

Date: - __ / __ / ____

Practical-1

- AIM:** - a) Write a C program to remove all the comments from the program.
- b) Write a C program to recognize identifiers and numbers.

Solution: -

- a) Write a C program to remove all the comments from the program

Code:

```
#include <stdio.h>

#include <stdbool.h>

void remove_comments(FILE* input_file, FILE* output_file) {

    int c, next_c;

    bool in_single_line_comment = false;

    bool in_multi_line_comment = false;

    while ((c = fgetc(input_file)) != EOF) {

        if (!in_single_line_comment && !in_multi_line_comment) {

            if (c == '/') {

                if ((next_c = fgetc(input_file)) == '/') {

                    in_single_line_comment = true;

                } else if (next_c == '*') {

                    in_multi_line_comment = true;

                } else {

                    fputc(c, output_file);

                }

            }

        }

    }

}
```

```
        if (next_c != EOF)

            fputc(next_c, output_file);

    }

} else {

    fputc(c, output_file);

}

} else if (in_single_line_comment) {

    if (c == '\n') {

        in_single_line_comment = false;

        fputc(c, output_file);

    }

} else if (in_multi_line_comment) {

    if (c == '*') {

        if ((next_c = fgetc(input_file)) == '/')

            in_multi_line_comment = false;

    }

}

}

}

int main() {

    FILE* input_file = fopen("input.c", "r");

    FILE* output_file = fopen("output.c", "w");

    if (input_file == NULL || output_file == NULL) {

        printf("Error opening files.\n");
```

```
    return 1;
}

remove_comments(input_file, output_file);

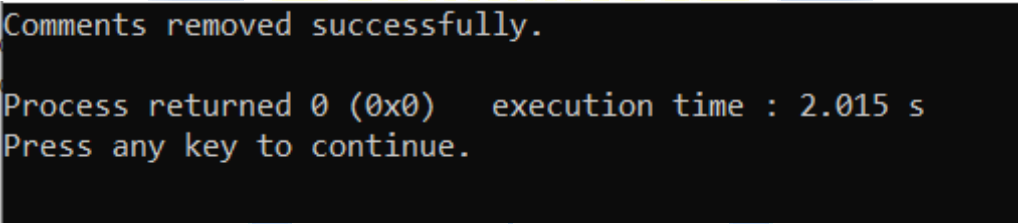
printf("Comments removed successfully.\n");

fclose(input_file);

fclose(output_file);

return 0;
}
```

Output:

A screenshot of a terminal window with a black background and white text. The text shows the output of a program: 'Comments removed successfully.', followed by 'Process returned 0 (0x0) execution time : 2.015 s' and 'Press any key to continue.'.

```
Comments removed successfully.
Process returned 0 (0x0)   execution time : 2.015 s
Press any key to continue.
```

MBIT

```
input.c x output.c x
1 #include <stdio.h>
2 int main() {
3
4     int i, n;
5
6     // initialize first and second terms
7     int t1 = 0, t2 = 1;
8
9     // initialize the next term (3rd term)
10    int nextTerm = t1 + t2;
11
12    // get no. of terms from user
13    printf("Enter the number of terms: ");
14    scanf("%d", &n);
15
16    // print the first two terms t1 and t2
17    printf("Fibonacci Series: %d, %d, ", t1, t2);
18
19    // print 3rd to nth terms
20    for (i = 3; i <= n; ++i) {
21        printf("%d, ", nextTerm);
22        t1 = t2;
23        t2 = nextTerm;
24        nextTerm = t1 + t2;
25    }
26
27    return 0;
28 }
29
30
```

1. With Comments

```
input.c x output.c x
1 #include <stdio.h>
2 int main() {
3
4     int i, n;
5
6
7     int t1 = 0, t2 = 1;
8
9
10    int nextTerm = t1 + t2;
11
12
13    printf("Enter the number of terms: ");
14    scanf("%d", &n);
15
16
17    printf("Fibonacci Series: %d, %d, ", t1, t2);
18
19
20    for (i = 3; i <= n; ++i) {
21        printf("%d, ", nextTerm);
22        t1 = t2;
23        t2 = nextTerm;
24        nextTerm = t1 + t2;
25    }
26
27    return 0;
28 }
29
30
```

2. Without Comments

b) Write a C program to recognize identifiers and numbers.

Code:

```
#include <stdio.h>

#include <stdbool.h>

#include <ctype.h>

#include <string.h>

bool isIdentifier(const char* str) {

    if (!isalpha(str[0]) && str[0] != '_')

        return false;

    for (int i = 1; i < strlen(str); i++) {

        if (!isalnum(str[i]) && str[i] != '_')

            return false;

    }

    return true;

}

bool isNumber(const char* str) {

    for (int i = 0; i < strlen(str); i++) {

        if (!isdigit(str[i]))

            return false;

    }

    return true;

}

int main() {
```

```
char input[100];

printf("Enter a string: ");

fgets(input, sizeof(input), stdin);

// Remove newline character

input[strcspn(input, "\n")] = '\0';

if (isIdentifier(input)) {

    printf("%s is an identifier.\n", input);

} else if (isNumber(input)) {

    printf("%s is a number.\n", input);

} else {

    printf("%s is neither an identifier nor a number.\n", input);

}

return 0;

}
```

Output:

```
Enter a string: _John75
_John75 is an identifier.

Process returned 0 (0x0)   execution time : 11.814 s
Press any key to continue.
```

Date: - __ / __ / ____

Practical-2

AIM: - Write a C program to generate tokens for a C program.

Solution: -

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

/* Token types */

typedef enum {

    IDENTIFIER,

    KEYWORD,

    OPERATOR,

    INTEGER_CONSTANT,

    FLOAT_CONSTANT,

    STRING_CONSTANT,

    PUNCTUATION,

    COMMENT,

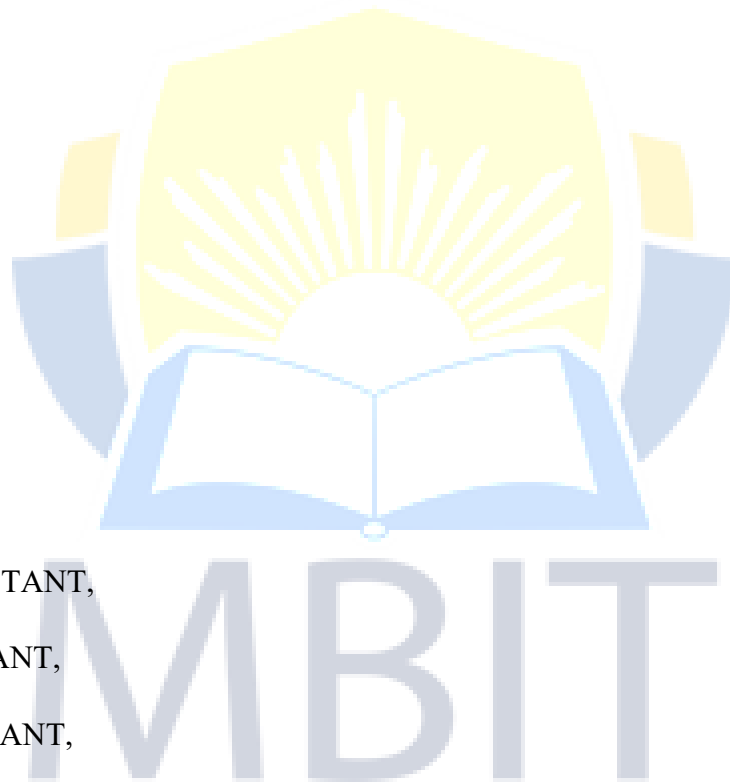
    NEWLINE

} TokenType;

/* Token structure */

typedef struct {

    TokenType type;
```



```
char lexeme[100];

} Token;

/* Function to check if a character is a valid identifier character */

int isIdentifierChar(char c) {

    return isalnum(c) || c == '_' ;

}

/* Function to print tokens */

void printToken(TokenType type, const char* lexeme) {

    const char* typeString;

    switch (type) {

        case IDENTIFIER:

            typeString = "IDENTIFIER";

            break;

        case KEYWORD:

            typeString = "KEYWORD";

            break;

        case OPERATOR:

            typeString = "OPERATOR";

            break;

        case INTEGER_CONSTANT:

            typeString = "INTEGER_CONSTANT";

            break;

        case FLOAT_CONSTANT:

            typeString = "FLOAT_CONSTANT";
```



```
        break;

    case STRING_CONSTANT:

        typeString = "STRING_CONSTANT";

        break;

    case PUNCTUATION:

        typeString = "PUNCTUATION";

        break;

    case COMMENT:

        typeString = "COMMENT";

        break;

    case NEWLINE:

        typeString = "NEWLINE";

        break;

    default:

        typeString = "UNKNOWN";

    }

    printf("Token: %-16s Lexeme: %s\n", typeString, lexeme);

}

int main() {


    char code[1000];

    printf("Enter C code:\n");

    fgets(code, sizeof(code), stdin);

    Token token;

    int length = strlen(code);
```



```
int i = 0;

while (i < length) {

    /* Skip whitespace */

    if (isspace(code[i])) {

        i++;

        continue;

    }

    /* Check for comments */

    if (code[i] == '/' && code[i + 1] == '/') {

        token.type = COMMENT;

        strncpy(token.lexeme, code + i, 2);

        token.lexeme[2] = '\0';

        printToken(token.type, token.lexeme);

        i += 2;

        /* Skip the rest of the line */

        while (i < length && code[i] != '\n') {

            i++;

        }

        token.type = NEWLINE;

        strncpy(token.lexeme, code + i, 1);

        token.lexeme[1] = '\0';

        printToken(token.type, token.lexeme);

        i++;

        continue;

    }

}
```

```

}

/* Check for identifiers and keywords */

if (isalpha(code[i]) || code[i] == '_') {

    int j = i;

    while (isIdentifierChar(code[j])) {

        j++;

    }

    int lexemeLength = j - i;

    strncpy(token.lexeme, code + i, lexemeLength);

    token.lexeme[lexemeLength] = '\0';

    /* Check if it's a keyword */

    const char* keywords[] = {

        "int", "float", "char", "if", "else", "while", "for", "return"

        // Add more keywords as needed

    };

    int numKeywords = sizeof(keywords) / sizeof(keywords[0]);

    int isKeyword = 0;

    for (int k = 0; k < numKeywords; k++) {

        if (strcmp(token.lexeme, keywords[k]) == 0) {

            isKeyword = 1;

            break;

        }

    }

    token.type = isKeyword ? KEYWORD : IDENTIFIER;

```

```
printToken(token.type, token.lexeme);

i = j;

continue;

}

/* Check for integer and floating-point constants */

if (isdigit(code[i])) {

    int j = i;

    int isFloat = 0;

    while (isdigit(code[j]) || code[j] == '.') {

        if (code[j] == '.') {

            isFloat = 1;

        }

        j++;

    }

    int lexemeLength = j - i;

    strncpy(token.lexeme, code + i, lexemeLength);

    token.lexeme[lexemeLength] = '\0';

    token.type = isFloat ? FLOAT_CONSTANT : INTEGER_CONSTANT;

    printToken(token.type, token.lexeme);

    i = j;

    continue;

}

/* Check for string constants */

if (code[i] == '"') {
```

```
int j = i + 1;

while (j < length && code[j] != '"') {

    j++;

}

int lexemeLength = j - i + 1;

strncpy(token.lexeme, code + i, lexemeLength);

token.lexeme[lexemeLength] = '\0';

token.type = STRING_CONSTANT;

printToken(token.type, token.lexeme);

i = j + 1;

continue;

}

/* Check for operators and punctuation */

const char* operators = "+-*/=<>!&|";

const char* punctuation = "{}()[];";

if (strchr(operators, code[i]) != NULL) {

    token.type = OPERATOR;

    strncpy(token.lexeme, code + i, 1);

    token.lexeme[1] = '\0';

    printToken(token.type, token.lexeme);

    i++;

    continue;

}

if (strchr(punctuation, code[i]) != NULL) {
```

```
token.type = PUNCTUATION;

strncpy(token.lexeme, code + i, 1);

token.lexeme[1] = '\0';

printToken(token.type, token.lexeme);

i++;

continue;

}

/* Handle unrecognized characters */

printf("Unrecognized character: %c\n", code[i]);

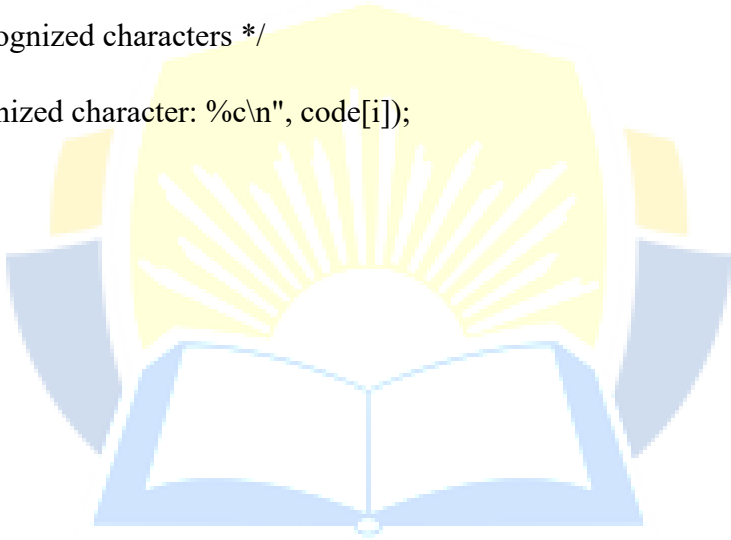
i++;

}

return 0;

}
```

Output:



```
Enter C code:
int a=5;
Token: KEYWORD           Lexeme: int
Token: IDENTIFIER        Lexeme: a
Token: OPERATOR          Lexeme: =
Token: INTEGER_CONSTANT  Lexeme: 5
Token: PUNCTUATION       Lexeme: ;

Process returned 0 (0x0)   execution time : 8.274 s
Press any key to continue.
```

Date: - __ / __ / ____

Practical-3

AIM: - a) To Study about Lexical Analyzer Generator (LEX)

b) Create a Lex program to take input from text file and count no. of characters, no. of lines & no. of words.

Solution: -

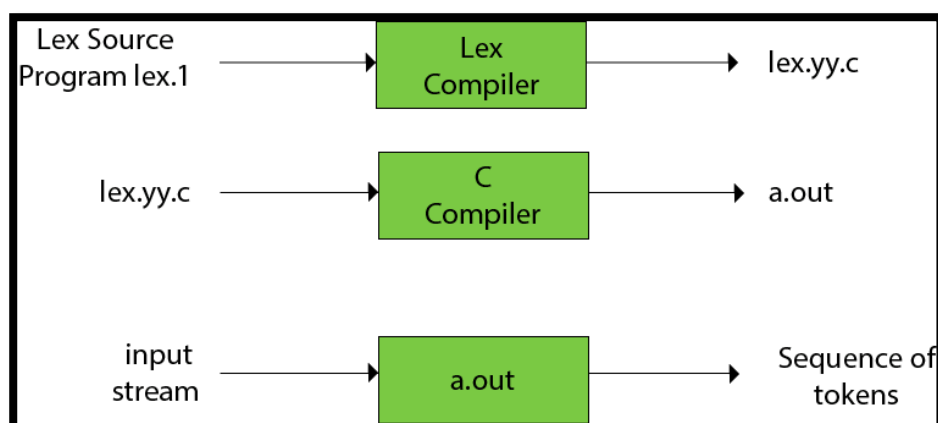
a) To Study about Lexical Analyzer Generator (LEX).

➤ **Lexical Analyzer Generator (LEX):**

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

➤ **The function of Lex is as follows:**

- Firstly lexical analyzer creates a program lex.l in the Lex language. Then Lex compiler runs the lex.l program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



➤ Lex File Format:

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

{ definitions }

%%

{ rules }

%%

{ user subroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form $p_1 \{action_1\} p_2 \{action_2\} \dots p_n \{action\}$.

Where p_i describes the regular expression and $action_1$ describes the actions what action the lexical analyzer should take when pattern p_i matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

➤ Parts of the LEX program:

The layout of a LEX source program is:

- a. Definitions
- b. Rules
- c. Auxiliary routines

A double modulus sign separates each section %%.

a. Definitions:

The definitions are at the top of the LEX source file. It includes the regular definitions, the C, directives, and any global variable declarations. They consist of two parts, auxiliary declarations and regular definitions. They are not processed by the lex tool instead are copied by the lex to the output file lex.yy.c file. The code specific to C is placed within %{and %}.

An example of auxiliary declarations

```
%{  
  
    #include<stdio.h>  
  
    int global_variable;  
  
}%
```

An example of Regular definitions

number [0-9] +

op [-|+|*|/|^|=]

b. Rules:

This section may contain multiple rules. Each rule consists of:

- A regular expression (name)
- A piece of code (output action)

They execute whenever a token in the input stream matches with the grammar.

An example

```
%%  
  
    {number} {printf(" number");}  
  
    {op} {printf(" operator");}  
  
%%
```

Using the above rules we have the following outputs for the corresponding inputs;

INPUT	OUTPUT
13	number
+	operator

INPUT	OUTPUT
13 + 17	number operator number

c. Auxiliary routines

This section includes functions that may be required in the rules section. Here, a function is written in regular C syntax. Most simple lexical analyzers only require the main() function.

The yytext keyword gives the current lexeme. The generated code is placed into a function called yylex(). The main() function always calls the yylex() function.

An example

```

/* declarations */

%%

/* rules */

%%

/* functions */

int main()
{
    yylex();

    return 1;
}

```



Declarations and functions are then copied to the lex.yy.c file which is compiled using the command gcc lex.yy.c.

➤ **yyvariables.**

These are variables given by the lex which enable the programmer to design a sophisticated lexical analyzer.

They include `yyin` which points to the input file, `yytext` which will hold the lexeme currently found and `yyleng` which is a `int` variable that stores the length of the lexeme pointed to by `yytext` as we shall see in later sections.

➤ **`yyin`.**

It points to the input file set by the programmer, if not assigned, it defaults to point to the console `input(stdin)`.

➤ **`yytext`.**

Each invocation of `yylex()` function will result in a `yytext` which carries a pointer to the lexeme found in the input stream `yylex()`.

This is overwritten on each `yylex()` function invocation.

➤ **`yyleng`.**

The output is the number of digits.

➤ **`yyfunctions`.**

These are `yylex()` and `yywrap()`

○ **`yylex()`.**

- It is defined by `lex` in `lex.yy.c` but it not called by it.
- It is called in the auxilliary functions section in the `lex` program and returns an `int`.
- Code generated by the `lex` is defined by `yylex()` function according to the specified rules.
- When called, input is read from `yyin`(not defined, therefore read from console) and scans through input for a matching pattern(part of or whole).
- When pattern is found, the corresponding action is executed(`return atoi(yytext)`).
- The matched number is stored in `num` variable and printed using `printf()`.
- This continues until a return statement is invoked or end of input is reached

○ **yywrap().**

- It is defined in the auxilliary function section.

It is called by the yylex() function when end of input is encountered and has an int return type.

- If the function returns a non-zero(true), yylex() will terminate the scanning process and returns 0, otherwise if yywrap() returns 0(false), yylex() will assume that there is more input and will continue scanning from location pointed at by yyin.
- *% option noyywrap* is declared in the declarations section to avoid calling of yywrap() in lex.yy.c file. It is mandatory to either define yywrap() or indicate its absence using the describe option above.



b) Create a Lex program to take input from text file and count no. of characters, no. of lines & no. of words.

Code:

```
%{  
  
#include<stdio.h>  
  
int sc=0,wc=0,lc=0,cc=0;  
  
%}  
  
%%  
  
[\\n] { lc++; cc+=yyleng;}  
[ \\t] { sc++; cc+=yyleng;}  
[^\\t\\n ]+ { wc++; cc+=yyleng;}  
  
%%  
  
int main(int argc ,char* argv[ ])  
{  
  
    printf("Enter the input:\\n");  
  
    yylex();  
  
    printf("The number of lines=%d\\n",lc);  
  
    printf("The number of spaces=%d\\n",sc);  
  
    printf("The number of words=%d\\n",wc);  
  
    printf("The number of characters are=%d\\n",cc);  
  
}
```

```
int yywrap()  
  
{  
  
    return 1;  
  
}
```

Output:

```
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ lex Practical4c.l  
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ cc lex.yy.c -ll  
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ ./a.out  
This is my house.  
It is very big.  
Line Count: 2  
Word Count: 8  
Space Count: 6  
Character Count: 26
```



Date: - __/ __/ ____

Practical-4

AIM: - a) WAP to implement yytext method in a LEX program.

b) WAP to implement ECHO, REJECT functions provided in Lex.

c) WAP to implement BEGIN directive in a LEX program.

Solution: -

a) WAP to implement yytext method in a LEX program.

Code:

```
/* LEX code to replace a word with another
```

```
taking input from file */
```

```
/* Definition section */
```

```
/* character array line can be
```

```
accessed inside rule section and main() */
```

```
%{
```

```
#include<stdio.h>
```

```
#include<string.h>
```

```
char line[100];
```

```
%}
```

```
/* Rule Section */
```

```
/* Rule 1 writes the string stored in line
```

```
character array to file output.txt */
```

```
/* Rule 2 copies the matched token
```

```
i.e every character except newline character
```

```

        to line character array */

%%

[ '\n' ] { fprintf(yyout,"%s\n",line);}

(.* )    { strcpy(line,yytext);}

<<EOF>> { fprintf(yyout,"%s",line); return 0;}

%%

int yywrap()

{

    return 1;

}

/* code section */

int main()

{

    extern FILE *yyin, *yyout;

    /* open the source file

    in read mode */

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

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

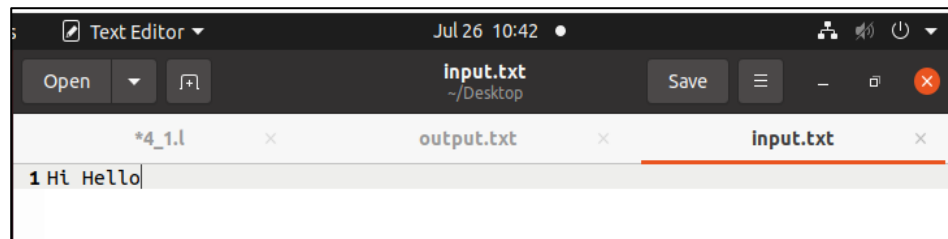
    yylex();

}

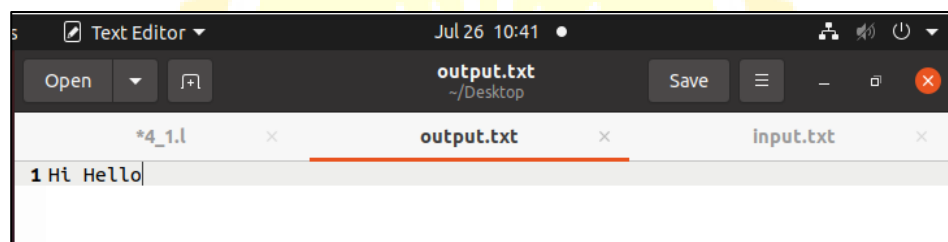
```

Output:


```
administrator@administrator-VirtualBox: ~/Desktop
(base) administrator@administrator-VirtualBox:~$ cd Desktop
(base) administrator@administrator-VirtualBox:~/Desktop$ lex 4_1.l
(base) administrator@administrator-VirtualBox:~/Desktop$ cc lex.yy.c -lfl
(base) administrator@administrator-VirtualBox:~/Desktop$ ./a.out
(base) administrator@administrator-VirtualBox:~/Desktop$
```



Input File



Output File

MBIT

b) WAP to implement ECHO, REJECT functions provided in Lex.

Code:

```
%{  
  
#include <stdio.h>  
  
void echo() {  
    printf("ECHO: %s\n", yytext);  
}  
%}  
  
DIGIT [0-9]  
LETTER [a-zA-Z]  
WHITESPACE [ \t\n]  
%%  
  
{DIGIT}+ { printf("NUMBER: %s\n", yytext); }  
{LETTER}+ { printf("WORD: %s\n", yytext); }  
{WHITESPACE} { /* Ignore whitespace */ }  
  
. { echo(); }  
%%  
  
int yywrap() {  
    return 1; // Indicate that there is no more input; used by Lex/Bison integration.}  
  
int main() {  
    yylex();  
    return 0;  
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ flex Practical4b.l
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ gcc lex.yy.c
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ ./a.out
I like to eat fruits very much
WORD: I
WORD: like
WORD: to
WORD: eat
WORD: fruits
WORD: very
WORD: much
```



c) WAP to implement BEGIN directive in a LEX program.

Code:

```
%{  
  
// Declarations and code to be executed before the rules  
  
#include <stdio.h>  
  
int line_count = 1; // Variable to keep track of line numbers  
  
%}  
  
%option noyywrap // Disable default EOF handling  
  
%%  
  
<<BEGIN>> {  
    // Code to be executed when encountering the "begin" directive  
    printf("Begin directive encountered at line %d\n", line_count);  
}  
  
. {  
    // Code to be executed for other tokens/rules  
  
    line_count++;  
  
    // Additional code for processing other tokens  
}  
  
%%  
  
int main() {  
  
    // Wrapper code for the Lex program  
  
    yylex(); // Invoke the generated lexer  
  
    return 0;
```

```
}
```

Output:

```
begin  
Begin directive encountered at line 1
```



Date: - __ / __ / ____

Practical-5

AIM: - a) Write a Lex program to count number of vowels and consonants in a given input string.

b) Write a Lex program to print out all numbers from the given file.

c) Write a Lex program to count the number of comment lines in a given C program.

Solution: -

a) Write a Lex program to count number of vowels and consonants in a given input string.

Code:

```
%{  
int vowel_count = 0;  
int consonant_count = 0;  
%}  
%%  
[a-zA-Z] {  
    if (strchr("aeiouAEIOU", yytext[0])) {  
        vowel_count++;  
    } else {  
        consonant_count++;  
    }  
}
```

```
    }  
    }  
    . ; // Ignore all other characters.  
%%  
  
int main() {  
    yylex();  
    printf("Vowel Count: %d\n", vowel_count);  
    printf("Consonant Count: %d\n", consonant_count);  
    return 0;  
}
```

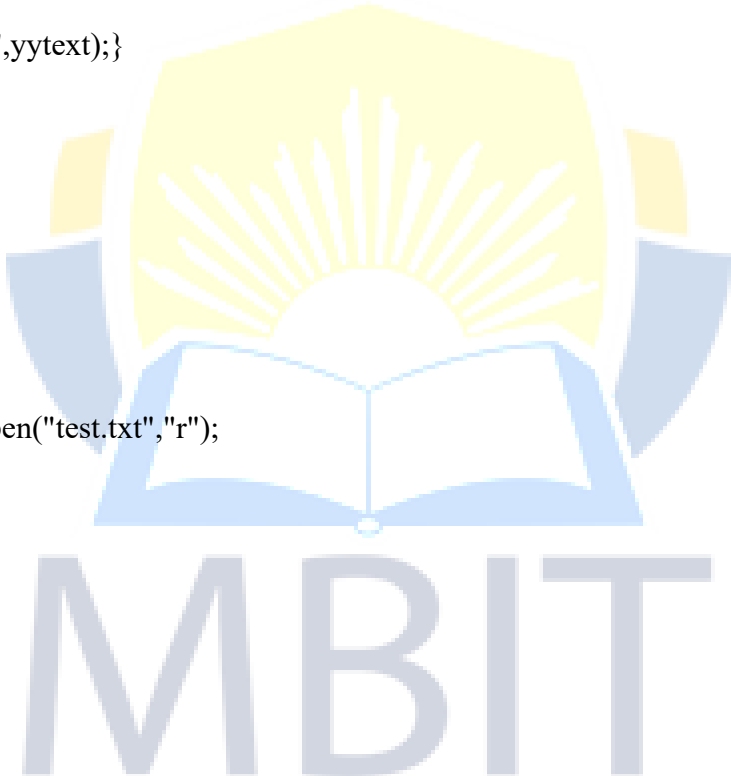
Output:

```
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ lex Practical5a.l  
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ cc lex.yy.c -ll  
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ ./a.out  
I want a car.  
  
Vowel Count: 4  
Consonant Count: 5
```

b) Write a Lex program to print out all numbers from the given file.

Code:

```
%{  
  
#include<stdio.h>  
  
%}  
  
%%  
  
[0-9]+ {printf("%s\n",yytext);}   
  
%%  
  
int yywrap() {  
  
int main()  
{  
FILE *input_file=fopen("test.txt","r");  
yyin= input_file;  
yylex();  
fclose(input_file);  
return 0;  
}
```

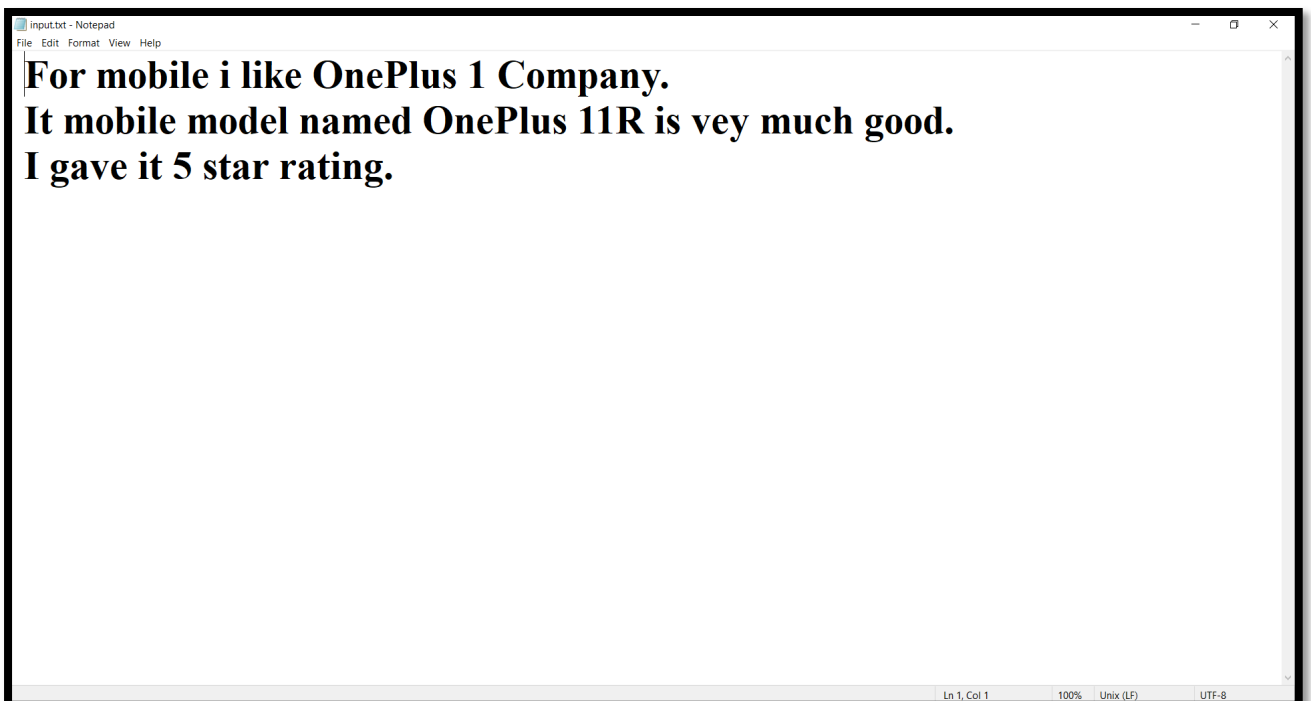


Output:

```
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ lex Practical5b.l
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ gcc lex.yy.c -o Practical5b -ll
(base) administrator@administrator-VirtualBox:~/Desktop/Practicals$ ./Practical5b input.txt
Number: 1

Number: 11

Number: 5
```



Input File

c) Write a Lex program to count the number of comment lines in a given C program.

Code:

```
%{  
  
#include<stdio.h>  
  
int nc=0;  
  
%}  
  
%%  
  
"/"[a-zA-Z0-9\n\t ]*"/" {nc++;}  
"//[a-zA-Z0-9\t ]*"\n" {nc++;}  
  
%%  
  
int main(int argc ,char* argv[])  
{  
    if(argc==2)  
    {  
        yyin=fopen(argv[1],"r");  
    }  
    else  
    {  
        printf("Enter the input\n");  
        yyin=stdin;  
    }  
  
    yyout=fopen("output.c","w");
```

```
yylex( );

printf("The number of comment lines=%d\n",nc);

fclose(yyin);

fclose(yyout);

}

int yywrap( )

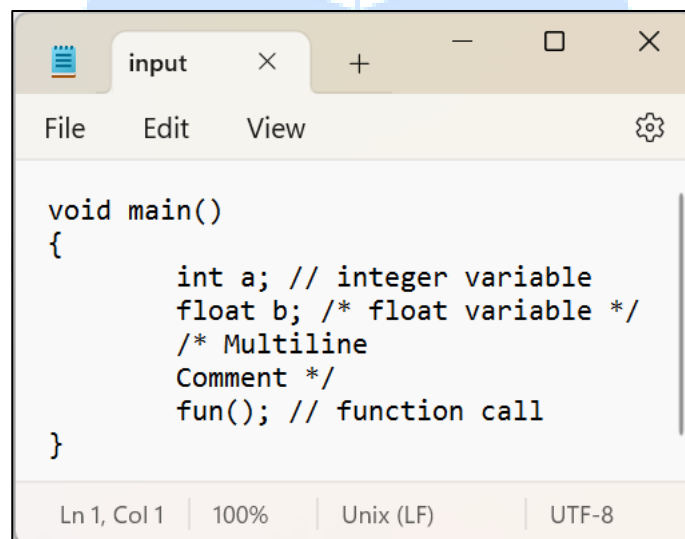
{

    return 1;

}
```

Output:

```
(base) administrator@administrator-VirtualBox:~/Desktop$ lex prac53.l
(base) administrator@administrator-VirtualBox:~/Desktop$ gcc lex.yy.c
(base) administrator@administrator-VirtualBox:~/Desktop$ ./a.out input.c
The number of comment lines=4
```



```
void main()
{
    int a; // integer variable
    float b; /* float variable */
    /* Multiline
    Comment */
    fun(); // function call
}
```

C Program File

Date: - __ / __ / ____

Practical-6

AIM: - a) WAP to implement unput and input.

b) WAP to implement yyterminate, yy_flush_buffer in LEX program.

c) WAP to implement yywrap in LEX program.

d) WAP to implement yymore and yyless in LEX program.

Solution: -

a) WAP to implement unput and input.

Code:

```
%{  
#include <stdio.h>  
%}  
%%  
[a-zA-Z]+ {  
    int i;  
    for (i = 1; i < yyleng; i++) {  
        unput(yytext[i]); // Unput the rest of the matched text  
    }  
    printf("Matched: %s\n", yytext);  
}
```

```

. {
    putchar(yytext[0]); // Output the current character
}

\n {
    // Handle newline character

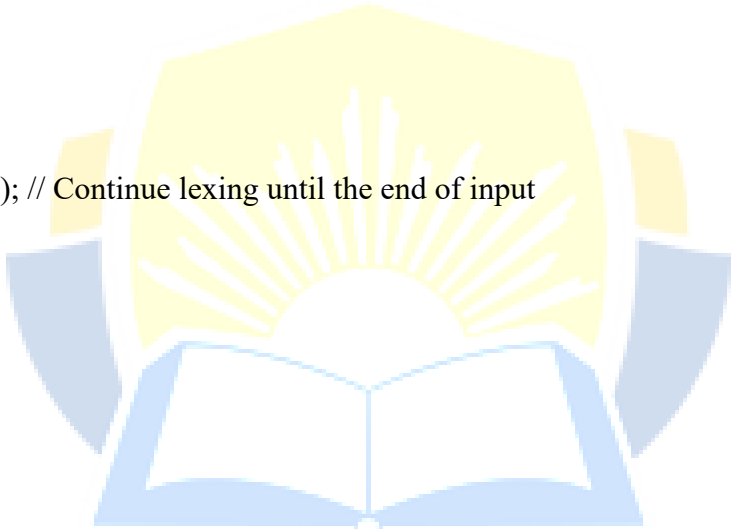
    putchar('\n');
}

%%

int main() {
    while (yylex() != 0); // Continue lexing until the end of input
    return 0;
}

```

Output:



```

ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ flex Practical6a.l
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ gcc -o lexer lex.yy.c -lfl
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ ./lexer
Hello
Matched: Helle

Matched: e111

Matched: 111

Matched: 11

Matched: 1

```

b) WAP to implement yyterminate, yy_flush_buffer in LEX program.

Code:

```
%{  
  
#include <stdio.h>  
  
%}  
  
%x FLUSH  
  
%%  
  
<FLUSH>[a-zA-Z]+ {  
    printf("Matched in FLUSH state: %s\n", yytext);  
}  
  
<FLUSH>. {  
    printf("Unmatched in FLUSH state: %c\n", yytext[0]);  
}  
  
<FLUSH>\n {  
    // Handle newline character in FLUSH state  
    putchar('\n');  
}  
  
[a-zA-Z]+ {  
    printf("Matched in INITIAL state: %s\n", yytext);  
}  
  
. {
```

```
printf("Unmatched in INITIAL state: %c\n", yytext[0]);
}

\n {
    // Handle newline character in INITIAL state

    putchar('\n');
}

<<EOF>> {
    printf("Lexer terminated.\n");
    break; // Terminate lexing
}

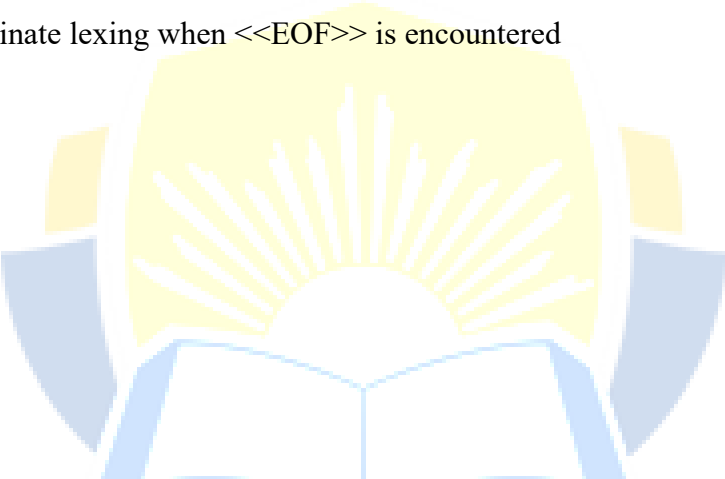
"<<FLUSH>>" {
    printf("Flushing input buffer.\n");
    BEGIN(FLUSH); // Enter FLUSH state
}

<INITIAL,FLUSH>{
    "<<FLUSH>>" {
        printf("Flushing input buffer.\n");
        BEGIN(FLUSH); // Enter FLUSH state
    }
}

<FLUSH>{
    "<<FLUSH>>" {
        printf("Flushing input buffer.\n");
        BEGIN(INITIAL); // Return to INITIAL state
    }
}
```

```
    }  
}  
%%  
  
int main() {  
    while (1) {  
        int token = yylex();  
        if (token == 0) {  
            break; // Terminate lexing when <<EOF>> is encountered  
        }  
    }  
    return 0;  
}
```

Output:

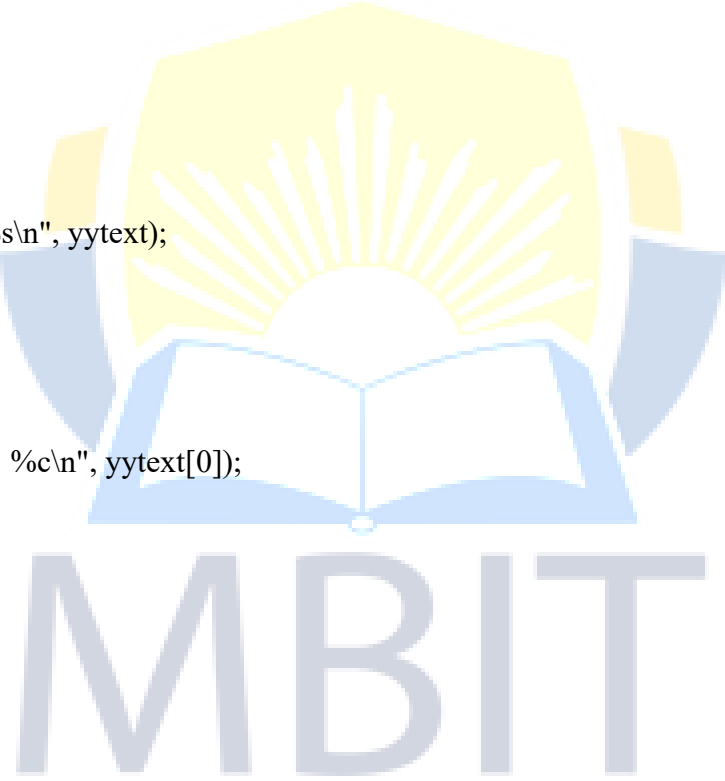


```
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ flex Practical6b.l  
Practical6b.l:52: warning, rule cannot be matched  
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ gcc -o lexer lex.yy.c -lfl  
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ ./lexer  
Hello World  
Matched in INITIAL state: Hello  
Unmatched in INITIAL state:  
Matched in INITIAL state: World
```


c) WAP to implement yywrap in LEX program.

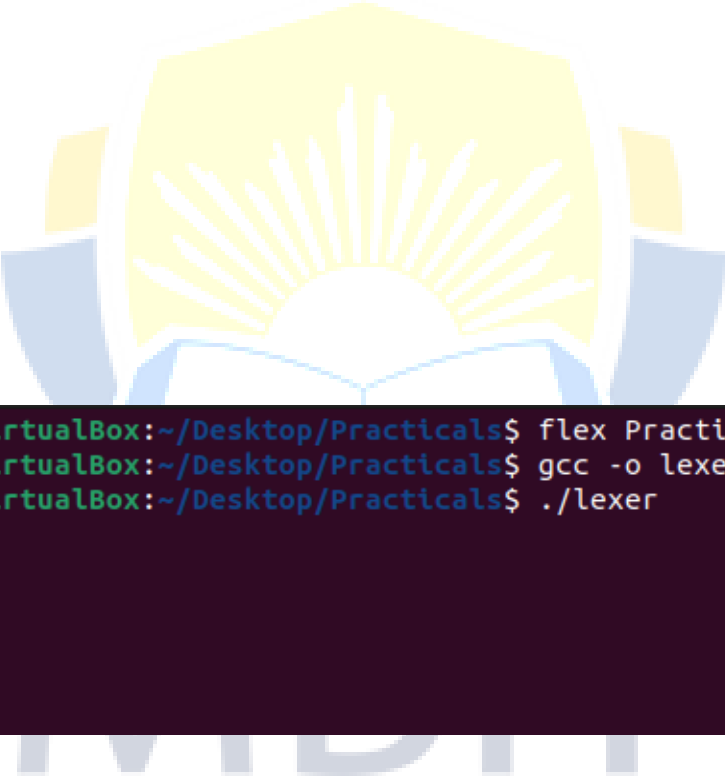
Code:

```
%{  
  
#include <stdio.h>  
  
%}  
  
%%  
  
[a-zA-Z]+ {  
    printf("Matched: %s\n", yytext);  
}  
  
. {  
    printf("Unmatched: %c\n", yytext[0]);  
}  
  
\n {  
    putchar('\n');  
}  
  
<<EOF>> {  
  
    if (yywrap()) {  
        printf("Lexer terminated.\n");  
  
        return 0; // Terminate lexing when EOF is encountered and yywrap() returns 1  
    }  
}
```



```
%%  
  
int yywrap() {  
    // Implement yywrap to return 1 when there is no more input to process  
  
    // In this example, we return 1 to indicate EOF  
  
    return 1;  
}  
  
int main() {  
    yylex();  
    return 0;  
}
```

Output:

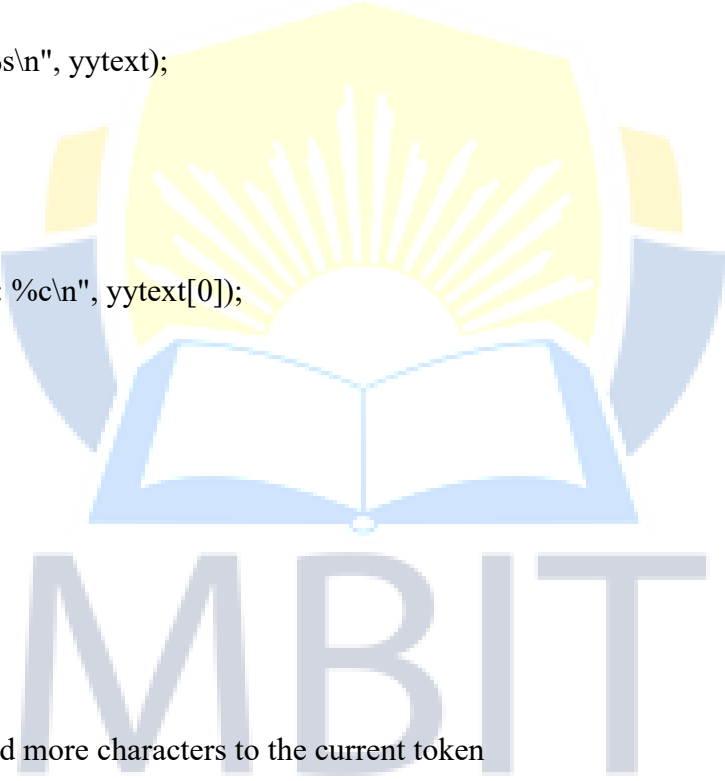


```
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ flex Practical6c.l  
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ gcc -o lexer lex.yy.c -lfl  
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ ./lexer  
Hello INDIA!!  
Matched: Hello  
Unmatched:  
Matched: INDIA  
Unmatched: !  
Unmatched: !
```

d) WAP to implement yymore and yyless in LEX program.

Code:

```
%{  
  
#include <stdio.h>  
  
%}  
  
%%  
  
[a-zA-Z]+ {  
    printf("Matched: %s\n", yytext);  
}  
  
. {  
    printf("Unmatched: %c\n", yytext[0]);  
}  
  
\n {  
    putchar('\n');  
}  
  
"+" {  
    yymore(); // Append more characters to the current token  
}  
  
"-" {  
    yyless(1); // Truncate the current token by one character  
}  
  
<<EOF>> {
```

A large, light blue watermark logo is centered in the background. It features a stylized open book at the base, with a yellow sunburst or fan-like shape above it. The letters 'MBIT' are written in a large, bold, sans-serif font across the middle of the logo.

```
printf("Lexer terminated.\n");

return 0; // Terminate lexing when EOF is encountered

}

%%

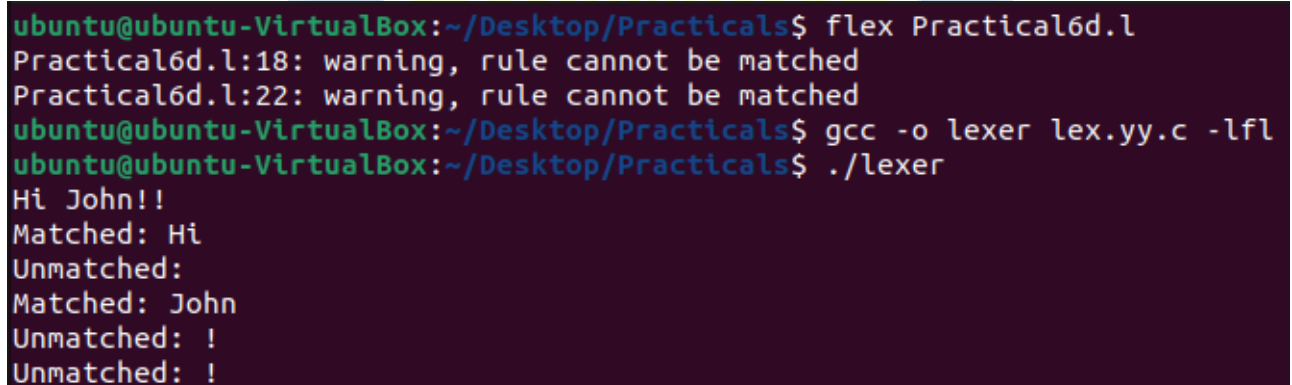
int main() {

    yylex();

    return 0;

}
```

Output:



```
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ flex Practical6d.l
Practical6d.l:18: warning, rule cannot be matched
Practical6d.l:22: warning, rule cannot be matched
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ gcc -o lexer lex.yy.c -lfl
ubuntu@ubuntu-VirtualBox:~/Desktop/Practicals$ ./lexer
Hi John!!
Matched: Hi
Unmatched:
Matched: John
Unmatched: !
Unmatched: !
```

Date: - __ / __ / ____

Practical-7

AIM: - WAP to Find the “First” set

Input: The string consists of grammar symbols.

Output: The First set for a given string.

Explanation:

The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the First set of the given string.

Solution: -

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define MAX_RULES 10
#define MAX_SYMBOLS 10

// Structure to represent a grammar rule

struct Rule {
    char nonTerminal;
    char production[MAX_SYMBOLS];
};
```

```

// Function to calculate the First set for a given string

void findFirstSet(char inputString[], struct Rule rules[], int numRules) {

    int i, j;

    bool isTerminal = false;

    char firstSet[MAX_SYMBOLS] = {'\0'};

    for (i = 0; i < strlen(inputString); i++) {

        isTerminal = true;

        for (j = 0; j < numRules; j++) {

            if (inputString[i] == rules[j].nonTerminal) {

                isTerminal = false;

                char tempFirstSet[MAX_SYMBOLS] = {'\0'};

                int k;

                // Find the First set for the production

                for (k = 0; rules[j].production[k] != '\0'; k++) {

                    if (rules[j].production[k] >= 'a' && rules[j].production[k] <= 'z') {

                        // Terminal symbol

                        tempFirstSet[0] = rules[j].production[k];

                        break;

                    } else {

                        // Non-terminal symbol, recursively find its First set

                        char nonTerminal[2] = {rules[j].production[k], '\0'};

                        findFirstSet(nonTerminal, rules, numRules);

                        // Merge First set of the non-terminal into tempFirstSet

                        strcat(tempFirstSet, firstSet);
                    }
                }
            }
        }
    }
}

```

```

    }

}

// Merge tempFirstSet into the overall First set

strcat(firstSet, tempFirstSet);

}

}

if (isTerminal) {

    // Terminal symbol, add it to the First set

    char terminal[2] = {inputString[i], '\0'};

    strcat(firstSet, terminal);

    break;

}

}

printf("First set for %s: {%s}\n", inputString, firstSet);

}

int main() {

    int numRules;

    printf("Enter the number of rules: ");

    scanf("%d", &numRules);

    struct Rule rules[MAX_RULES];

    for (int i = 0; i < numRules; i++) {

        printf("Enter rule %d (non-terminal production): ", i + 1);

        scanf(" %c %s", &rules[i].nonTerminal, rules[i].production);

    }

```

```
char inputString[MAX_SYMBOLS];

printf("Enter the string to find the First set: ");

scanf("%s", inputString);

findFirstSet(inputString, rules, numRules);

return 0;

}
```

Output:

```
Enter the number of rules: 3
Enter rule 1 (non-terminal production): S aAB
Enter rule 2 (non-terminal production): A b
Enter rule 3 (non-terminal production): B c
Enter the string to find the First set: S
First set for S: {a}

Process returned 0 (0x0)   execution time : 30.188 s
Press any key to continue.
```

MBIT

Date: - __ / __ / ____

Practical-8

AIM: - WAP to Find the “Follow” set.

Input: The string consists of grammar symbols.

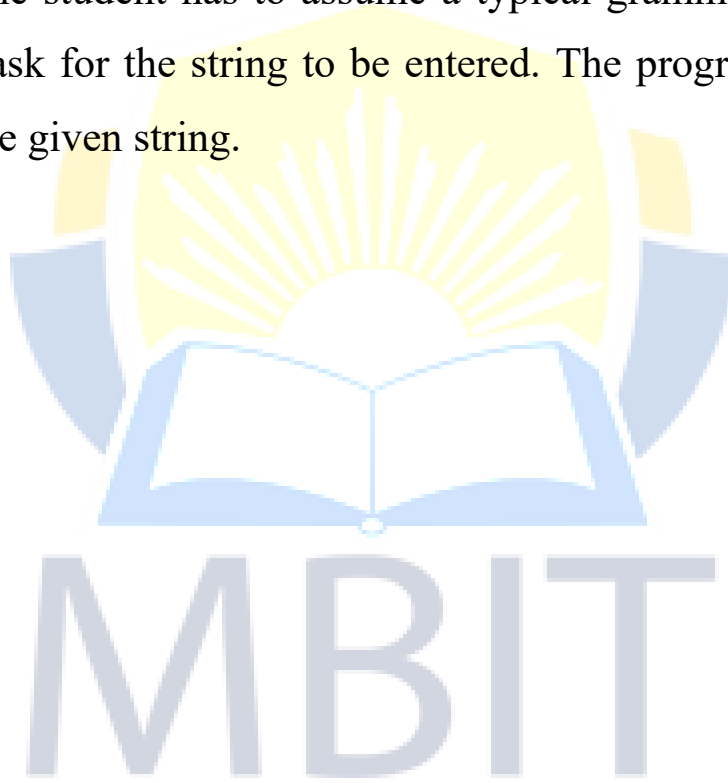
Output: The Follow set for a given string.

Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the Follow set of the given string.

Solution: -

Code:

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
int main()
{
    int i,z;
    char c,ch;
    printf("Enter the no.of productions:");
    scanf("%d",&n);
```



```
printf("Enter the productions(epsilon=$):\n");

for(i=0;i<n;i++)

    scanf("%s%c",a[i],&ch);

do

{

    m=0;

    printf("Enter the element whose FOLLOW is to be found:");

    scanf("%c",&c);

    follow(c);

    printf("FOLLOW(%c) = { ",c);

    for(i=0;i<m;i++)

        printf("%c ",f[i]);

    printf(" }\n");

    printf("Do you want to continue(0/1)?");

    scanf("%d%c",&z,&ch);

}

while(z==1);

}

void follow(char c)

{

    if(a[0][0]==c)f[m++]='$';

    for(i=0;i<n;i++)

    {

        for(j=2;j<strlen(a[i]);j++)
```

```

{
    if(a[i][j]==c)
    {
        if(a[i][j+1]!='\0')first(a[i][j+1]);
        if(a[i][j+1]=='\0'&& c!=a[i][0])
            follow(a[i][0]);
    }
}
}
}

void first(char c)
{
    int k;

    if(!(isupper(c)))f[m++]=c;

    for(k=0;k<n;k++)
    {
        if(a[k][0]==c)
        {
            if(a[k][2]=='$') follow(a[i][0]);

            else if(islower(a[k][2]))f[m++]=a[k][2];

            else first(a[k][2]);
        }
    }
}
}

```

Output:

```
Enter the no.of productions:5
Enter the productions(epsilon=$):
S=ABCD
A=b
B=c
C=d
D=e
Enter the element whose FOLLOW is to be found:C
FOLLOW(C) = { e }
Do you want to continue(0/1)?0

Process returned 0 (0x0)   execution time : 66.884 s
Press any key to continue.
```



Date: - __ / __ / ____

Practical-9

AIM: - Construct a recursive descent parser for a given grammar.

Solution: -

Code:

```
#include <stdio.h>

#include <string.h>

#define SUCCESS 1
#define FAILED 0

// Function prototypes
int E(), Edash(), T(), Tdash(), F();

const char *cursor;

char string[64];

int main() {
    puts("Enter the string");
    scanf("%s", string); // Read input from the user

    cursor = string;

    puts("");

    puts("Input Action");

    puts("      ");

    // Call the starting non-terminal E

    if (E() && *cursor == '\0')
```

```

{
    // If parsing is successful and the cursor has reached the end
    puts(" ");
    puts("String is successfully parsed");
    return 0;
}

else
{
    puts("      ");
    puts("Error in parsing String");
    return 1;
}
}

// Grammar rule: E -> T E'

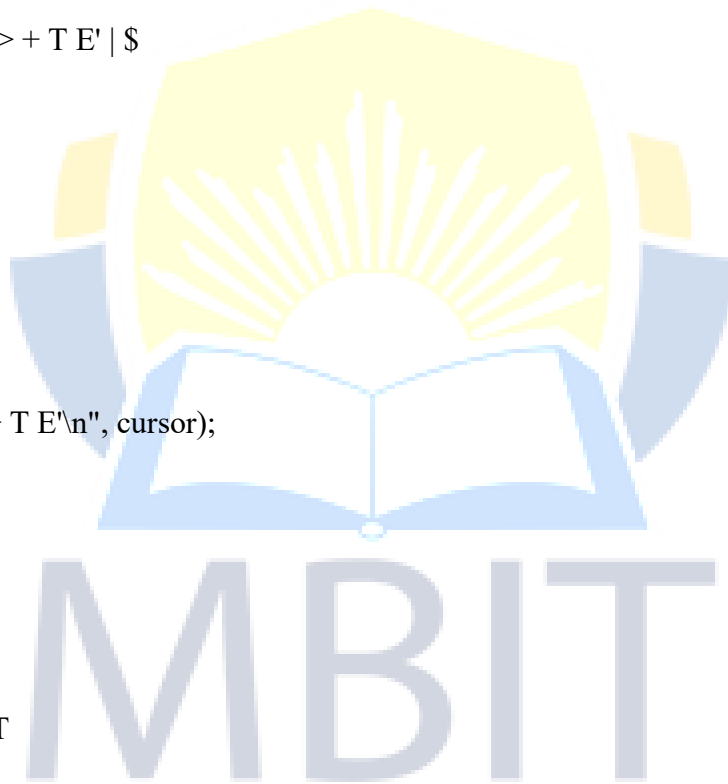
int E() {
    printf("%-16s E -> T E'\n", cursor);
    if (T())
    {
        // Call non-terminal T
        if (Edash())

        // Call non-terminal E'
        return SUCCESS;
    }

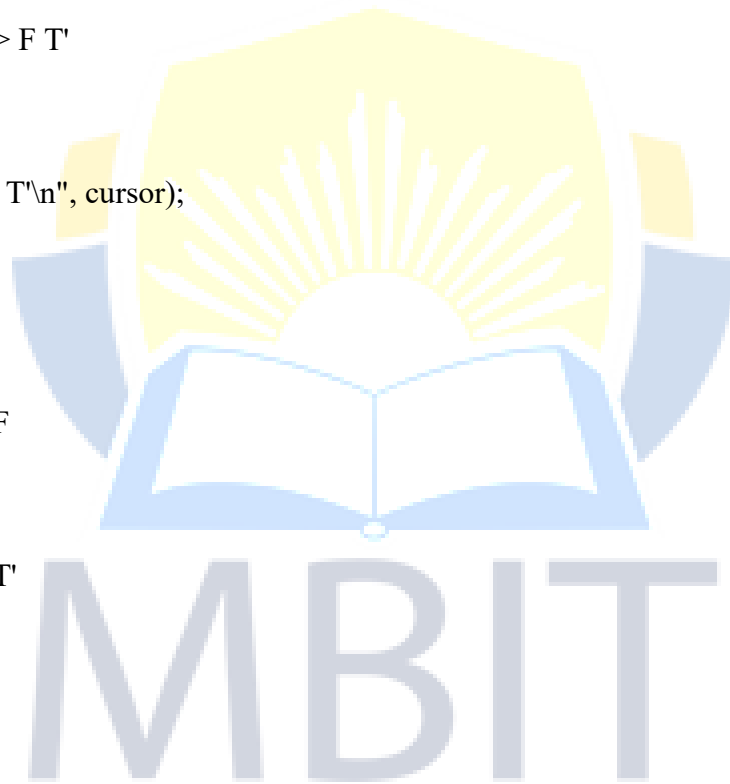
    else if

```

```
{  
    return FAILED;  
}  
  
else  
  
{  
    return FAILED;  
}  
  
// Grammar rule: E' -> + T E' | $  
  
int Edash()  
{  
    if (*cursor == '+')  
    {  
        printf("%-16s E' -> + T E'\n", cursor);  
        cursor++;  
        if (T())  
        {  
            // Call non-terminal T  
            if (Edash())  
            {  
                // Call non-terminal E'  
                return SUCCESS;  
            }  
            else if  
            {  
                return FAILED;  
            }  
        }  
    }  
}
```



```
}  
  
else  
  
{  
  
return FAILED;  
  
}  
  
printf("%-16s E' -> $\n", cursor);  
  
return SUCCESS;  
  
// Grammar rule: T -> F T'  
  
int T() {  
printf("%-16s T -> F T'\n", cursor);  
if (F())  
{  
// Call non-terminal F  
if (Tdash())  
// Call non-terminal T'  
return SUCCESS;  
else if  
{  
return FAILED;  
}  
else  
{  
return FAILED;  
}  
}
```




```
// Grammar rule: T' -> * F T' | $

int Tdash()

{
    if (*cursor == '*')
    {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F())
        {
            // Call non-terminal F
            if (Tdash()) //
            Call non-terminal T'
            return SUCCESS;
        else if
        {
            return FAILED;
        }
        else
        {
            printf("%-16s T' -> $\n", cursor);
            return SUCCESS;
        }
    }

    // Grammar rule: F -> ( E ) |
```



```
i int F()

{
if (*cursor == '(')
{
printf("%-16s F -> ( E )\n", cursor);

cursor++;

if (E())
{
// Call non-terminal E
if (*cursor == ')')
{
cursor++;
return SUCCESS;
}
else
{
return FAILED;
}
}
else if (*cursor == 'i')
{
printf("%-16s F -> i\n", cursor);

cursor++;

return SUCCESS;
}
```



else

{

return FAILED;

}

Output:

```
i+(i+i)*i

Input      Action
-----
i+(i+i)*i  E -> T E'
i+(i+i)*i  T -> F T'
i+(i+i)*i  F -> i
+(i+i)*i   T' -> $
+(i+i)*i   E' -> + T E'
(i+i)*i    T -> F T'
(i+i)*i    F -> ( E )
i+i)*i     E -> T E'
i+i)*i     T -> F T'
i+i)*i     F -> i
+i)*i      T' -> $
+i)*i      E' -> + T E'
i)*i       T -> F T'
i)*i       F -> i
)*i        T' -> $
)*i        E' -> $
*i         T' -> * F T'
i          F -> i
           T' -> $
           E' -> $
-----
String is successfully parsed

-----
Process exited after 3.774 seconds with return value 0
Press any key to continue . . . |
```

Date: - __ / __ / ____

Practical-10

AIM: - Write a C program for constructing of LL (1) parsing.

Solution: -

Code:

```
#include<stdio.h>

#include<ctype.h>

#include<string.h>

void followfirst(char , int , int);

void findfirst(char , int , int);

void follow(char c);

int count,n=0;

char calc_first[10][100];

char calc_follow[10][100];

int m=0;

char production[10][10], first[10];

char f[10];

int k;

char ck;

int e;

int main(int argc,char **argv)

{
```

```
int jm=0;

int km=0;

int i,choice;

char c,ch;

printf("How many productions ? :");

scanf("%d",&count);

printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",count);

for(i=0;i<count;i++)

{

scanf("%s%c",production[i],&ch);

}

int kay;

char done[count];

int ptr = -1;

for(k=0;k<count;k++)

{

for(kay=0;kay<100;kay++)

{

calc_first[k][kay] = '!';

}

}

int point1 = 0, point2, xxx;

for(k=0;k<count;k++)

{
```



```
c=production[k][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

findfirst(c,0,0);

ptr+=1;

done[ptr] = c;

printf("\n First(%c)= { ",c); calc_first[point1][point2++] = c; for(i=0+jm;i<n;i++)

{

int lark = 0,chk = 0;

for(lark=0;lark<point2;lark++)

{

if (first[i] == calc_first[point1][lark])

{

chk = 1;

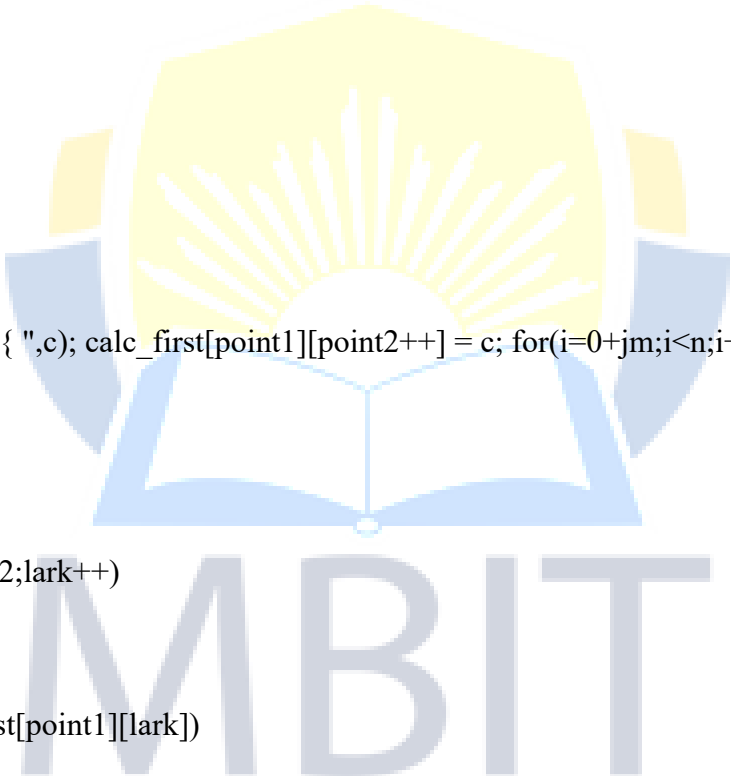
break;

}

}

if(chk == 0)

{
```



```
printf("%c, ",first[i]);

calc_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm=n;

point1++;

}

printf("\n");

printf("\n\n");

char donee[count];

ptr = -1;

for(k=0;k<count;k++)

{

    for(kay=0;kay<100;kay++)

    {

        calc_follow[k][kay] = '!';

    }

}

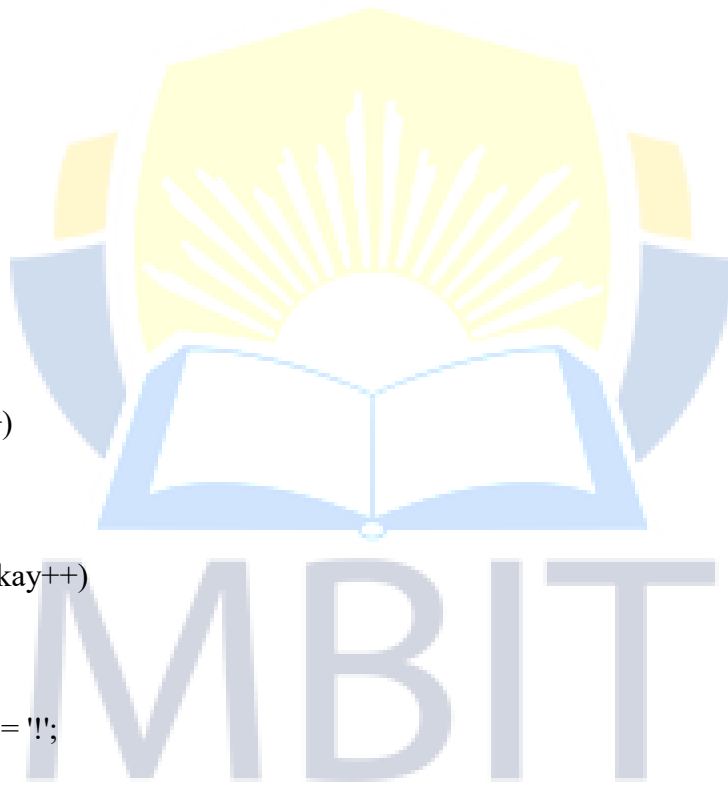
point1 = 0;

int land = 0;

for(e=0;e<count;e++)

{

    ck=production[e][0];
```



```
point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == donee[kay])

xxx = 1;

if (xxx == 1) continue;

land += 1;

follow(ck);

ptr+=1;

donee[ptr] = ck;

printf(" Follow(%c) = { ",ck); calc_follow[point1][point2++] = ck; for(i=0+km;i<m;i++)

{

int lark = 0,chk = 0;

for(lark=0;lark<point2;lark++)

{

if (f[i] == calc_follow[point1][lark])

{

chk = 1;

break;

}

}

if(chk == 0)

{

printf("%c, ",f[i]);
```



```
calc_follow[point1][point2++] = f[i];

}

}

printf(" } \n\n");

km=m;

point1++;

}

char ter[10];

for(k=0;k<10;k++)

{

ter[k] = '!';

}

int ap, vp, sid = 0;

for(k=0;k<count;k++)

{

for(kay=0;kay<count;kay++)

{

if(!isupper(production[k][kay]) && production[k][kay] != '#' && production[k][kay] != '=' &&

production[k][kay] != '\0')

{

vp = 0;

for(ap = 0; ap < sid; ap++)

{

if(production[k][kay] == ter[ap])
```

```
{
vp = 1;

break;

}

}

if(vp == 0)

{

ter[sid] = production[k][kay];

sid ++;

}

}

}

}

ter[sid] = '$';

sid++;

printf("\n\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");

printf("\n\t\t\t\t\t ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");

printf("\n\t\t\t=====

=====\\n");

printf("\t\t\t\t\t");

for(ap = 0;ap < sid; ap++)

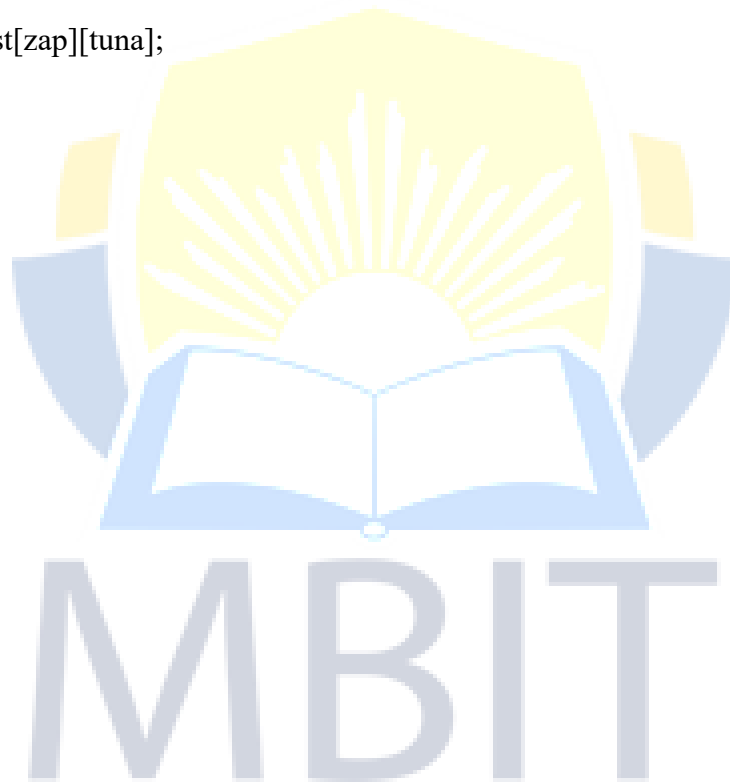
{

printf("%c\t\t",ter[ap]);

}
```

```
for(zap=0;zap<count;zap++)
```

```
{  
if(calc_first[zap][0] == production[ap][k])  
  
{  
for(tuna=1;tuna<100;tuna++)  
  
{  
if(calc_first[zap][tuna] != '!')  
  
{  
tem[ct++] = calc_first[zap][tuna];  
}  
else  
{  
break;  
}  
}  
tem[ct++] = '_';  
}  
k++;  
}  
  
int zap = 0,tuna;  
  
for(tuna = 0;tuna<ct;tuna++)  
  
{  
  
if(tem[tuna] == '#')  
  
{  
  
zap = 1;
```

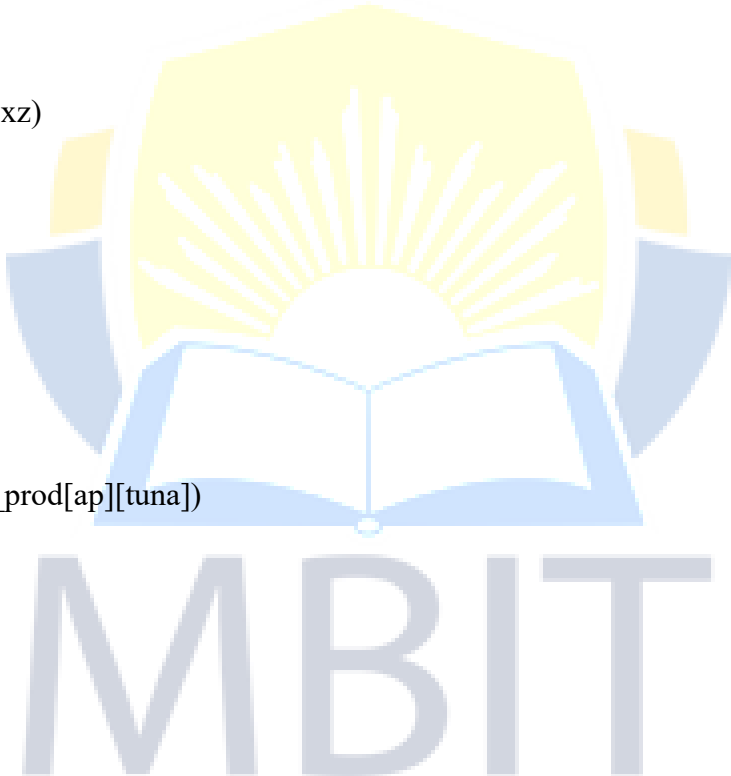


```
}  
  
else if(tem[tuna] == '_')  
  
{  
  
if(zap == 1)  
  
{  
  
zap = 0;  
  
}  
  
else  
  
{  
  
break;  
  
}  
  
first_prod[ap][destiny++] = tem[tuna];  
  
}  
  
}  
  
}  
  
char table[land][sid+1];  
  
ptr = -1;  
  
for(ap = 0; ap < land ; ap++)  
  
{  
  
for(kay = 0; kay < (sid + 1) ; kay++)  
  
{  
  
table[ap][kay] = '!';  
  
}  
  
}
```

```
for(ap = 0; ap < count ; ap++)  
{  
    ck = production[ap][0];  
    xxx = 0;  
    for(kay = 0; kay <= ptr; kay++)  
        if(ck == table[kay][0])  
            xxx = 1;  
    if (xxx == 1)  
        continue;  
    else  
    {  
        ptr = ptr + 1;  
        table[ptr][0] = ck;  
    }  
}  
for(ap = 0; ap < count ; ap++)  
{  
    int tuna = 0;  
    while(first_prod[ap][tuna] != '\0')  
    {  
        int to,ni=0;  
        for(to=0;to<sid;to++)  
        {  
            if(first_prod[ap][tuna] == ter[to])
```



```
{  
  
ni = 1;  
  
}  
  
}  
  
if(ni == 1)  
  
{  
  
char xz = production[ap][0];  
  
int cz=0;  
while(table[cz][0] != xz)  
{  
  
cz = cz + 1;  
  
}  
  
int vz=0;  
while(ter[vz] != first_prod[ap][tuna])  
  
{  
  
vz = vz + 1;  
  
}  
table[cz][vz+1] = (char)(ap + 65);  
  
}  
  
tuna++;  
  
}  
  
}  
  
for(k=0;k<sid;k++)  
  
{
```



```
for(kay=0;kay<100;kay++)
{
if(calc_first[k][kay] == '!')
{
break;
}

else if(calc_first[k][kay] == '#')
{
int fz = 1; while(calc_follow[k][fz] != '!')
{
char xz = production[k][0];
int cz=0; while(table[cz][0] != xz)
{
cz = cz + 1;
}
int vz=0;
while(ter[vz] != calc_follow[k][fz])
{
vz = vz + 1;
}
table[k][vz+1] = '#'; fz++;
}
break;
}
```



```
}  
  
}  
  
for(ap = 0; ap < land ; ap++)  
  
{  
  
printf("\t\t\t %c\t\t",table[ap][0]);  
  
for(kay = 1; kay < (sid + 1) ; kay++)  
  
{  
  
if(table[ap][kay] == '!')  
  
printf("\t\t");  
  
else if(table[ap][kay] == '#')  
  
printf("%c=#\t\t",table[ap][0]);  
  
else  
  
{  
  
int mum = (int)(table[ap][kay]);  
  
mum -= 65;  
  
printf("%s\t\t",production[mum]);  
  
}  
  
}  
  
printf("\n");  
  
printf("\t\t\t");  
  
printf("\n");  
  
}  
  
int j;  
  
printf("\n\nPlease enter the desired INPUT STRING = ");
```

```

char input[100];

scanf("%s%c",input,&ch);

printf("\n\t\t\t\t\t=====
=====\\n");

printf("\t\t\t\t\tStack\t\t\tInput\t\t\tAction");
printf("\n\t\t\t\t\t=====
=====\\n");

int i_ptr = 0,s_ptr = 1;

char stack[100]; s
tack[0] = '$';
stack[1] = table[0][0];
while(s_ptr != -1)
{
printf("\t\t\t\t\t");

int vamp = 0;
for(vamp=0;vamp<=s_ptr;vamp++)
{
printf("%c",stack[vamp]);

}

printf("\t\t\t\t\t");

vamp = i_ptr;

while(input[vamp] != '\0')
{

printf("%c",input[vamp]);

```

```
vamp++;  
  
}  
  
printf("\t\t\t");  
  
char her = input[i_ptr];  
  
char him = stack[s_ptr];  
  
s_ptr--; if(!isupper(him))  
  
{  
  
if(her == him)  
  
{  
  
i_ptr++;  
  
printf("POP ACTION\n");  
  
}  
  
else  
  
{  
  
printf("\nString Not Accepted by LL(1) Parser !!\n"); exit(0);  
  
}  
  
else  
  
{  
  
for(i=0;i<sid;i++)  
  
{  
  
if(ter[i] == her) break;  
  
}  
  
char produ[100];  
  
for(j=0;j<land;j++)
```

```
{
    if(him == table[j][0])
    {
        if (table[j][i+1] == '#')
        {
            printf("%c=#\n",table[j][0]);
            produ[0] = '#';
            produ[1] = '\0';
        }
        else if(table[j][i+1] != '!')
        {
            int mum = (int)(table[j][i+1]);
            mum -= 65;
            strcpy(produ,production[mum]);
            printf("%s\n",produ);
        }
        else
        {
            printf("\nString Not Accepted by LL(1) Parser !!\n");
            exit(0);
        }
    }
}

int le = strlen(produ);
```

```
le = le - 1;

if(le == 0)

{

continue;

for(j=le;j>=2;j--)

{

s_ptr++;

stack[s_ptr] = produ[j];

}

}

printf("\n\t\t\t=====\\n");

if (input[i_ptr] == '\0')

{

printf("\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\\n");

}

else

printf("\n\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\\n");

printf("\t\t\t\t\t=====\\n");

}

void follow(char c)

{
```

```
int i ,j; if(production[0][0]==c)

{

    f[m++]='$';

}

for(i=0;i<10;i++)

{

    for(j=2;j<10;j++)

    {

        if(production[i][j]==c)

        {

            if(production[i][j+1]!='\0')

            {

                followfirst(production[i][j+1],i,(j+2));

            }

            if(production[i][j+1]=='\0'&& c!=production[i][0])

            {

                follow(production[i][0]);

            }

        }

    }

}

}

}

void findfirst(char c ,int q1 , int q2)

{
```

```
int j;

if(!(isupper(c)))

{

first[n++]=c;

}

for(j=0;j<count;j++)

{

if(production[j][0]==c)

{

if(production[j][2]=='#')

{

if(production[q1][q2] == '\0')

first[n++]='#';

else if(production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))

{

findfirst(production[q1][q2], q1, (q2+1));

}

else first[n++]='#';

}

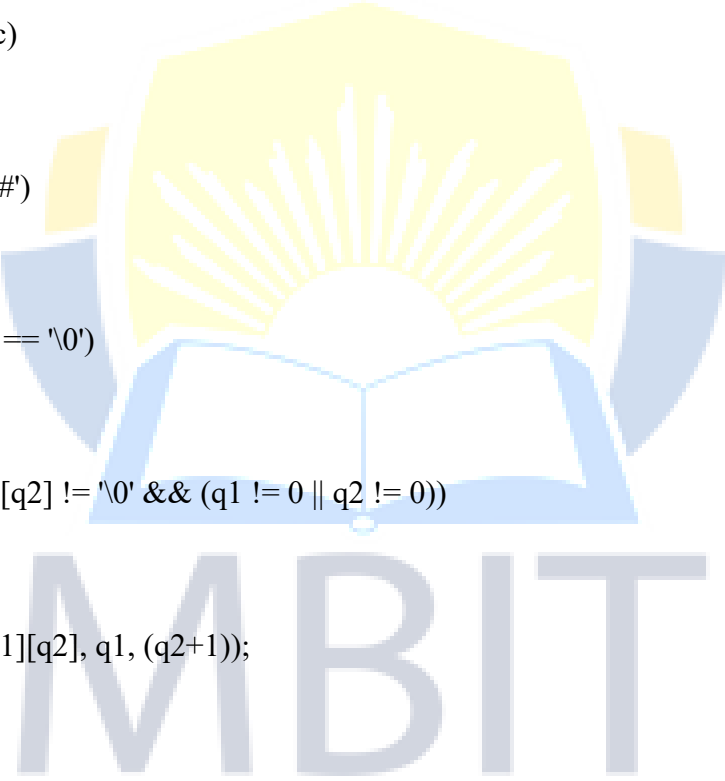
else if(!isupper(production[j][2]))

{

first[n++]=production[j][2];

}

else
```



```
{  
findfirst(production[j][2], j, 3);
```

```
}
```

```
}
```

```
}
```

```
}
```

```
void followfirst(char c, int c1 , int c2)
```

```
{
```

```
int k; if(!(isupper(c))) f[m++] = c;
```

```
else
```

```
{
```

```
int i=0,j=1;
```

```
for(i=0;i<count;i++)
```

```
{
```

```
if(calc_first[i][0] == c)
```

```
break;
```

```
}
```

```
while(calc_first[i][j] != '!')
```

```
{
```

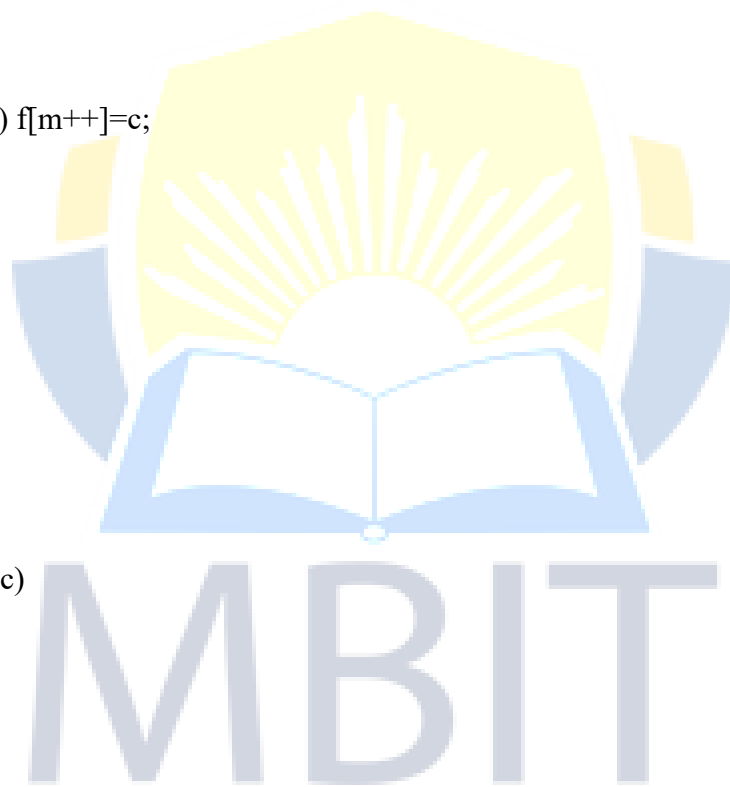
```
if(calc_first[i][j] != '#')
```

```
{
```

```
f[m++] = calc_first[i][j];
```

```
}
```

```
else{
```




```
if(production[c1][c2] == '\0')  
{  
  follow(production[c1][0]);  
}  
else  
{  
  followfirst(production[c1][c2],c1,c2+1);  
}  
}  
j++;  
}  
}  
}
```



Output:

```
How many productions ? :8

Enter 8 productions in form A=B where A and B are grammar symbols :

E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
F=(E)
F=i

First(E)= { (, i, }
First(R)= { +, #, }
First(T)= { (, i, }
First(Y)= { *, #, }
First(F)= { (, i, }

-----

Follow(E) = { $, ), }
Follow(R) = { $, ), }
Follow(T) = { +, $, ), }
Follow(Y) = { +, $, ), }
Follow(F) = { *, +, $, ), }
```

The LL(1) Parsing Table for the above grammar :-

		+	*	()	i	\$
E				E=TR		E=TR	
R		R=+TR			R=#		R=#
T				T=FY		T=FY	
Y		Y=#	Y=*FY		Y=#		Y=#
F				F=(E)		F=i	

```
Please enter the desired INPUT STRING = i+i+i$

=====
Stack      Input      Action
=====
$E          i+i+i$      E=TR
$RT          i+i+i$      T=FY
$RYF         i+i+i$      F=i
$RYi         i+i+i$      POP ACTION
$RY          i+i+i$      Y=#
$R           i+i+i$      R=+TR
$RT+         i+i+i$      POP ACTION
$RT          i+i$      T=FY
$RYF         i+i$      F=i
$RYi         i+i$      POP ACTION
$RY          i$      Y=+FY
$RYF+        i$      POP ACTION
$RYF         i$      F=i
$RYi         i$      POP ACTION
$RY          $      Y=#
$R           $      R=#
$            $      POP ACTION

=====
YOUR STRING HAS BEEN ACCEPTED !!
=====
```



Date: - __ / __ / ____

Practical-11

AIM: - Implement a C program to implement operator precedence parsing.

Solution: -

Code:

```
#include<stdio.h>

#include<stdlib.h>

// Define the Basic_tree struct
struct Basic_tree
{
    char data;

    struct Basic_tree* lptr;
    struct Basic_tree* rptr;
};

typedef struct Basic_tree node;

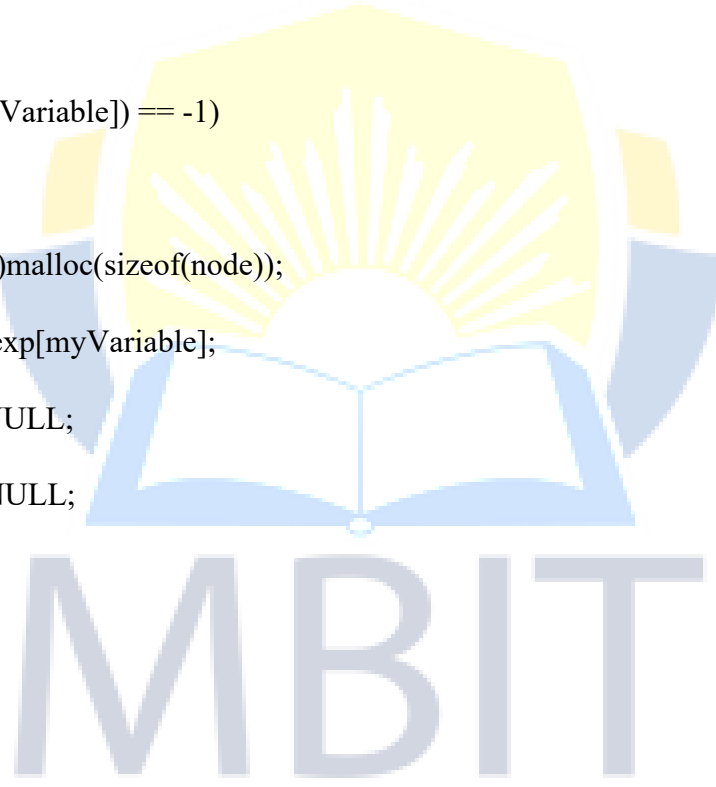
// Define a typedef for convenience
int Operator_Position_Finder(char);

int isOperator(char);

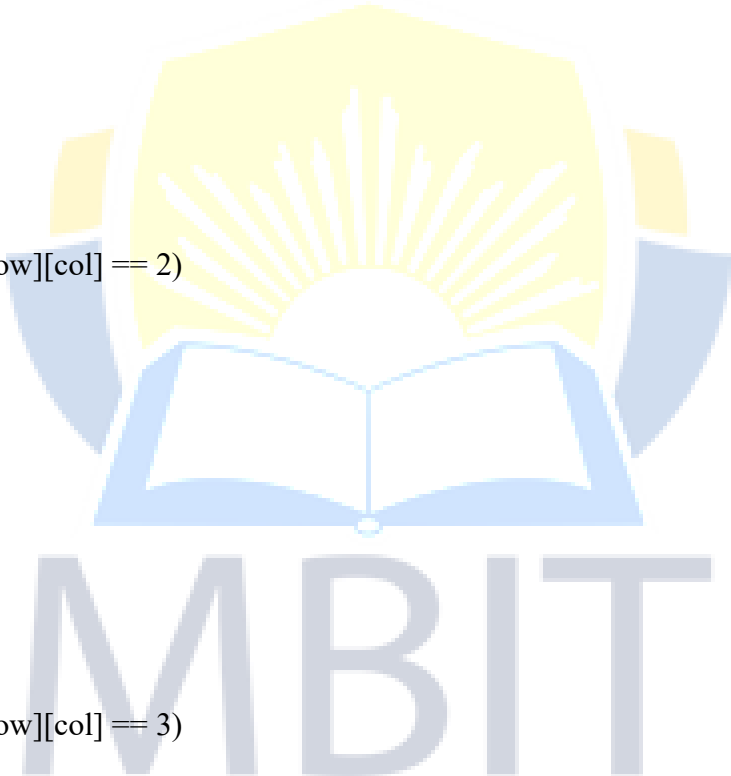
int T4Tutorials[5][5] =
{
    {1, 0, 0, 1, 1},
```

```
{1, 1, 0, 1, 1},  
  
{0, 0, 0, 2, 3},  
  
{1, 1, 3, 1, 1},  
  
{0, 0, 0, 3, 2}  
  
};  
  
int tos = -1; struct opr  
  
{  
  
char op_name;  
  
node* t;  
  
}  
  
oprate[50];  
  
char current_operators[5] = {'+', '*', '(', ')', '['};  
char OperatorsInStack[5] = {'+', '*', '(', ')', '['};  
  
void DisplayTree(node*);  
  
void T4Tutorials_value(void);  
  
int main()  
  
{  
  
char exp[10];  
  
int myVariable = 0, row = 0, col = 0;  
  
node* temp;  
  
printf("Enter Exp: ");  
  
scanf("%s", exp);  
  
T4Tutorials_value();  
  
while (exp[myVariable] != '\0')
```

```
{
if (myVariable == 0)
{
tos++;
oprte[tos].op_name = exp[tos];
}
else
{
if (isOperator(exp[myVariable]) == -1)
{
oprte[tos].t = (node*)malloc(sizeof(node));
oprte[tos].t->data = exp[myVariable];
oprte[tos].t->lptr = NULL;
oprte[tos].t->rptr = NULL;
}
else
{
row = Operator_Position_Finder(oprte[tos].op_name);
col = Operator_Position_Finder(exp[myVariable]);
if (T4Tutorials[row][col] == 0)
{
tos++;
oprte[tos].op_name = exp[myVariable];
}
}
```



```
else if (T4Tutorials[row][col] == 1)
{
temp = (node*)malloc(sizeof(node));
temp->data = oprate[tos].op_name;
temp->lptra = oprate[tos - 1].t;
temp->rptra = oprate[tos].t;
tos--;
oprate[tos].t = temp;
myVariable--;
}
else if (T4Tutorials[row][col] == 2)
{
temp = oprate[tos].t;
tos--;
oprate[tos].t = temp;
}
else if (T4Tutorials[row][col] == 3)
{
printf("\nExpression is Invalid...\n");
printf("%c %c can not occur simultaneously\n", oprate[tos].op_name, exp[myVariable]);
break;
}
}
}
```



```
myVariable++;

}

printf("Show tree\n\n\n");

DisplayTree(oprate[tos].t);

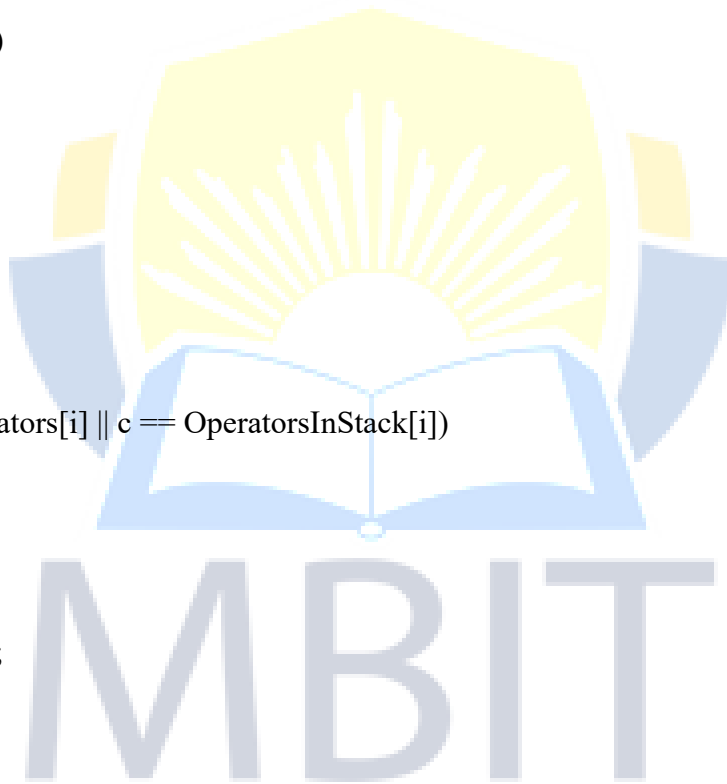
printf("Over\n");

return 0;

}

int isOperator(char c)
{
int i = 0;
for (i = 0; i < 5; i++)
{
if (c == current_operators[i] || c == OperatorsInStack[i])
break;
}
if (i == 5) return (-1);
else
return i;
}

int Operator_Position_Finder(char c)
{
int i;
for (i = 0; i < 5; i++)
{
```




```
if (c == current_operators[i] || c == OperatorsInStack[i])
```

```
break;
```

```
}
```

```
return i;
```

```
}
```

```
void DisplayTree(node* start)
```

```
{
```

```
if (start->lptr != NULL) DisplayTree(start->lptr);
```

```
if (start->rptr != NULL) DisplayTree(start->rptr);
```

```
printf("%c \n", start->data);
```

```
}
```

```
void T4Tutorials_value(void)
```

```
{
```

```
int i, j;
```

```
printf("OPERATOR PRECEDENCE Matrix\n");
```

```
printf("===== \n ");
```

```
for (i = 0; i < 5; i++)
```

```
{
```

```
printf("%c ", OperatorsInStack[i]);
```

```
}
```

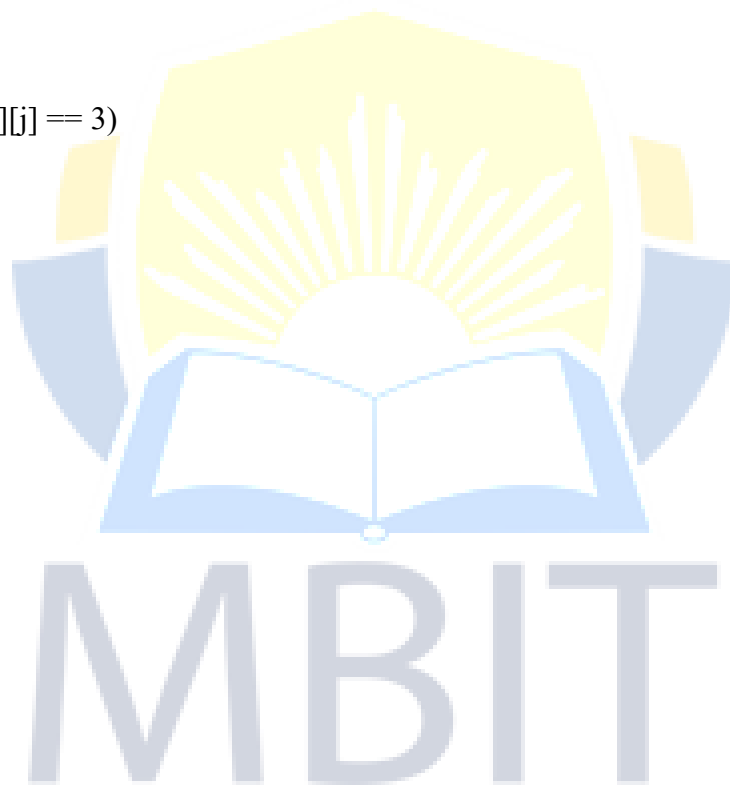
```
printf("\n");
```

```
for (i = 0; i < 5; i++)
```

```
{
```

```
printf("%c ", current_operators[i]);
```

```
for (j = 0; j < 5; j++)  
{  
    if (T4Tutorials[i][j] == 0)  
        printf("< ");  
    else if (T4Tutorials[i][j] == 1)  
        printf("> ");  
    else if (T4Tutorials[i][j] == 2)  
        printf("=" );  
    else if (T4Tutorials[i][j] == 3)  
        printf(" ");  
}  
printf("\n");  
}  
}
```

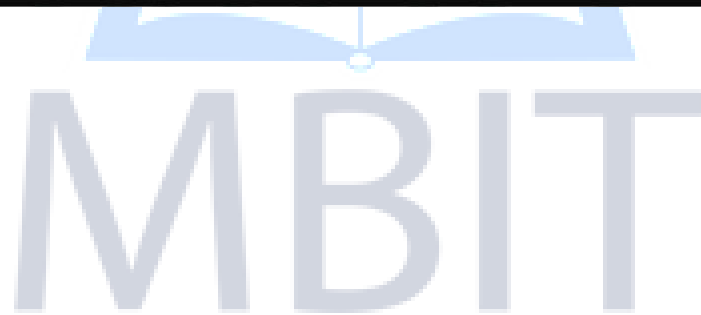


Output:

```
Enter Exp: [a+b*c]
OPERATOR PRECEDENCE Matrix
=====
      + * ( ) ]
+   > < < > >
*   > > < > >
(   < < < =
)   > >   > >
[   < < <  =
Show tree

a
b
c
*
+
Over

-----
Process exited after 62.2 seconds with return value 0
Press any key to continue . . . |
```



Date: - __ / __ / ____

Practical-12

AIM: - Given a parsing table, Parse the given input using Shift Reduce Parser for any unambiguous grammar.

Solution: -

Code:

```
//Including Libraries

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

//Global Variables

int z = 0, i = 0, j = 0, c = 0;

// Modify array size to increase

// length of string to be parsed

char a[16], ac[20], stk[15], act[10];

// This Function will check whether

// the stack contain a production rule

// which is to be Reduce.

// Rules can be E->2E2 , E->3E3 , E->4

void check()

{

// Copying string to be printed as action

strcpy(ac,"REDUCE TO E -> ");
```

```

// c=length of input string

for(z = 0; z < c; z++)

{

//checking for producing rule E->4 i

f(stk[z] == '4')

{

printf("%s4", ac);

stk[z] = 'E';

stk[z + 1] = '\0';

//printing action

printf("\n%s\t%s\t", stk, a);

}

}

for(z = 0; z < c - 2; z++)

{

//checking for another production

if(stk[z] == '2' && stk[z + 1] == 'E' && stk[z + 2] == '2')

{

printf("%s2E2", ac);

stk[z] = 'E';

stk[z + 1] = '\0';

stk[z + 2] = '\0';

printf("\n%s\t%s\t", stk, a);

i = i - 2;

```

```

}

}

for(z=0; z<c-2; z++)

{

//checking for E->3E3

if(stk[z] == '3' && stk[z + 1] == 'E' && stk[z + 2] == '3')

{

printf("%s3E3", ac);

stk[z]='E';

stk[z + 1]='\0';

stk[z + 1]='\0';

printf("\n%s\t%s\t", stk, a);

i = i - 2;

}

}

return ; //return to main

}

//Driver Function

int main()

{

printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");

// a is input string

strcpy(a,"32423");

```

```
// strlen(a) will return the length of a to c
c=strlen(a);

// "SHIFT" is copied to act to be printed
strcpy(act,"SHIFT");

// This will print Labels (column name)
printf("\nstack \t input \t action");

// This will print the initial
// values of stack and input
printf("\n$\t%s$\t", a);

// This will Run upto length of input string
for(i = 0; j < c; i++, j++)
{
    // Printing action
    printf("%s", act);

    // Pushing into stack
    stk[i] = a[j];
    stk[i + 1] = '\0';

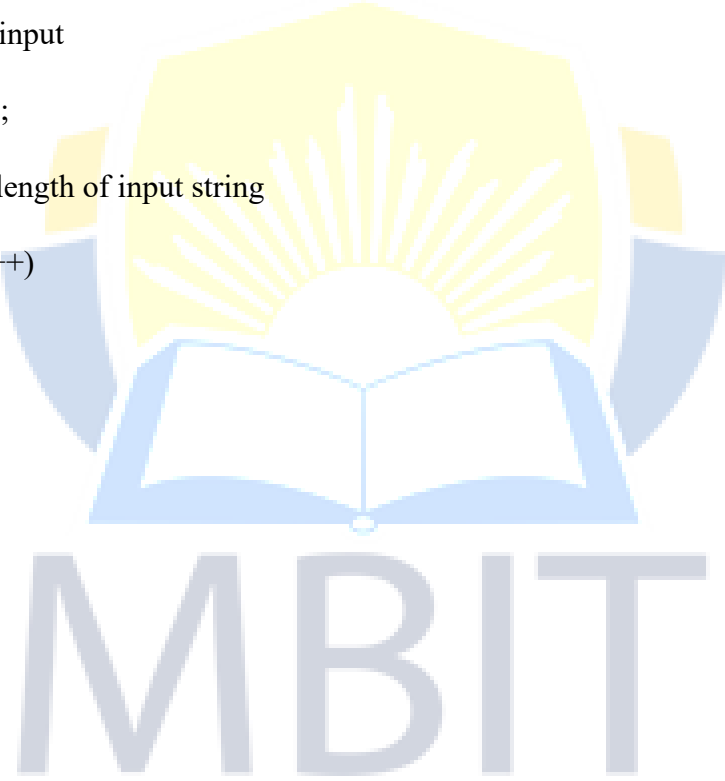
    // Moving the pointer a[j]=' ';

    // Printing action printf("\n%s\t%s$\t", stk, a);

    // Call check function ..which will

    // check the stack whether its contain
    // any production or not

    check();
```



```

}

// Rechecking last time if contain

// any valid production then it will

// replace otherwise invalid check();

// if top of the stack is E(starting symbol)

// then it will accept the input

if(stk[0] == 'E' && stk[1] == '\0')

printf("Accept\n");

else

//else reject

printf("Reject\n");

}

```



Output:

```

GRAMMAR is -
E->2E2
E->3E3
E->4

stack   input   action
$       32423$  SHIFT
$3      2423$   SHIFT
$32     423$    SHIFT
$324    23$     REDUCE TO E -> 4
$32E    23$     SHIFT
$32E2   3$      REDUCE TO E -> 2E2
$3E     3$      SHIFT
$3E3    $       REDUCE TO E -> 3E3
$E      $       Accept

-----
Process exited after 0.3601 seconds with return value 0
Press any key to continue . . . |

```


Date: - __ / __ / ____

Practical-13

AIM: - Introduction to YACC and generate calculator program.

Solution: -

Code:

Lexical Analyzer Source Code:

```
%{  
  
/* Definition section */  
  
#include<stdio.h>  
  
#include "y.tab.h"  
  
extern int yylval;  
  
%}  
  
/* Rule Section */  
  
%%  
  
[0-9]+ {  
    yylval=atoi(yytext);  
    return NUMBER;  
}  
  
[\t] ;  
  
[\n] return 0;  
  
. return yytext[0];  
  
%%
```



```
int yywrap()

{

return 1;

}
```

Parser Source Code:

```
%{

/* Definition section */

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

/* Rule Section */

%%

ArithmeticExpression: E{

printf("\nResult=%d\n", $$);

return 0;

};

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

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

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


```



```
|E/'E' {$$=$1/$3;}
```

```
|E'%E' {$$=$1%$3;}
```

```
|('E') {$$=$2;}
```

```
| NUMBER {$$=$1;}
```

```
;
```

```
%%
```

```
//driver code
```

```
void main()
```

```
{
```

```
printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction,  
Multiplication, Division, Modulus and Round brackets:\n");
```

```
yyvsparse();
```

```
if(flag==0)
```

```
printf("\nEnter arithmetic expression is Valid\n\n");
```

```
}
```

```
void yyerror()
```

```
{
```

```
printf("\nEnter arithmetic expression is Invalid\n\n");
```

```
flag=1;
```

```
}
```

Output:

```
thakur@thakur-VirtualBox:~/Documents/new$ lex calc.l
thakur@thakur-VirtualBox:~/Documents/new$ yacc calc.y
calc.y:39 parser name defined to default : 'parse'
thakur@thakur-VirtualBox:~/Documents/new$ gcc lex.yy.c y.tab.c -w
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
4+5
Result=9
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

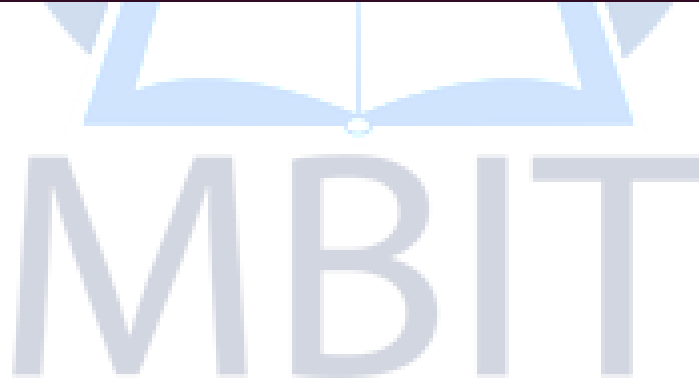
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
10-5
Result=5
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
10+5-
Entered arithmetic expression is Invalid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
10/5
Result=2
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
(2+5)*3
Result=21
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
(2*4)+
Entered arithmetic expression is Invalid
```



Date: - __ / __ / ____

Practical-14

AIM: - Generate 3-tuple intermediate code for given infix expression.

Solution: -

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct ThreeAddressCode
{
    char op;
    char arg1[10];
    char arg2[10];
    char result[10];
};

char* newTempVar()
{
    static int tempVarCount = 0;
    char* tempVar = (char*)malloc(10);
    sprintf(tempVar, "t%d", tempVarCount++);
    return tempVar;
}
```

```
struct ThreeAddressCode* generateCode(char* expression)
{
    struct ThreeAddressCode* code = (struct ThreeAddressCode*)malloc(sizeof(struct
    ThreeAddressCode) * strlen(expression));

    char* tokens = strtok(expression, "+-*/");

    int codeIndex = 0;

    char* tempVar;

    while (tokens != NULL)
    {
        if (codeIndex > 0)
        {
            code[codeIndex].op = expression[codeIndex - 1];
            strcpy(code[codeIndex].arg1, code[codeIndex - 1].result);
            strcpy(code[codeIndex].arg2, tokens);
            tempVar = newTempVar();
            strcpy(code[codeIndex].result, tempVar);
        }
        else
        {
            // First token

            strcpy(code[codeIndex].arg1, tokens);

            strcpy(code[codeIndex].arg2, "");

            code[codeIndex].op = ' ';

            tempVar = newTempVar();
```

```
strcpy(code[codeIndex].result, tempVar);

}

codeIndex++;

tokens = strtok(NULL, "+-*/");

}

return code;

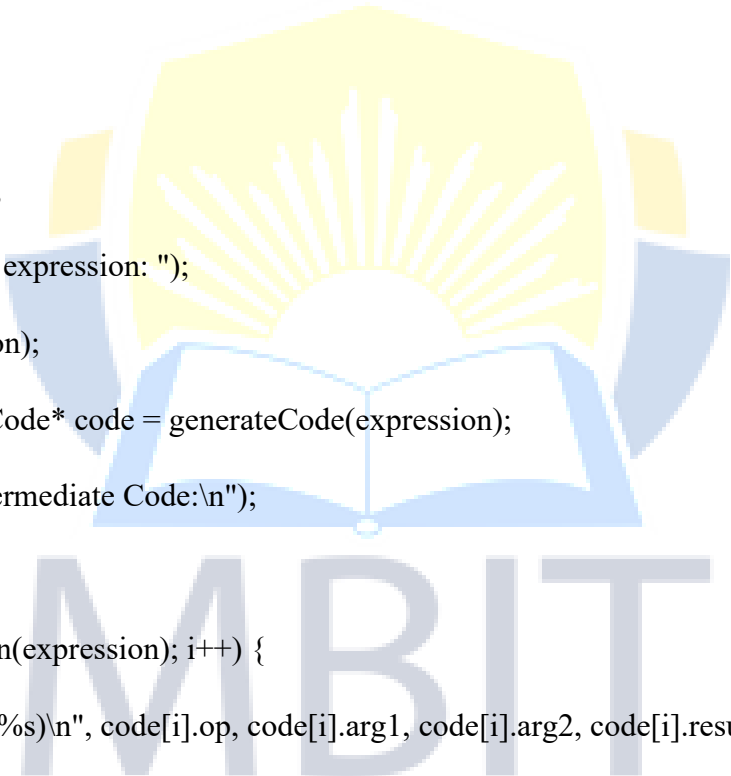
}

int main()
{
char expression[100];
printf("Enter an infix expression: ");
scanf("%s", expression);
struct ThreeAddressCode* code = generateCode(expression);
printf("\n3-Tuple Intermediate Code:\n");
printf("\n");
for (int i = 0; i < strlen(expression); i++) {
printf("(%c, %s, %s, %s)\n", code[i].op, code[i].arg1, code[i].arg2, code[i].result);
}

free(code);

return 0;

}
```



Output:

```
Enter an infix expression: a + b * c - d / e
```

```
3-Tuple Intermediate Code:
```

```
-----
```

```
( , a, , t0)
```

```
(+, t0, b, t1)
```

```
(*, t1, c, t2)
```

```
(-, t2, d, t3)
```

```
(/, t3, e, t4)
```



Date: - __ / __ / ____

Practical-15

AIM: - Extract predecessor and successor from given control flow graph.

Solution: -

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Maximum number of characters in a basic block label
#define MAX_LABEL_LENGTH 20

// Maximum number of basic blocks
#define MAX_BLOCKS 100

int main()
{
    int numBlocks;

    // Prompt for the number of basic blocks
    printf("Enter the number of basic blocks in the CFG: ");

    scanf("%d", &numBlocks);

    // Arrays to store the predecessors and successors

    char predecessors[MAX_BLOCKS][MAX_BLOCKS][MAX_LABEL_LENGTH];

    char successors[MAX_BLOCKS][MAX_BLOCKS][MAX_LABEL_LENGTH];

    // Input successors for each basic block
```

```
for (int i = 0; i < numBlocks; i++)
{
    int numSuccessors;

    printf("Enter the number of successors for Basic Block %d: ", i);

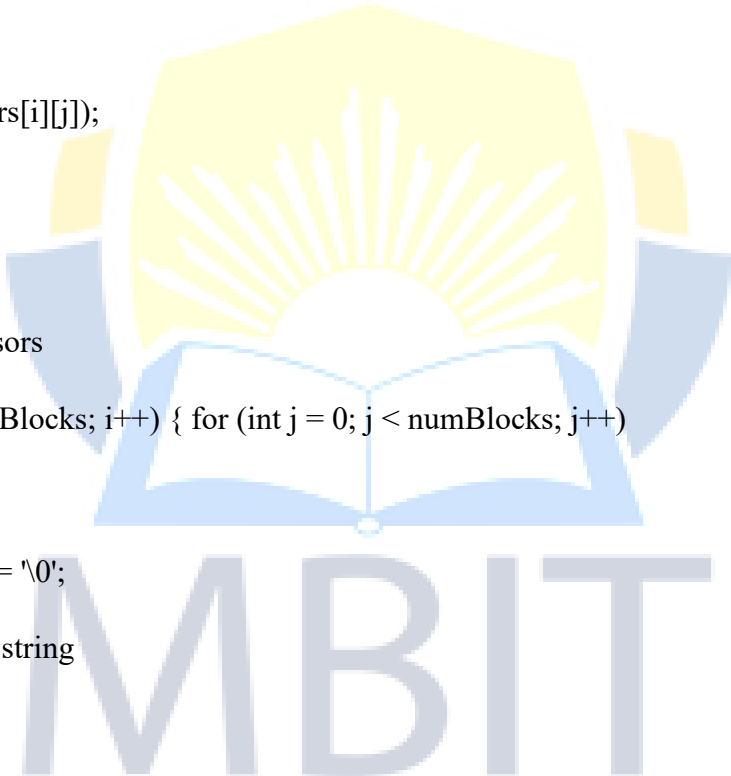
    scanf("%d", &numSuccessors);

    printf("Enter successors for Basic Block %d (space-separated labels): ", i);

    for (int j = 0; j < numSuccessors; j++)
    {
        scanf("%s", successors[i][j]);
    }
}

// Calculate predecessors
for (int i = 0; i < numBlocks; i++) { for (int j = 0; j < numBlocks; j++)
{
    predecessors[i][j][0] = '\0';
    // Initialize to empty string
}
}

for (int i = 0; i < numBlocks; i++)
{
    for (int j = 0; j < numBlocks; j++)
    {
        for (int k = 0; k < numBlocks; k++)
        {
```



```

for (int l = 0; l < numBlocks; l++)
{
    if (strcmp(successors[i][j], successors[k][l]) == 0)
    {
        printf(predecessors[i][j], "%sBB%d ", predecessors[i][j], k);
    }
}

}

}

}

// Print the results
printf("\nPredecessors and Successors:\n");
for (int i = 0; i < numBlocks; i++) {
    printf("Basic Block BB%d:\n", i);
    printf("Predecessors: %s\n", predecessors[i][0]);
    printf("Successors: ");
    for (int j = 0; j < numBlocks; j++)
    {
        printf("%s", successors[i][j]);
        if (j < numBlocks - 1)
        {
            printf(" ");
        }
    }
}

```

```
printf("\n\n");  
  
}  
  
return 0;  
  
}
```

Output:

```
Enter the number of basic blocks in the CFG: 3  
Enter the number of successors for Basic Block 0: 2  
Enter successors for Basic Block 0 (space-separated labels): BB1 BB2  
Enter the number of successors for Basic Block 1: 2  
Enter successors for Basic Block 1 (space-separated labels): BB1  
BB2  
Enter the number of successors for Basic Block 2: 1  
Enter successors for Basic Block 2 (space-separated labels): BB2  
Predecessors and Successors:  
Basic Block BB0:  
Predecessors: BB0 BB1  
Successors: BB1 BB2  
  
Basic Block BB1:  
Predecessors: BB0 BB1  
Successors: BB1 BB2  
  
Basic Block BB2:  
Predecessors: BB0 BB2  
Successors: BB2
```