2=t1%c%c\n",s[k],s[k+1]);
}
}
}
else
{
printf("t1=%c%c%c\n",s[k-1],s[k],s[k+1]);
```

```
}  
f1=0;  
}  
j++;  
}  
printf("%c=t2",o[0]);  
}  
else  
{  
printf("t1=%s\n",s);  
printf("%c=t1",o[0]);  
}  
}
```

## OUTPUT:



The screenshot displays the Visual Studio Code interface with a C program open in the editor. The program is a recursive function that prints the address of the first element of an array and then prints the array elements in reverse order. The terminal window shows the execution of the program, where the user enters the input string 'c=daa/e'. The output shows the address of the first element (t1=e) and the array elements in reverse order (t2=a/t1, t3=t2nd, c=t3). The terminal also shows the optimized code (t1=a/e, t2=t1nd, c=t2).

```
150 printf("t2=t1%c%c\n",s[k],s[k-1]);
151 }
152 else
153 {
154 printf("t2=t1%c%c\n",s[k],s[k+1]);
155 }
156 }
157 else
158 {
159 printf("t1=%c%c%c\n",s[k-1],s[k],s[k+1]);
160 }
161 f1=0;
162 }
163 j++;
164 }
165 printf("%c=t2",o[0]);
166 }
167 else
168 {
169 printf("t1=%s\n",s);
170 printf("%c=t1",o[0]);
171 }
172 }
```

Enter the input string  
c=daa/e

Three address code is  
t1=e  
t2=a/t1  
t3=t2nd  
c=t3  
OPTIMIZED CODE  
t1=a/e  
t2=t1nd  
c=t2

## RESULT:

The given program implements code optimization phase of the compiler is executed and compiled successfully.