LAB MANUAL

OF

Compiler Design

Laboratory

(CSE606)

Bachelor of Technology (CSE)

By

Mahi V Prajapati – 22000996

Third Year, Semester 6

Course In-Charge: Prof Vaibhavi Patel

Department of Computer Science & Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester 2025

TABLE OF CONTENT

Sr. No		Experiment Title
1		**a)** Write a program to recognize strings starts with 'a' over {a, b}. **b)** Write a program to recognize strings end with 'a'. **c)** Write a program to recognize strings end with 'ab'. Take the input from text file. **d)** Write a program to recognize strings contains 'ab'. Take the input from text file.
2		a) Write a program to recognize the valid identifiers. b) Write a program to recognize the valid operators. c) Write a program to recognize the valid number. d) Write a program to recognize the valid comments. e) Program to implement Lexical Analyzer.
3		To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)
4		Implement following programs using Lex. a. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words. b. Write a Lex program to take input from text file and count number of vowels and consonants. c. Write a Lex program to print out all numbers from the given file. d. Write a Lex program which adds line numbers to the given file and display the same into different file. e. Write a Lex program to printout all markup tags and HTML comments in file.
5		a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file. b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.
6		Program to implement Recursive Descent Parsing in C.
7		a. To Study about Yet Another Compiler-Compiler(YACC). b. Create Yacc and Lex specification files to recognizes arithmetic

		expressions involving +, -, * and / . c. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments. d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

100

   60-CA: 30: Mid exam

       30: project

40: 20: end sem exam

  20: project

Experiment – 1

a) Write a program to recognize strings starts with 'a' over {a, b}.

Code:

```c
#include <stdio.h>

int main()
{
    char input[10];
    int i=0, state=0;

    printf("Enter a string:" );
    scanf("%s",&input);

    while(input[i]!='\0')
    {
        switch(state)
        {
            case 0:
                if (input[i]=='a')
                {
                    state=1;
                }
                else
                state=0;
                break;

            case 1:
                if (input[i]=='a')
                {
                    state=1;
                }
                else
                state=0;
                break;
        }
        i++;
    }
```

```
if(state==1){ printf("String is Valid");}
else {printf("String is Invalid");}
return 0;
}
```

O/P:

```
Enter a string:abb
String is Invalid
```

b) Write a program to recognize strings end with 'a'.

Code:
```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[100];

    // Prompt the user to enter a string
    printf("Enter a string: ");
    scanf("%99s", str); // Limiting input to avoid buffer overflow

    // Get the length of the string
    int length = strlen(str);

    // Check if the last character is 'a'
    if (length > 0 && str[length - 1] == 'a') {
        printf("The string \"%s\" ends with 'a'.\n", str);
    } else {
        printf("The string \"%s\" does not end with 'a'.\n", str);
    }

    return 0;
}
```

O/P:

```
Enter a string: abababa
The string "abababa" ends with 'a'.
```

c) Write a program to recognize strings end with 'ab'. Take the input from text file.

Code:

```c
#include <stdio.h>

#include <string.h>


#define MAX_LINE_LENGTH 100


void checkStringsEndingWithAb(const char *filePath) {

  FILE *file = fopen(filePath, "r");

  if (file == NULL) {

    printf("Error: Could not open file '%s'\n", filePath);

    return;

  }


  char line[MAX_LINE_LENGTH];

  printf("Strings that end with 'ab':\n");

  while (fgets(line, sizeof(line), file) != NULL) {

    // Remove trailing newline character if present

    size_t len = strlen(line);

    if (len > 0 && line[len - 1] == '\n') {

      line[len - 1] = '\0';

    }


    // Check if the string ends with "ab"
```

```c
        len = strlen(line);

        if (len >= 2 && line[len - 2] == 'a' && line[len - 1] == 'b') {

            printf("%s\n", line);

        }

    }


    fclose(file);

}


int main() {

    char filePath[100];

    printf("Enter the path to the text file: ");

    scanf("%s", filePath);


    checkStringsEndingWithAb(filePath);


    return 0;

}
```
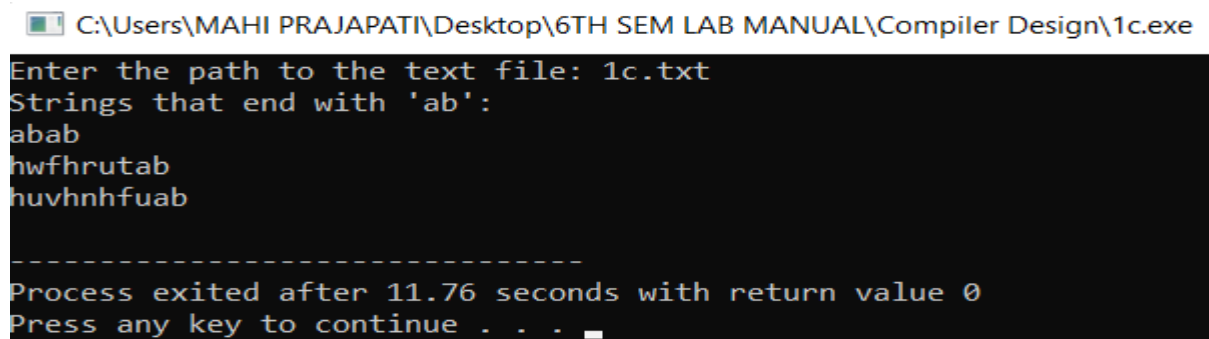
O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\1c.exe

```
Enter the path to the text file: 1c.txt
Strings that end with 'ab':
abab
hwfhrutab
huvhnhfuab

--------------------------------
Process exited after 11.76 seconds with return value 0
Press any key to continue . . . _
```

d) Write a program to recognize strings contains 'ab'. Take the input from text file.

Code:

```c
#include <stdio.h>

#include <string.h>


#define MAX_LINE_LENGTH 100 // Maximum length for each line


// Function to check strings containing "ab"
void checkStringsContainingAb(const char *filePath) {
    FILE *file = fopen(filePath, "r"); // Open the file in read mode
    if (file == NULL) {
        printf("Error: Could not open file '%s'\n", filePath);
        return; // Exit the function if the file cannot be opened
    }


    char line[MAX_LINE_LENGTH]; // Buffer to hold each line
    printf("Strings that contain 'ab':\n");


    // Read each line from the file
    while (fgets(line, sizeof(line), file) != NULL) {
        // Remove the trailing newline character if present
        size_t len = strlen(line);
        if (len > 0 && line[len - 1] == '\n') {
            line[len - 1] = '\0';
```

```c
        }

        // Check if the string contains "ab"
        if (strstr(line, "ab") != NULL) {
            printf("%s\n", line);
        }
    }

    fclose(file); // Close the file
}

int main() {
    char filePath[100];

    // Prompt the user for the input file path
    printf("Enter the path to the text file: ");
    scanf("%s", filePath);

    // Call the function to check strings
    checkStringsContainingAb(filePath);

    return 0;
}
```

O/P:

```
C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\1d.exe

Enter the path to the text file: 1d.txt
Strings that contain 'ab':
ababab
abcc
hjklabjk


--------------------------------
Process exited after 5.695 seconds with return value 0
Press any key to continue . . .
```

Experiment – 2

a) Write a program to recognize the valid identifiers and keywords.

Code:

```c
#include <stdio.h>

#include <ctype.h>

#include <string.h>


int main() {

    char input[20];

    int state = 0, i = 0;


    printf("Enter a string: ");

    scanf("%s", input);


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

        switch (state) {

            case 0:

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

                    state = 1;

                } else if (input[i] == '_' || isalpha(input[i])) {

                    state = 5;

                } else {

                    printf("Not a keyword or identifier\n");

                    return 0;

                }
```

```
      break;


case 1:
   if (input[i] == 'n') {

      state = 2;

   } else if (input[i] == '_' || isalpha(input[i]) || isdigit(input[i])) {

      state = 5;

   } else {

      printf("Identifier: %s\n", input);

      return 0;

   }

   break;


case 2:
   if (input[i] == 't') {

      state = 3;

   } else if (input[i] == '_' || isalpha(input[i]) || isdigit(input[i])) {

      state = 5;

   } else {

      state=0;

      printf("Identifier: %s\n", input);

      return 0;

   }

   break;
```

```
      case 3:

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

             state = 5;

          } else {

             state=0;

             printf("Keyword: int\n");

             return 0;

          }

          break;


      case 5:

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

             state = 5;

          } else {

             printf("Identifier: %s\n", input);

             return 0;

          }

          break;

   }

   i++;

}


// Final state check

if (state == 3) {

   printf("Keyword: int\n");
```
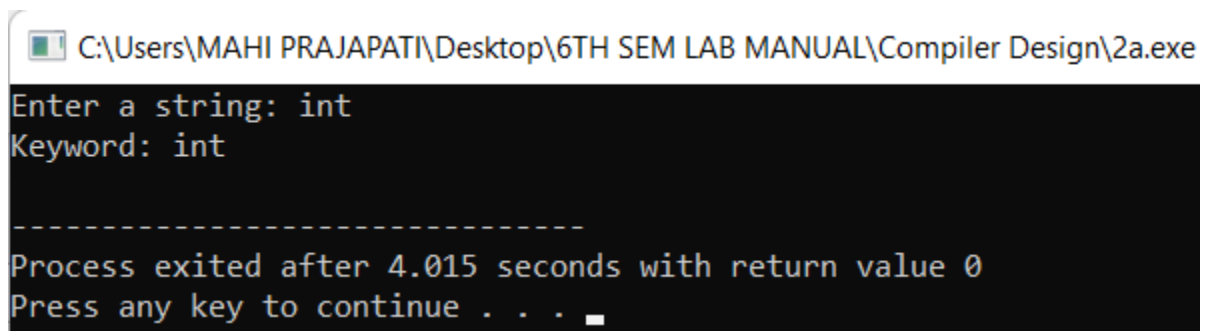
```
    } else if (state == 5) {

        printf("Identifier: %s\n", input);

    } else {

        printf("Invalid input\n");

    }


    return 0;

}
```

O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\2a.exe

```
Enter a string: int
Keyword: int

--------------------------------
Process exited after 4.015 seconds with return value 0
Press any key to continue . . .
```

b) Write a program to recognize the valid operators.

Code:

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

void recognize_operator(char *lexeme);

int main() {
    // Sample input string with various operators
    char input[] = "+ - * / % = == != < > <= >= && || ^ ~ !";

    // Split the input string by spaces
    char *token = strtok(input, " ");
```

```c
    // Loop through each token (operator) and recognize it
    while (token != NULL) {
      recognize_operator(token);
      token = strtok(NULL, " ");
    }

    return 0;
}

void recognize_operator(char *lexeme) {
    // List of valid operators
    char *operators[] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=",
">=", "&&", "||", "^", "~", "!"};

    int is_operator = 0;

    // Check if lexeme is a valid operator
    for (int i = 0; i < 18; i++) {
      if (strcmp(lexeme, operators[i]) == 0) {
         is_operator = 1;
         break;
      }
    }

    if (is_operator) {
      printf("%s is a valid operator\n", lexeme);
    } else {
      printf("%s is not a valid operator\n", lexeme);
    }
}
```
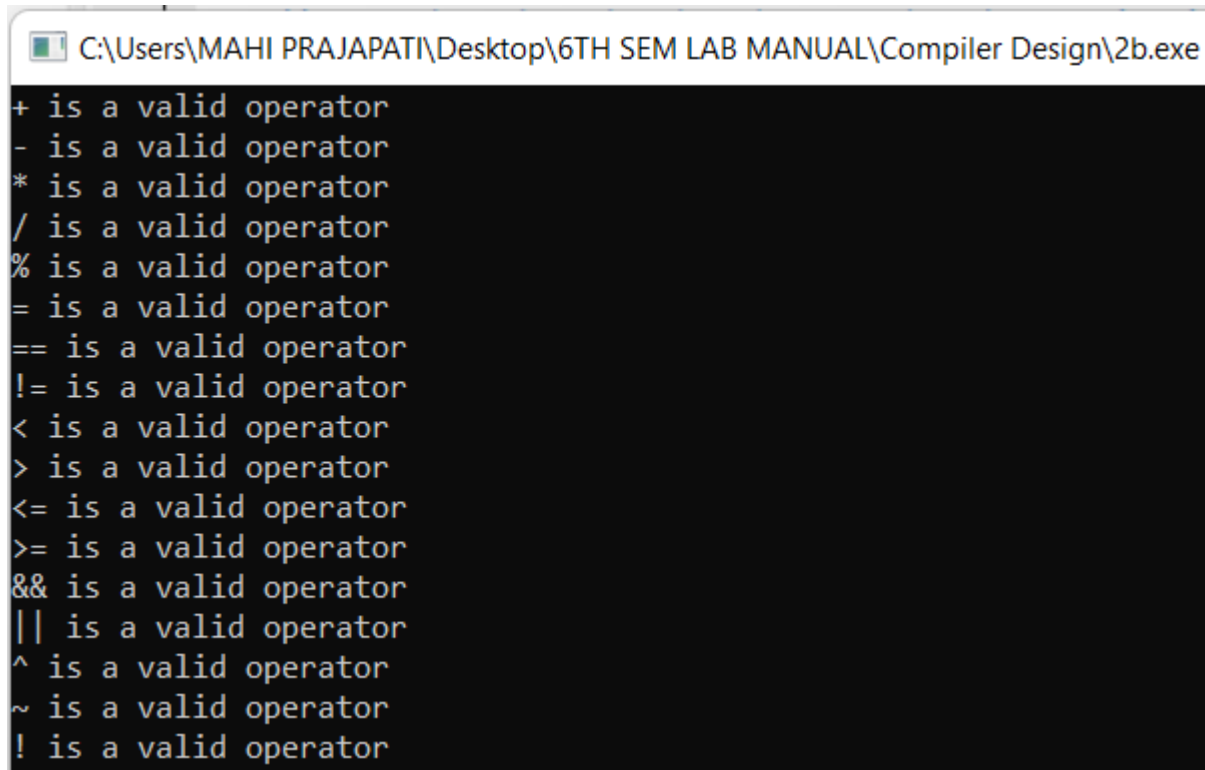
O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\2b.exe

```
+ is a valid operator
- is a valid operator
* is a valid operator
/ is a valid operator
% is a valid operator
= is a valid operator
== is a valid operator
!= is a valid operator
< is a valid operator
> is a valid operator
<= is a valid operator
>= is a valid operator
&& is a valid operator
|| is a valid operator
^ is a valid operator
~ is a valid operator
! is a valid operator
```

c) **Write a program to recognize the valid number.**
   Code:
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

void recognize_number(char *lexeme);

int main() {
    // Sample input string containing potential numbers
    char input[] = "123 45.67 3.14e5 0.003 -123 56e-3 1000.0";

    // Split the input string by spaces to analyze each token
    char *token = strtok(input, " ");

    // Loop through each token (number) and recognize it
    while (token != NULL) {
        recognize_number(token);
```

16

```c
      token = strtok(NULL, " ");
   }

   return 0;
}

void recognize_number(char *lexeme) {
   // Regular expressions to check for valid numbers
   int is_valid = 1;

   // Check if the number is a valid integer or floating-point number
   // Integer: can have optional + or - sign, digits
   // Floating-point: can have an optional sign, digits, and a decimal point with
digits after it
   // Scientific notation: can be 'e' or 'E' followed by an optional sign and digits

   // Check if it's a valid integer or decimal
   char *endptr;
   strtod(lexeme, &endptr); // Converts string to double
   if (*endptr != '\0') {     // If the endptr points to non-null, it's an invalid
number
      is_valid = 0;
   }

   // If valid number
   if (is_valid) {
      printf("%s is a valid number\n", lexeme);
   } else {
      printf("%s is not a valid number\n", lexeme);
   }
}
```
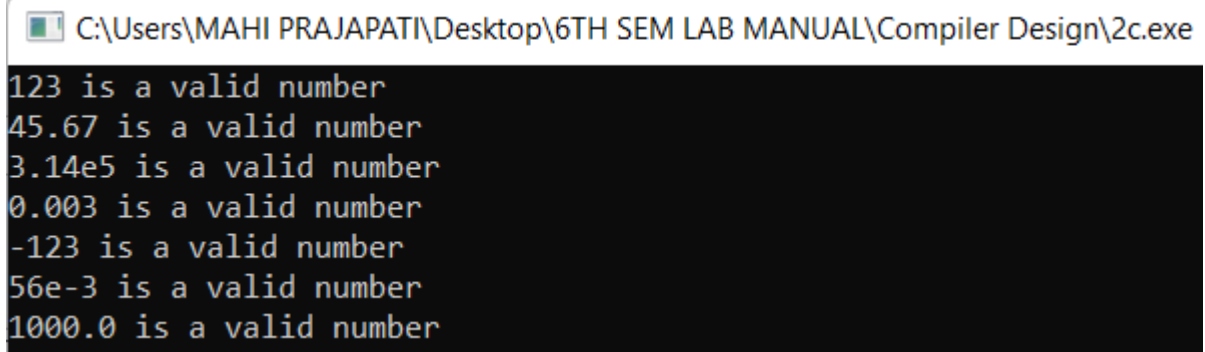
O/P:

```
123 is a valid number
45.67 is a valid number
3.14e5 is a valid number
0.003 is a valid number
-123 is a valid number
56e-3 is a valid number
1000.0 is a valid number
```

d) Write a program to recognize the valid comments.

Code:
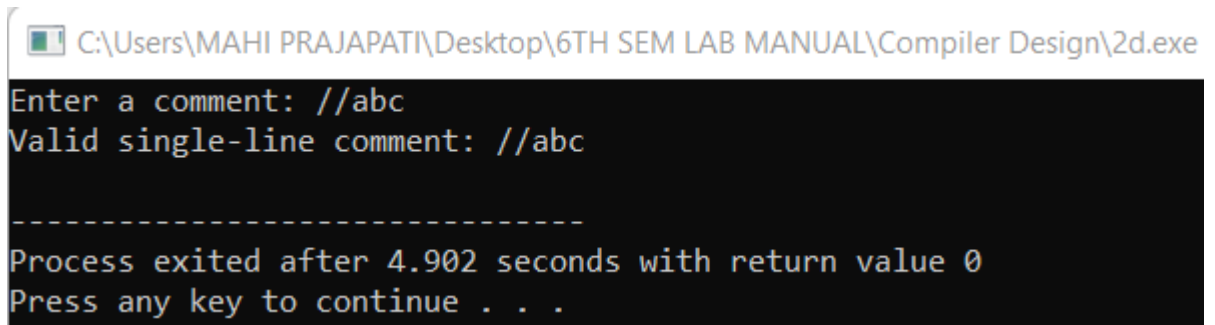
```c
#include <stdio.h>

#include <string.h>


void checkComment(char *line) {
   if (line[0] == '/' && line[1] == '/') {
      printf("Valid single-line comment: %s\n", line);
   } else if (line[0] == '/' && line[1] == '*') {
      int len = strlen(line);
      if (line[len - 2] == '*' && line[len - 1] == '/') {
         printf("Valid multi-line comment: %s\n", line);
      } else {
         printf("Invalid comment or incomplete multi-line comment.\n");
      }
   } else {
      printf("Not a comment.\n");
   }
```

18

```
}


int main() {
    char input[256];
    printf("Enter a comment: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0; // Remove newline character


    checkComment(input);
    return 0;
}
```

O/P:

```
C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\2d.exe

Enter a comment: //abc
Valid single-line comment: //abc

--------------------------------
Process exited after 4.902 seconds with return value 0
Press any key to continue . . .
```

Transition diagram for unsigned numbers
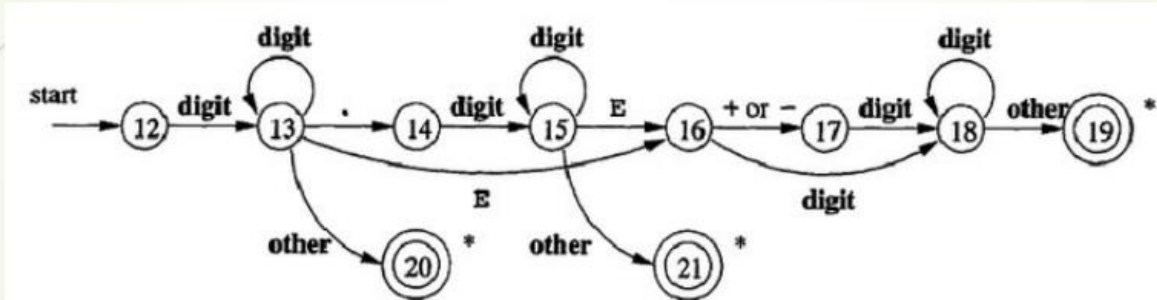
Code:

#include <stdio.h>

#include <ctype.h>

// Enum to represent states in the transition diagram

typedef enum {

   START=12, DIGIT1=13, DOT=14, DIGIT2=15, E_STATE=16, SIGN=17, DIGIT3=18, OTHER=19, ERROR=20, ERROR2=21

} State;

int main() {

   char c;

   State state = START;

   printf("Enter a number: ");

   while ((c = getchar()) != '\n') {

      switch (state) {

```
case START:

    if (isdigit(c)) state = DIGIT1;

    else state = ERROR;

    break;


case DIGIT1:

    if (isdigit(c)) state = DIGIT1;

    else if (c == '.') state = DOT;

    else if (c == 'E' || c == 'e') state = E_STATE;

    else state = ERROR;

    break;


case DOT:

    if (isdigit(c)) state = DIGIT2;

    else state = ERROR;

    break;


case DIGIT2:

    if (isdigit(c)) state = DIGIT2;

    else if (c == 'E' || c == 'e') state = E_STATE;

    else state = ERROR;

    break;


case E_STATE:

    if (c == '+' || c == '-') state = SIGN;
```

```c
            else if (isdigit(c)) state = DIGIT3;

            else state = ERROR;

            break;


        case SIGN:
            if (isdigit(c)) state = DIGIT3;

            else state = ERROR;

            break;


        case DIGIT3:
            if (isdigit(c)) state = DIGIT3;

            else state = ERROR;

            break;


        default:
            state = ERROR;

            break;
    }


    if (state == ERROR) break;
}


if (state == DIGIT1 || state == DIGIT2 || state == DIGIT3) {

    printf("Valid unsigned number.\n");

} else {
```
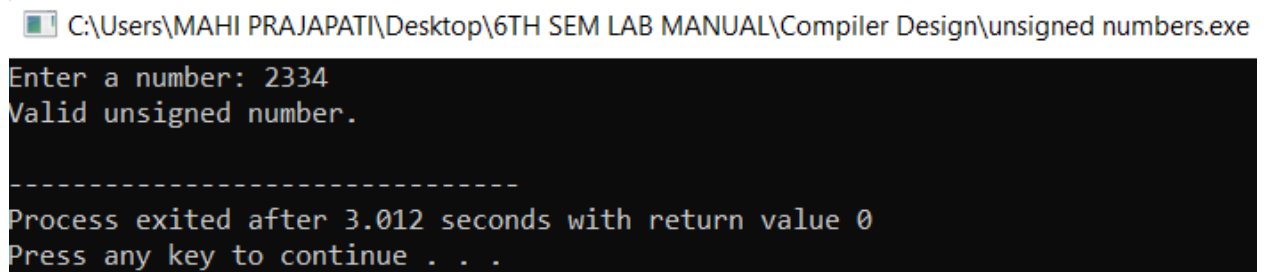
22

```
        printf("Invalid number.\n");

    }


    return 0;

}
```

O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\unsigned numbers.exe

```
Enter a number: 2334
Valid unsigned number.

-------------------------------
Process exited after 3.012 seconds with return value 0
Press any key to continue . . .
```

e) Program to implement Lexical Analyzer.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


#define BUFFER_SIZE 1000


void check(char *lexeme);


int main() {

    FILE *f1;
```

```c
    char buffer[BUFFER_SIZE], lexeme[50]; // Static buffer for input and
lexeme storage
    char c;
    int f = 0, state = 0, i = 0;


    f1 = fopen("input.txt", "r");
    if (f1 == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fread(buffer, sizeof(char), BUFFER_SIZE - 1, f1);
    buffer[BUFFER_SIZE - 1] = '\0'; // Null termination
    fclose(f1);


    while (buffer[f] != '\0') {
        c = buffer[f];
        switch (state) {
            case 0:
                if (isalpha(c) || c == '_') {
                    state = 1;
                    lexeme[i++] = c;
                } else if (isdigit(c)) {
                    state = 13;
                    lexeme[i++] = c;
                } else if (isspace(c)) {
```

```c
            state = 0;
        } else if (c == '/') {
            state = 11;
        } else if (c == '\') {
            state = 19;
            lexeme[i++] = c;
        } else if (c == '"') {
            state = 20;
            lexeme[i++] = c;
        } else if (strchr(";,{}()", c)) {
            printf("%c is a symbol\n", c);
        } else if (strchr("+-*=<>!", c)) {
            if (buffer[f + 1] == '=') {
                printf("%c%c is an operator\n", c, buffer[f + 1]);
                f++;
            } else {
                printf("%c is an operator\n", c);
            }
        }
        break;

    case 1:
        if (isalnum(c) || c == '_') {
            lexeme[i++] = c;
        } else {
```

```
            lexeme[i] = '\0'; // Null-terminate the lexeme

            check(lexeme);

            state = 0;

            i = 0;

            f--; // Reprocess current character

        }

        break;


    case 11:

                    if (c == '/') { // Single-line comment

                        printf("//"); // Print start of comment

                            f++;

                    while (buffer[f] != '\n' && buffer[f] != '\0') {

                    printf("%c", buffer[f]); // Print the comment

                            f++;

                            }

                            printf(" is a single-line comment\n");

                    } else if (c == '*') { // Multi-line comment

                    printf("/*"); // Print start of comment

                                f++;

                    while (buffer[f] != '\0' && !(buffer[f] == '*' && buffer[f
+ 1] == '/')) {

                    printf("%c", buffer[f]); // Print the comment

                            f++;

                            }
```

```
                    printf("*/ is a multi-line comment\n");

                            f += 2;

    }
        state = 0;

        break;




case 13:

    if (isdigit(c)) {

        lexeme[i++] = c;

    } else if (c == '.') {

        state = 14;

        lexeme[i++] = c;

    } else if (c == 'E' || c == 'e') {

        state = 16;

        lexeme[i++] = c;

    } else {

        lexeme[i] = '\0';

        printf("%s is a valid number\n", lexeme);

        state = 0;

        i = 0;

        f--;

    }

    break;
```

```c
case 14:
   if (isdigit(c)) {

     lexeme[i++] = c;

   } else if (c == 'E' || c == 'e') {

     state = 16;

     lexeme[i++] = c;

   } else {

     lexeme[i] = '\0';

     printf("%s is a valid floating-point number\n", lexeme);

     state = 0;

     i = 0;

     f--;

   }

   break;


case 16:
   if (c == '+' || c == '-') {

     state = 17;

     lexeme[i++] = c;

   } else if (isdigit(c)) {

     state = 18;

     lexeme[i++] = c;

   } else {

     printf("Invalid scientific notation\n");

     state = 0;
```

```
            i = 0;
        }
        break;

case 17:
    if (isdigit(c)) {
        state = 18;
        lexeme[i++] = c;
    } else {
        printf("Invalid scientific notation\n");
        state = 0;
        i = 0;
    }
    break;

case 18:
    if (isdigit(c)) {
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        printf("%s is a valid scientific notation number\n", lexeme);
        state = 0;
        i = 0;
        f--;
    }
```

```c
      break;

case 19:
  if (isalnum(c)) {
    lexeme[i++] = c;
  } else if (c == '\'') {
    lexeme[i++] = c;
    lexeme[i] = '\0';
    printf("%s is a valid character literal\n", lexeme);
    state = 0;
    i = 0;
  } else {
    state = 0;
    i = 0;
  }
  break;

case 20:
  if (c != '"') {
    lexeme[i++] = c;
  } else {
    lexeme[i++] = c;
    lexeme[i] = '\0';
    printf("%s is a valid string literal\n", lexeme);
    state = 0;
```
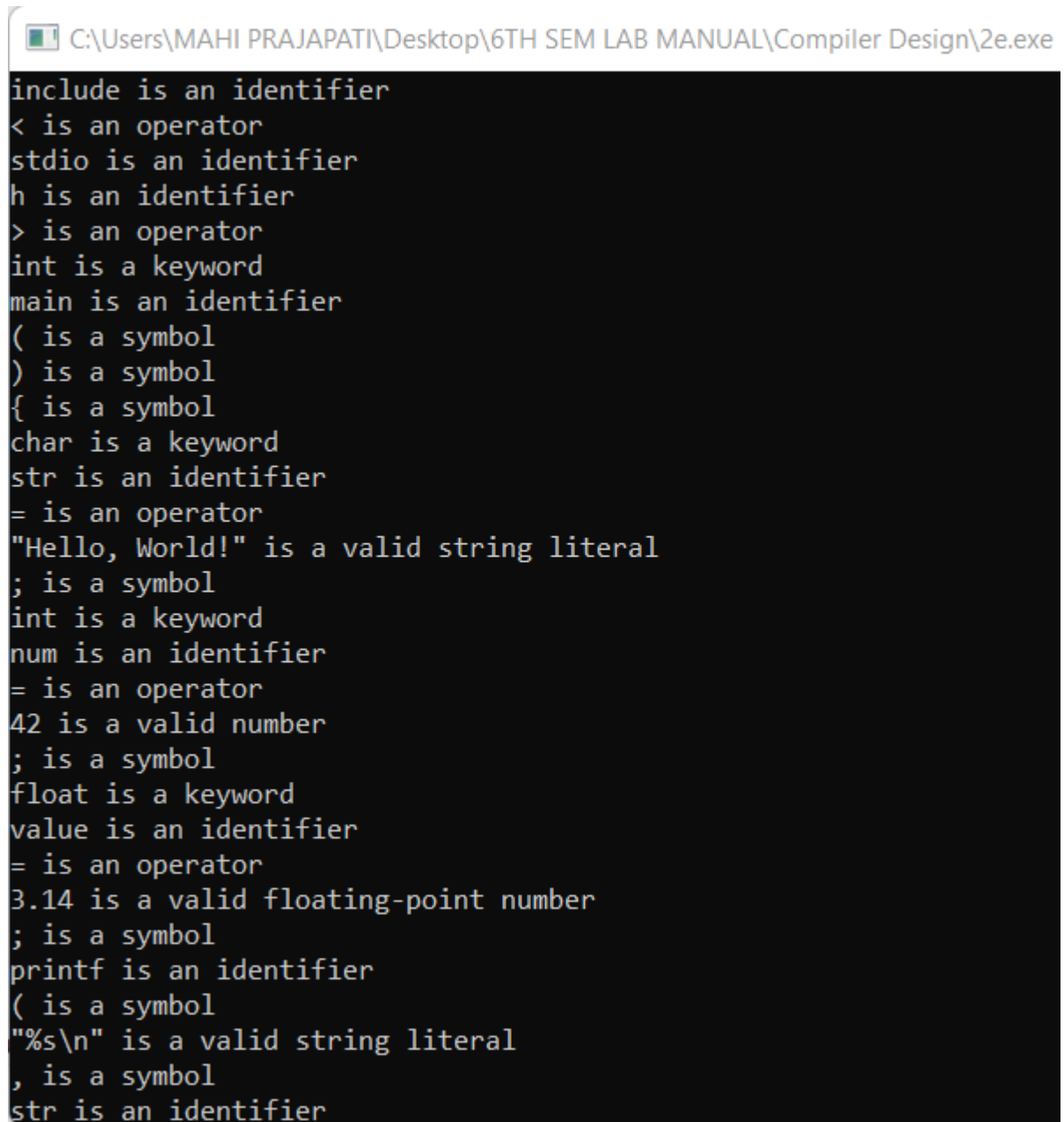
30

```c
                i = 0;

            }

         break;

     }

     f++;

  }

  return 0;

}


void check(char *lexeme) {

   char *keywords[] = {

      "auto", "break", "case", "char", "const", "continue", "default", "do",

      "double", "else", "enum", "extern", "float", "for", "goto", "if",

      "inline", "int", "long", "register", "restrict", "return", "short", "signed",

      "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned",
"void", "volatile", "while"

   };

   for (int i = 0; i < 32; i++) {

      if (strcmp(lexeme, keywords[i]) == 0) {

         printf("%s is a keyword\n", lexeme);

         return;

      }

   }

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

}
```

O/P:

```
C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\2e.exe

include is an identifier
< is an operator
stdio is an identifier
h is an identifier
> is an operator
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
char is a keyword
str is an identifier
= is an operator
"Hello, World!" is a valid string literal
; is a symbol
int is a keyword
num is an identifier
= is an operator
42 is a valid number
; is a symbol
float is a keyword
value is an identifier
= is an operator
3.14 is a valid floating-point number
; is a symbol
printf is an identifier
( is a symbol
"%s\n" is a valid string literal
, is a symbol
str is an identifier
```

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\2e.exe

```
) is a symbol
; is a symbol
// Print string is a single-line comment
printf is an identifier
( is a symbol
"Number: %d, Value: %.2f\n" is a valid string literal
, is a symbol
num is an identifier
, is a symbol
value is an identifier
) is a symbol
; is a symbol
return is a keyword
0 is a valid number
; is a symbol
} is a symbol
//hi is a single-line comment
/*hgbjrifudhdueuhwuweriuru*/ is a multi-line comment

---------------------------------
```

Experiment – 3

Aim: To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)

Introduction:

A Lexical Analyzer converts an input stream (source code) into a sequence of tokens, which are then used by the parser in a compiler. Lex and Flex are tools designed for this purpose.

1. Lexical Analyzer Generator (LEX)

LEX is a tool used to generate lexical analyzers. It takes a set of regular expressions (token patterns) as input and produces a C program that can identify these tokens.

Working of LEX:

1. Specification File:

A LEX program consists of three sections:

- Definition Section: Declare header files and global variables.

- Rules Section: Define token patterns using regular expressions.

- C Code Section: Additional helper functions (optional).

2. Compilation Process:

- The LEX file (.l) is compiled using lex to generate lex.yy.c.

- The lex.yy.c file is compiled with a C compiler (gcc lex.yy.c -o output).

- The executable processes input and tokenizes it.

Example LEX Program:

```
%{
#include <stdio.h>
%}
```

```
%%

[0-9]+    { printf("Number: %s\n", yytext); }

[a-zA-Z]+ { printf("Identifier: %s\n", yytext); }

.         { printf("Special Symbol: %s\n", yytext); }

%%

int main() {

   yylex();

   return 0;

}

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

Commands to Run:

lex filename.l

gcc lex.yy.c -o output

./output < input.txt

2. Fast Lexical Analyzer (FLEX)

Flex is an improved and faster version of Lex. It provides better performance and extended functionality.

Key Features of FLEX:

• Works similarly to Lex, but faster.

• Generates a more optimized lex.yy.c.

• Supports additional options like debugging and performance tuning.

Example FLEX Program:

(Same structure as LEX)

```
%{
#include <stdio.h>
%}


%%
[0-9]+    { printf("Number: %s\n", yytext); }
[a-zA-Z]+ { printf("Identifier: %s\n", yytext); }
.         { printf("Special Symbol: %s\n", yytext); }
%%


int main() {
   yylex();
   return 0;
}
int yywrap() { return 1; }
```

Commands to Run:

flex filename.l

gcc lex.yy.c -o output

./output < input.txt

Comparison: LEX vs FLEX

Feature	LEX	FLEX

Speed	Slower	Faster
Compatibility	Traditional UNIX tool	GNU version, supports more platforms
Debugging	Limited	More debugging options
Performance	Basic optimization	Highly optimized DFA

Conclusion:

• Lex and Flex automate the creation of lexical analyzers.

• Flex is an enhanced version of Lex and is more commonly used today.

• These tools simplify token generation in compiler design.

Experiment – 4

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

Code:
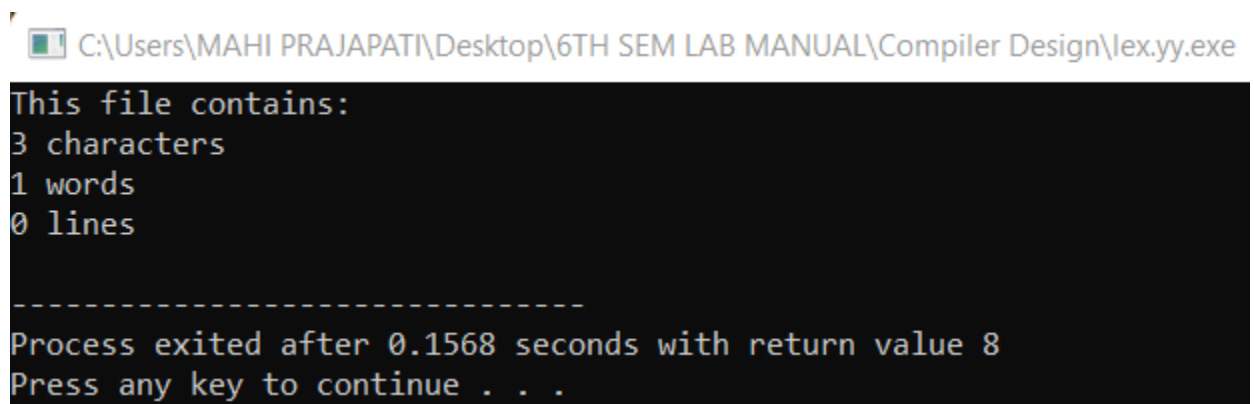
```
%{
#include<stdio.h>


int l = 0, w = 0, c = 0;
int in_word = 0; // Flag to track words
%}
%%
\n      { l++; c++; in_word = 0; }  // Newline increases line and character count, resets word flag
[ \t]+  { c += yyleng; in_word = 0; }  // Spaces & tabs count as characters, reset word flag
.       { c++; if (!in_word) { w++; in_word = 1; } }  // Count characters, track words
%%


void main() {
    yyin = fopen("input.txt", "r");
    if (!yyin) {
        printf("Error opening file\n");
        return;
    }
    yylex();
```
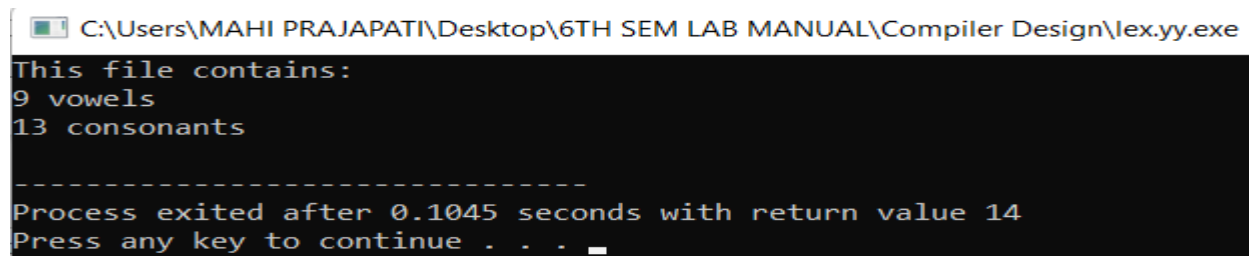
```
    fclose(yyin);


    printf("This file contains:\n");

    printf("%d characters\n", c);

    printf("%d words\n", w);

    printf("%d lines\n", l);

}


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

O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\lex.yy.exe

```
This file contains:
3 characters
1 words
0 lines

--------------------------------
Process exited after 0.1568 seconds with return value 8
Press any key to continue . . .
```

b) Write a Lex program to take input from text file and count number of vowels and consonants.
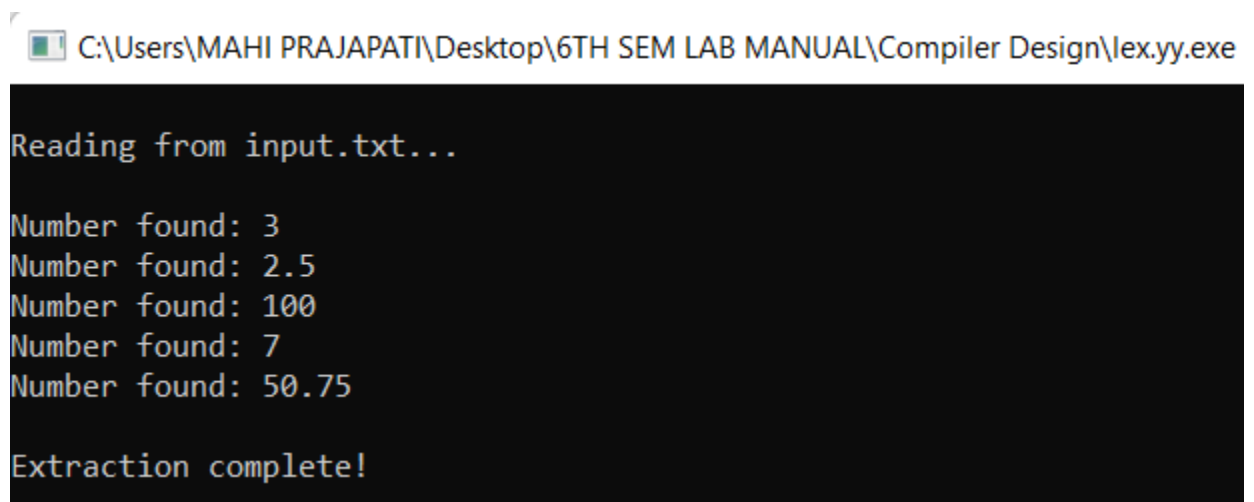
Code:

```
%{

#include<stdio.h>


int vowels = 0, consonants = 0;

%}

%%
```

```
[aAeEiIoOuU]   { vowels++; }      // Count vowels

[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z]  { consonants++; }  // Count
consonants

.          { }  // Ignore other characters

\n         { }  // Ignore newlines

%%

void main() {

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

   if (!yyin) {

      printf("Error opening file\n");

      return;

   }

   yylex();

   fclose(yyin);


   printf("This file contains:\n");

   printf("%d vowels\n", vowels);

   printf("%d consonants\n", consonants);

}

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

O/P:

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

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>


FILE *fp;
%}


%%


[0-9]+(\.[0-9]+)?  { printf("Number found: %s\n", yytext); }


.|\n { } // Ignore everything else


%%


int main() {
    fp = fopen("input.txt", "r");  // Open input file
    if (fp == NULL) {
        printf("\nError: Could not open input.txt!\n");
        return 1;
    }


    yyin = fp;  // Set input source to file
```

printf("\nReading from input.txt...\n\n");

yylex();   // Process input file

fclose(fp); // Close file after processing

printf("\nExtraction complete!\n");

return 0;

}

int yywrap() { return 1; }

O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\lex.yy.exe

```
Reading from input.txt...

Number found: 3
Number found: 2.5
Number found: 100
Number found: 7
Number found: 50.75

Extraction complete!
```

d) **Write a Lex program which adds line numbers to the given file and display the same into different file.**

Code:

%{

#include <stdio.h>

int line_number = 1;

```
FILE *output;

%}

%%

^.*   { fprintf(output, "%d: %s\n", line_number++, yytext); }  // Add line
numbers

%%

void main() {

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

  if (!yyin) {

    printf("Error opening input file\n");

    return;

  }


  output = fopen("output.txt", "w");  // Open output file

  if (!output) {

    printf("Error opening output file\n");

    return;

  }


  yylex();  // Process file


  fclose(yyin);

  fclose(output);


  printf("Processed file saved as 'output.txt'\n");
```
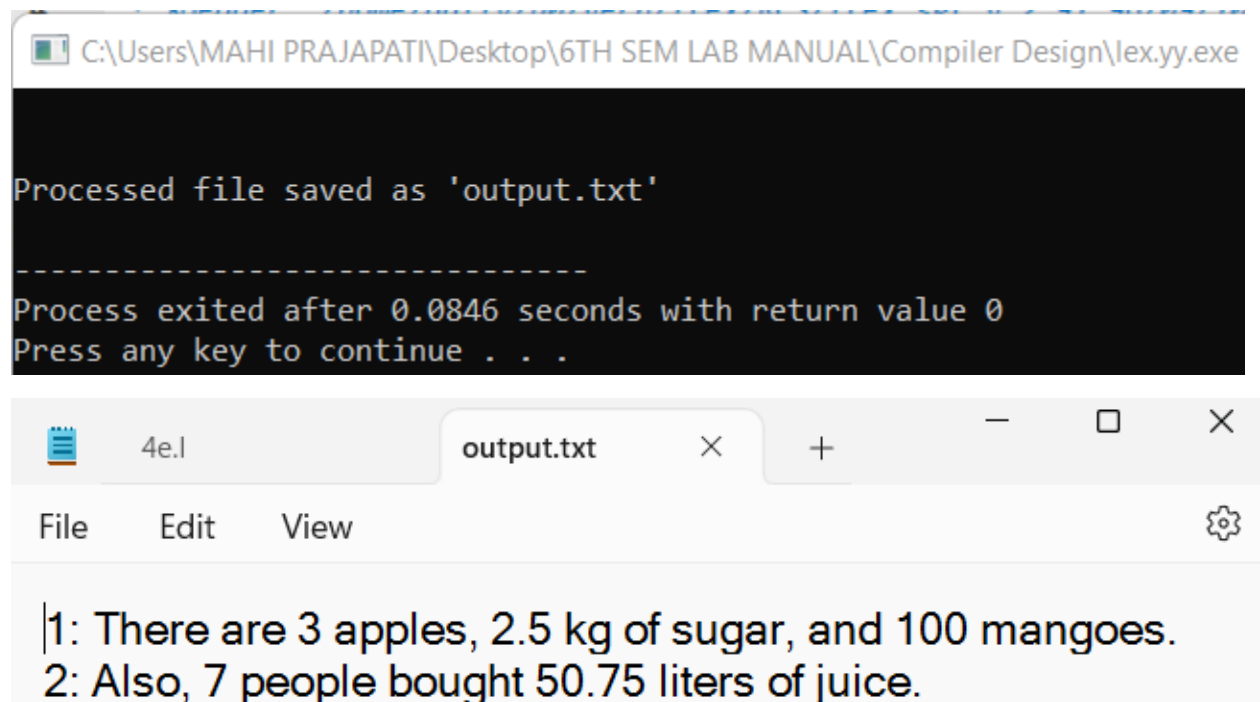
}

int yywrap() { return 1; }

O/P:

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design\lex.yy.exe

```
Processed file saved as 'output.txt'

-------------------------------
Process exited after 0.0846 seconds with return value 0
Press any key to continue . . .
```

4e.l   output.txt   ×   +

File   Edit   View

```
1: There are 3 apples, 2.5 kg of sugar, and 100 mangoes.
2: Also, 7 people bought 50.75 liters of juice.
```

e. **Write a Lex program to printout all markup tags and HTML comments in file.**

Code:

Lex.l:

%{

#include<stdio.h>

int num=0;

%}

%%

"<"[/A-Za-z0-9]+">" printf("%s is valid markup tag\n",yytext);

```
"<!--"[^--]*"-->"   num++;

\n    ;

.     ;

%%

int main()

{

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

yylex();

return 0;

}

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

myfile.txt:

```
<html>

<div>

<body>

</html>

<head>

<a>

</a>
```

O/P:

```
C:\Windows\System32\cmd.exe

C:\Users\MAHI PRAJAPATI\Downloads>flex cd4.l

C:\Users\MAHI PRAJAPATI\Downloads>gcc lex.yy.c -o tagprinter

C:\Users\MAHI PRAJAPATI\Downloads>tagprinter.exe
<html> is valid markup tag
<div> is valid markup tag
<body> is valid markup tag
</html> is valid markup tag
<head> is valid markup tag
<a> is valid markup tag
</a> is valid markup tag
```

Experiment – 5

a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.

Code:

Lex.l:

```
%{
#include <stdio.h>
int c = 0;  // Counter for comment lines
%}

%x COMMENT

%%

"//".*     { c++; }  // Count single-line comments

"/*"       { c++; BEGIN(COMMENT); }  // Start multi-line comment
<COMMENT>"*/" { BEGIN(INITIAL); }    // End multi-line comment
<COMMENT>\n    { c++; }              // Count each line in multi-line
comment
<COMMENT>.    { /* ignore comment content */ }

.|\n       { fprintf(yyout, "%s", yytext); }  // Copy non-comment text

%%

int main() {
   yyin = fopen("code.txt", "r");
   yyout = fopen("output.txt", "w");

   if (!yyin) {
      perror("Error opening input file");
      return 1;
   }
```

```
    if (!yyout) {
        perror("Error opening output file");
        return 1;
    }

    yylex();

    printf("Total comment lines removed: %d\n", c);

    fclose(yyin);
    fclose(yyout);
    return 0;
}

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

Code.txt:
```
#include <stdio.h>

// This is a single-line comment
int main() {
    printf("Hello, World!\n");  /* This is a block comment */
    return 0;
//gyhtyrt
}
```

O/P:

```
C:\Windows\System32\cmd.exe

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design>flex 5a.1

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design>gcc lex.yy.c -o commentremover

C:\Users\MAHI PRAJAPATI\Desktop\6TH SEM LAB MANUAL\Compiler Design>commentremover.exe
Total comment lines removed: 3
```

b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

Code:

Lex.l –

```
%{
#include <stdio.h>
%}

%%
"int"|"float"|"if"|"else"|"while"|"return"|"char"|"for"|"do"|"switch" {
printf("%s is a keyword\n", yytext); }
[aeiouAEIOU] {printf("%s is a vowels\n",yytext);}
[a-zA-Z_][a-zA-Z0-9_]* { printf("%s is an identifier\n", yytext); }
[ \t\n]+    ;  // This will ignore spaces, tabs, and newlines
%%
int main() {
    yyin=fopen("cd1.txt","r");
    yylex();
    return 0;
}
int yywrap() { return 1; }
```

txt file –
helo_12
An apple
Int

O/P:

C:\Windows\System32\cmd.exe

```
C:\Users\MAHI PRAJAPATI\Downloads>flex keywords.l

C:\Users\MAHI PRAJAPATI\Downloads>gcc lex.yy.c -o tagscanner

C:\Users\MAHI PRAJAPATI\Downloads>tagscanner.exe
helo_12 is an identifier
An is an identifier
apple is an identifier
int is a keyword
```

Experiment – 6

Aim: Program to implement Recursive Descent Parsing in C.

Code:

```c
#include<stdio.h>
#include<stdlib.h>
/*
E-> iE_
E_-> +iE_ / -iE_ / epsilon
*/
char s[20];
int i=1;
char l;
int match(char t)
{
   if(l==t){
      l=s[i];
      i++; }
   else{
     printf("Sytax error");
     exit(1);}
}
int E_()
{
   if(l=='+'){
      match('+');
```
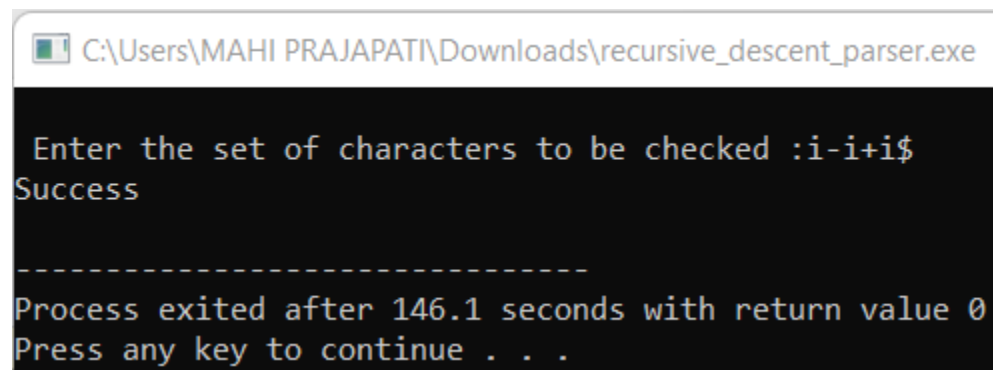
```
        match('i');

        E_(); }

    else  if(l=='-'){

        match('-');

        match('i');

        E_(); }

    else

        return(1);

}

int E()

{

   if(l=='i'){

        match('i');

        E_(); }

}


int main()

{

   printf("\n Enter the set of characters to be checked :");

   scanf("%s",&s);

   l=s[0];

   E();

   if(l=='$')

   {

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

```
    }
    else{
        printf("syntax error");
    }
    return 0;
}
```

O/P:

C:\Users\MAHI PRAJAPATI\Downloads\recursive_descent_parser.exe

```
 Enter the set of characters to be checked :i-i+i$
Success


--------------------------------
Process exited after 146.1 seconds with return value 0
Press any key to continue . . .
```

Experiment – 7
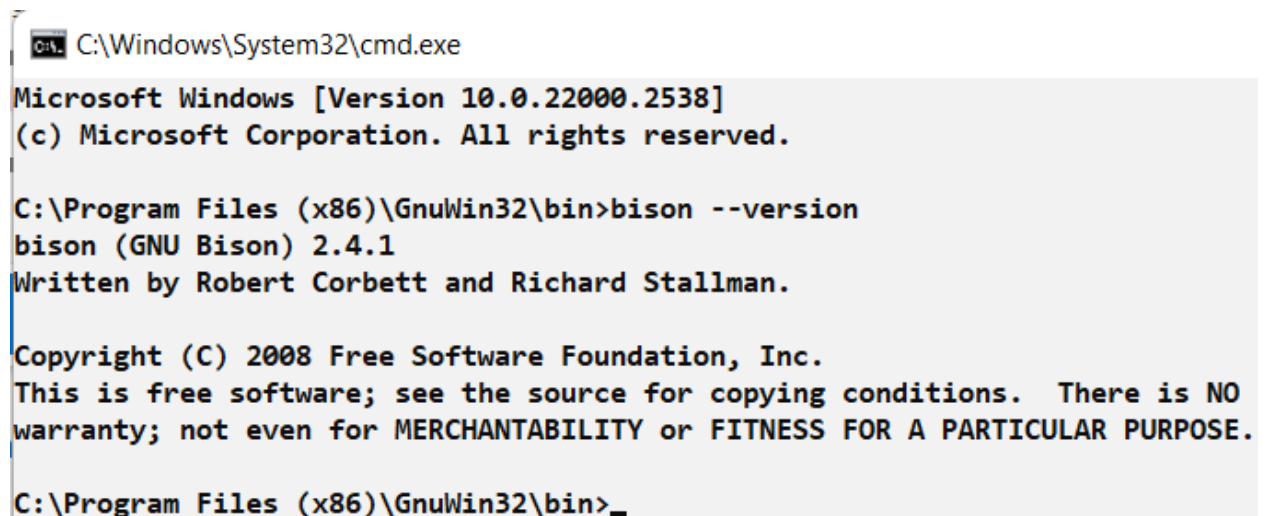
a. To Study about Yet Another Compiler-Compiler (YACC).

Introduction:

YACC (Yet Another Compiler-Compiler) is a parser generator developed to automate the process of creating the syntax analysis phase of a compiler. It reads a grammar specification written in a format similar to Backus-Naur Form (BNF) and generates a parser in the C programming language. This parser can recognize the syntax of a programming language and build syntax trees or perform semantic actions.

YACC is typically used in combination with **LEX/Flex**, where LEX handles lexical analysis and YACC handles syntax analysis. Together, they allow for the development of efficient and structured compilers. The grammar rules in YACC are written in the form of production rules, and associated C code (semantic actions) can be embedded within these rules to define what happens when a rule is recognized.

YACC Features:
- Supports LALR(1) parsing.
- Allows easy embedding of semantic actions using C code.
- Automatically handles parsing tables and parser generation.
- Provides error-handling mechanisms for syntax errors.
- Easily integrates with Flex for complete compiler front-end development.

```
C:\Windows\System32\cmd.exe

Microsoft Windows [Version 10.0.22000.2538]
(c) Microsoft Corporation. All rights reserved.

C:\Program Files (x86)\GnuWin32\bin>bison --version
bison (GNU Bison) 2.4.1
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

C:\Program Files (x86)\GnuWin32\bin>
```

b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .
Lex.l:

```
%{

#include <stdlib.h>

void yyerror(char *);

#include "yacc.tab.h"

%}
%%

[0-9]+ {yylval = atoi(yytext); return NUM;}

[a-zA-Z_][a-zA-Z_0-9]* {return id;}

[-+*/\n] {return *yytext;}

[ \t] { }

. yyerror("invalid character");
%%

int yywrap() {

 return 0;

}
```

yacc.y:

```
%{

 #include <stdio.h>

 int yylex(void);

 void yyerror(char *);

%}

%token NUM
```

```
%token id
%%
S: E '\n' { printf("Valid Syntax"); return(0); }
E: E '+' T  { }
 | E '-' T  { }
 | T       { }
T : T '*' F { }
 | T '/' F { }
 | F      { }
F:NUM      { }
 | id      { }
%%
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
int main() {
 yyparse();
 return 0;
}
```

O/P:

c. **Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.**
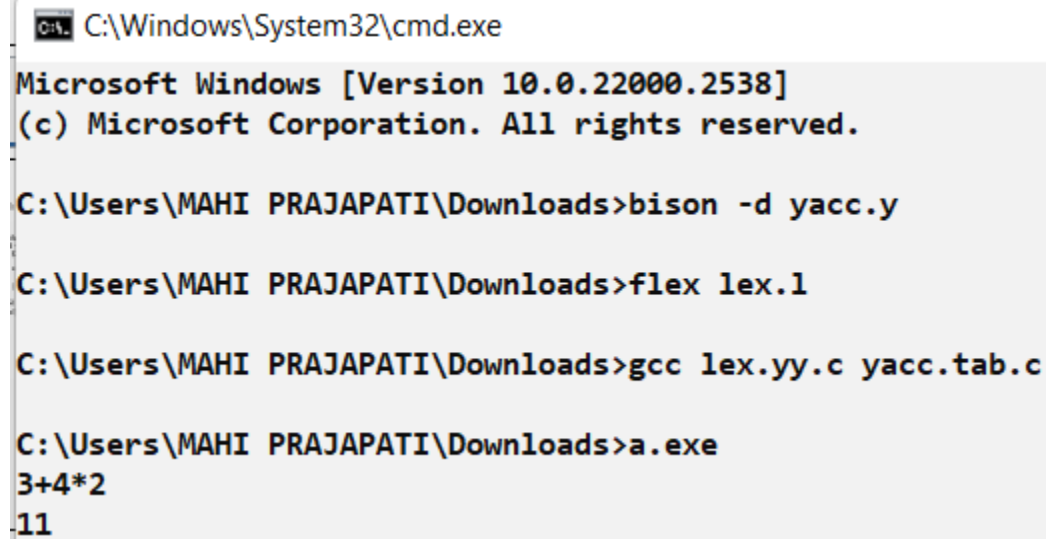**Code:**

Lex.l:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "yacc.tab.h"
%}
%%
[0-9]+ {yylval = atoi(yytext); return NUM;}
[-+*\n] {return *yytext;}
[ \t] { }
. yyerror("invalid character");
%%
int yywrap() {
 return 0;
}
```

yacc.y –

```
%{
 #include <stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
```

```
%token NUM
%%
S: E '\n' { printf("%d\n", $1); return(0); }
E: E '+' T   { $$ = $1 + $3; }
 | E '-' T  { $$ = $1 - $3; }
 | T      { $$ = $1; }
T : T '*' F { $$ = $1 * $3; }
 | F      { $$ = $1; }
F:NUM   { $$ = $1; }
%%
void yyerror(char *s) {
 fprintf(stderr, "%s\n", s);
}
int main() {
 yyparse();
 return 0;
}
```

O/P:


```
C:\Windows\System32\cmd.exe

Microsoft Windows [Version 10.0.22000.2538]
(c) Microsoft Corporation. All rights reserved.

C:\Users\MAHI PRAJAPATI\Downloads>bison -d yacc.y

C:\Users\MAHI PRAJAPATI\Downloads>flex lex.l

C:\Users\MAHI PRAJAPATI\Downloads>gcc lex.yy.c yacc.tab.c

C:\Users\MAHI PRAJAPATI\Downloads>a.exe
3+4*2
11
```

d. **Create Yacc and Lex specification files are used to convert infix**
   expression to postfix expression.
   Code:
   Lex.l –

```
%{
#include <stdlib.h>
```

58

```
#include "infix_to_postfix.tab.h"
void yyerror(char *);
%}
%%
[0-9]+ {yylval.num = atoi(yytext); return INTEGER;}
[A-Za-z_][A-Za-z0-9_]* {yylval.str = yytext; return ID; }
[-+*\n] {return *yytext;}
[ \t] ;
. yyerror("invalid character");
%%

int yywrap() {
 return 1;
}
```
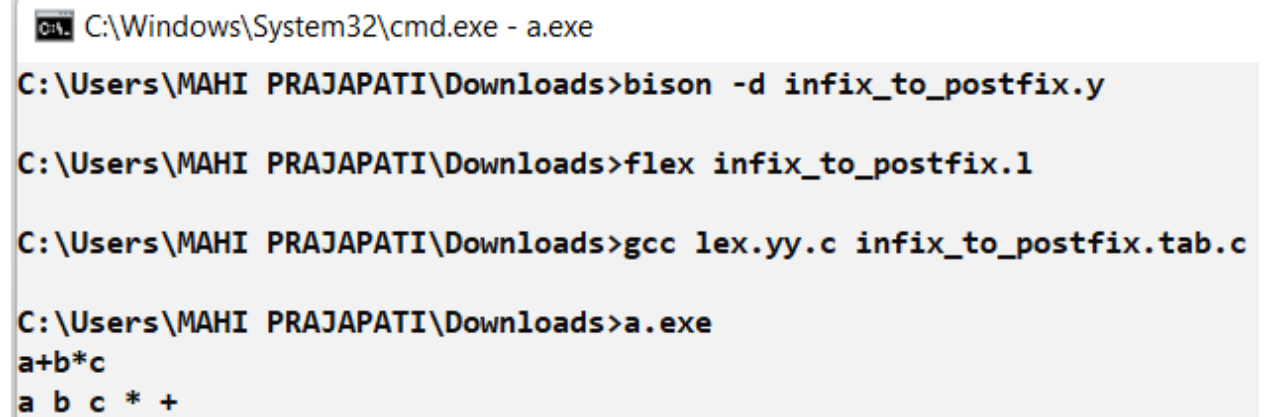
Yacc.y –

```
%{
 #include <stdio.h>
 int yylex(void);
 void yyerror(char *);
%}
%union {
      char *str;
      int num;
}
%token <num> INTEGER
%token <str> ID
%%
S: E '\n' { printf("\n"); }
E: E '+' T   { printf("+ "); }
 | E '-' T  { printf("- "); }
 | T        { }
T : T '*' F { printf("* "); }
 | F       { }
F: INTEGER  { printf("%d ",$1); }
 | ID      {  printf("%s ",$1); }
%%
void yyerror(char *s) {
```

```
 printf("%s\n", s);
}
int main() {
 yyparse();
 return 0;
}
```

O/P: