A

Assignment of

“Compiler Designer Laboratory”

Is Submitted to

School of Engineering and Technology

Toward the fulfilment of the requirements of the Subject

Compiler Designer Laboratory

(CSE606)

SUBMITED BY

Dev Ambekar 22000987

Subject In-Charge:- Prof. Vaibhavi Patel

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF ENGINEERING AND TECHNOLOGY

BHAYLI, VASNA-BHAYLI MAIN ROAD VADODARA

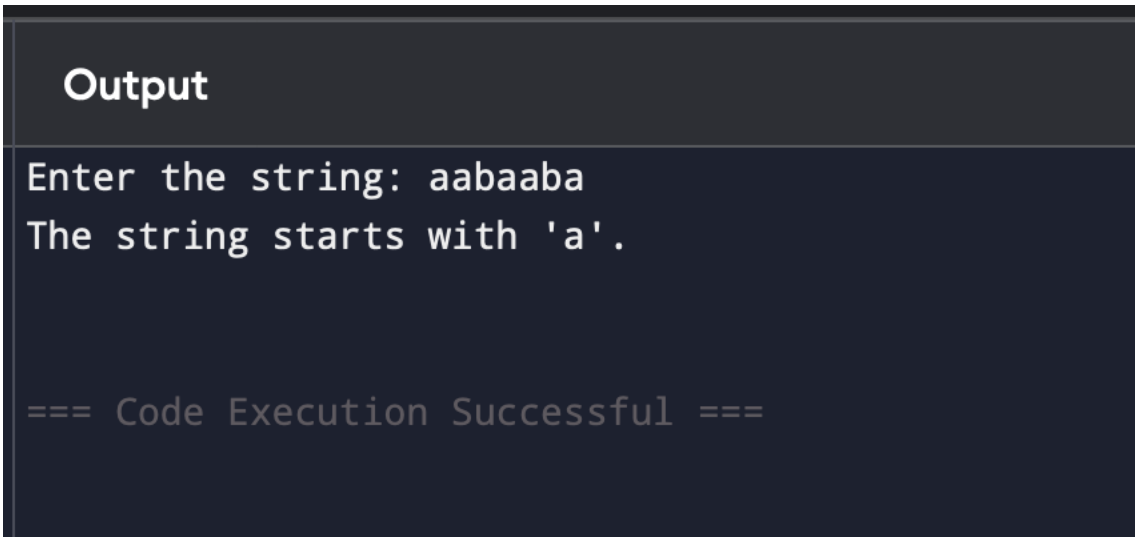
Task-1

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
D. text file.
E. Write a program to recognize strings contains ‘ab’. Take the input from
F. text file.

1).Code:

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

int main() {
   char str[100];
   printf("Enter the string: ");
   scanf("%s", str);

   if(str[0] == 'a') {
      printf("The string starts with 'a'.\n");
   } else {
      printf("The string does not start with 'a'.\n");
   }

   return 0;
}
```

Output:



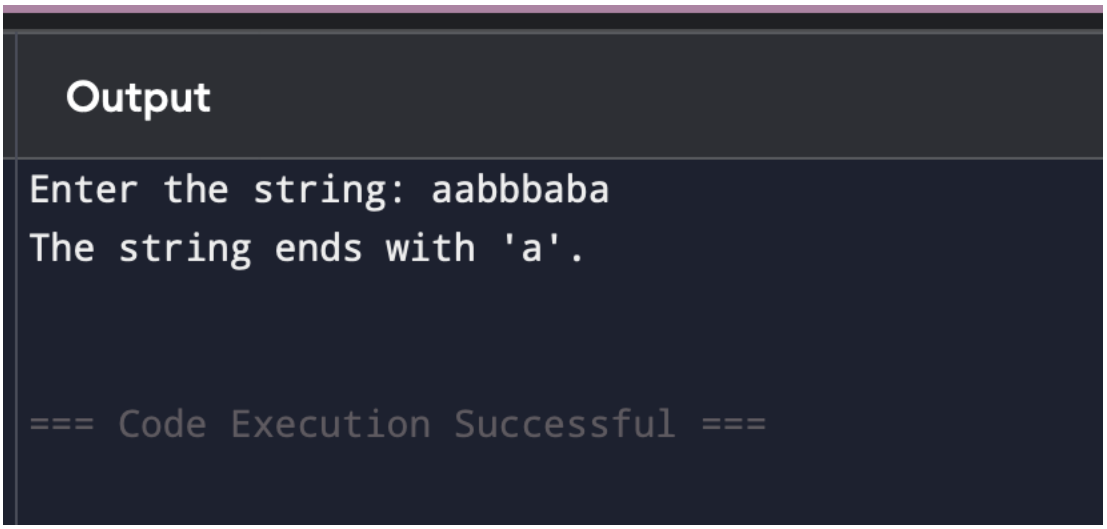
2). code:

#include <stdio.h>

#include <string.h>

```
int main() {
    char str[100];
    printf("Enter the string: ");
    scanf("%s", str);

    int len = strlen(str);
    if(len > 0 && str[len - 1] == 'a') {
        printf("The string ends with 'a'.\n");
    } else {
        printf("The string does not end with 'a'.\n");
    }

    return 0;
}
```

Output:

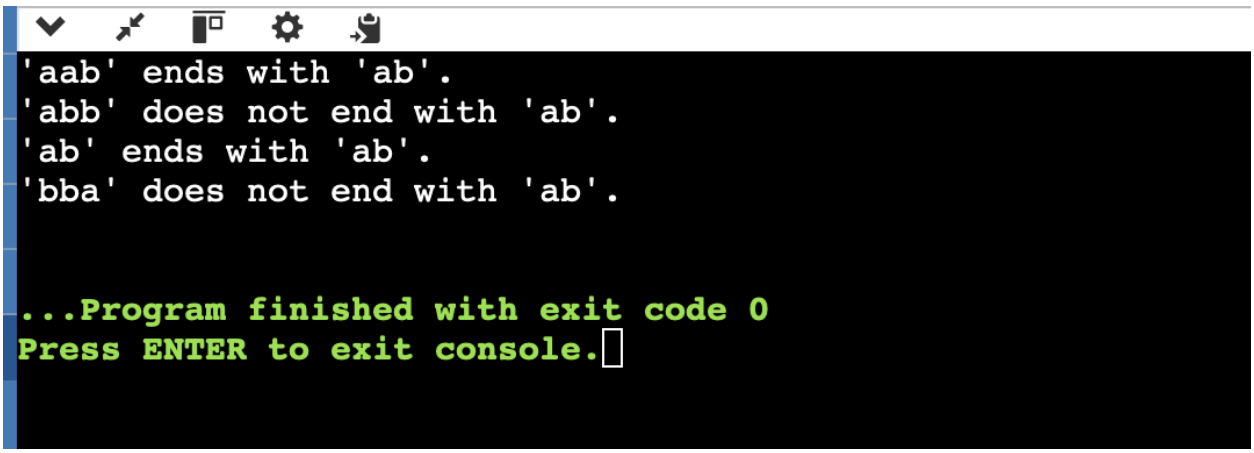


3). Code:

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

int main() {
FILE *file = fopen("input.txt", "r");
char str[100];

if (file == NULL) {
printf("Could not open input.txt\n");
return 1;
}

while (fgets(str, sizeof(str), file)) {
// Remove newline character
str[strcspn(str, "\n")] = '\0';
```

```
int len = strlen(str);


if(len >= 2 && str[len - 2] == 'a' && str[len - 1] == 'b') {
printf("'%s' ends with 'ab'.\n", str);
} else {
printf("'%s' does not end with 'ab'.\n", str);
}
}


fclose(file);
return 0;
}
```

Output:

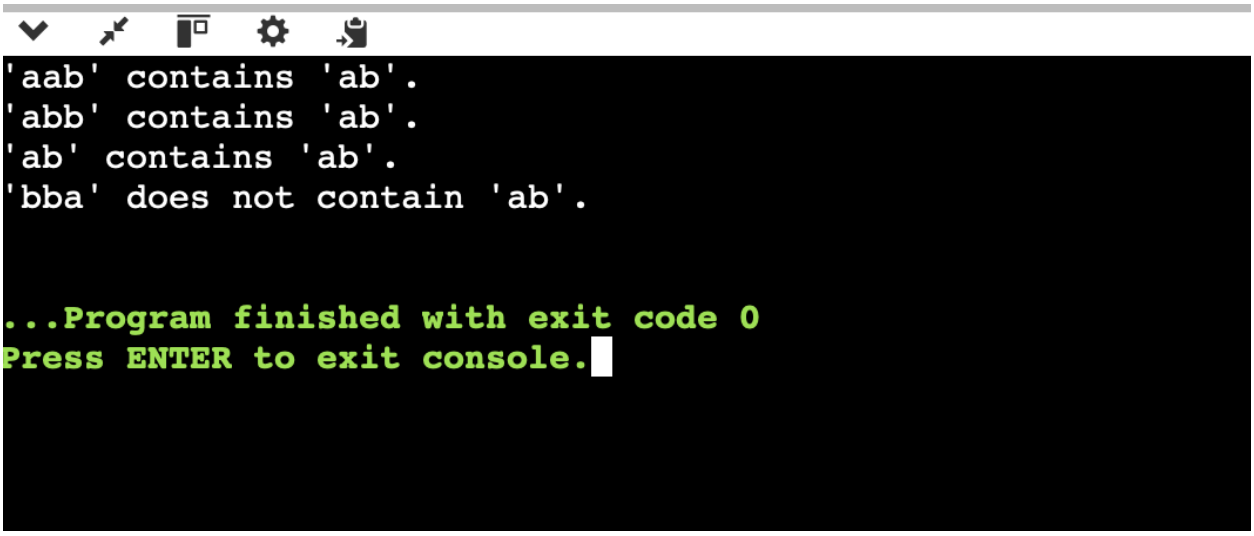


4).Code:

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

int main() {
   FILE *file = fopen("input.txt", "r");
   char str[100];

   if (file == NULL) {
      printf("Could not open input.txt\n");
      return 1;
   }

   while (fgets(str, sizeof(str), file)) {
      // Remove newline character
      str[strcspn(str, "\n")] = '\0';

      if (strstr(str, "ab") != NULL) {
         printf("'%s' contains 'ab'.\n", str);
      } else {
         printf("'%s' does not contain 'ab'.\n", str);
      }
   }
```

```
    fclose(file);
    return 0;
}
```

Output:



2).

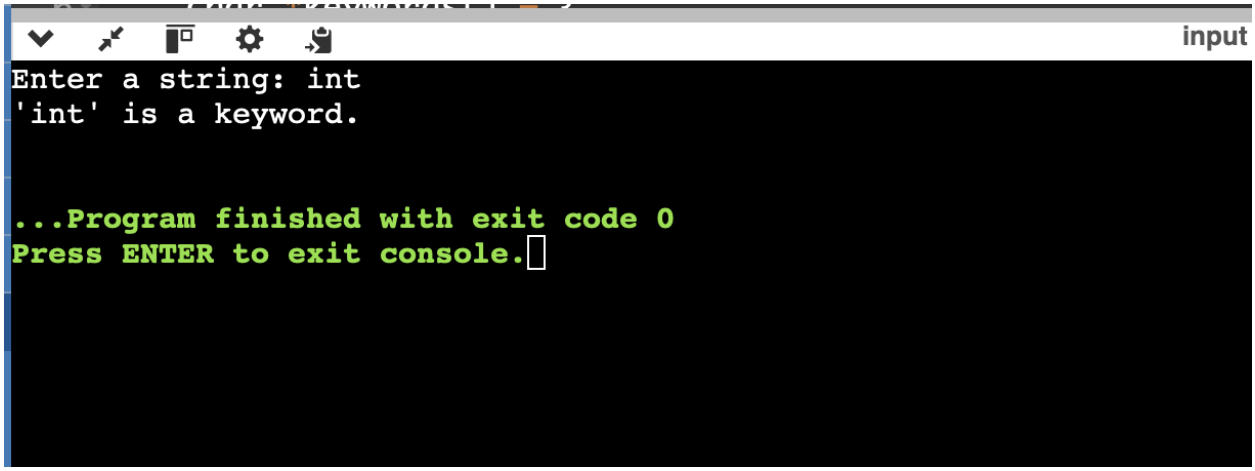
A. Write a program to recognize the valid identifiers and keywords.
B. Write a program to recognize the valid operators.
C. Write a program to recognize the valid number.
D. Write a program to recognize the valid comments.
E. Program to implement Lexical Analyzer.

A).Code:

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

int isKeyword(char *word) {
    char *keywords[] = {
        "int", "float", "if", "else", "while", "return", "for", "do", "break",
        "continue", "char", "double", "long", "short", "void", "switch", "case"
    };
    for (int i = 0; i < 17; i++) {
        if (strcmp(keywords[i], word) == 0)
            return 1;
    }
    return 0;
}

int isValidIdentifier(char *str) {
    if (!isalpha(str[0]) && str[0] != '_')
        return 0;
    for (int i = 1; str[i]; i++) {
        if (!isalnum(str[i]) && str[i] != '_')
            return 0;
```

```
    }
    return 1;
}

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    if (isKeyword(str))
        printf("'%s' is a keyword.\n", str);
    else if (isValidIdentifier(str))
        printf("'%s' is a valid identifier.\n", str);
    else
        printf("'%s' is not a valid identifier.\n", str);

    return 0;
}
```

Output:

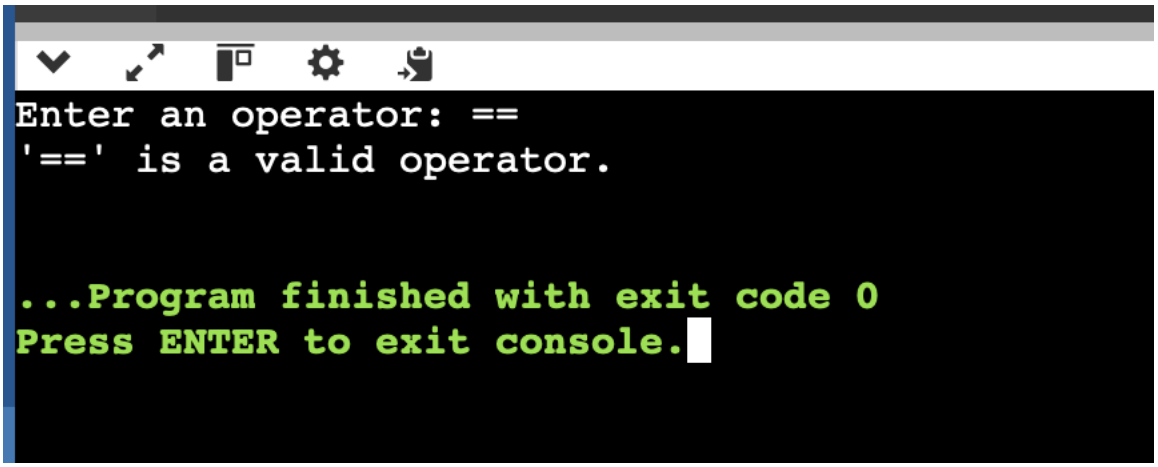


B).Code:

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

int isOperator(char *op) {
   char *operators[] = {
      "+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||", "!", "++", "--"
   };
   for (int i = 0; i < 17; i++) {
      if (strcmp(operators[i], op) == 0)
         return 1;
   }
   return 0;
}

int main() {
   char op[3];
   printf("Enter an operator: ");
   scanf("%s", op);

   if (isOperator(op))
```

```
        printf("'%s' is a valid operator.\n", op);
    else
        printf("'%s' is not a valid operator.\n", op);

    return 0;
}
```

Output:

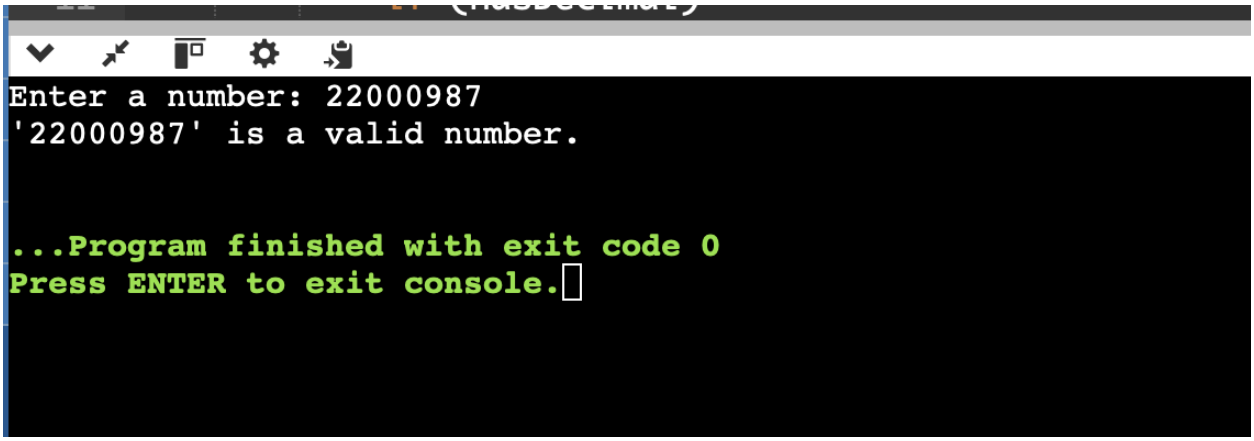


C). Code:

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

int isValidNumber(char *str) {
    int i = 0, hasDecimal = 0;

    if (str[i] == '-' || str[i] == '+') i++;

    for (; str[i]; i++) {
        if (str[i] == '.') {
            if (hasDecimal)
```

```
            return 0;
        hasDecimal = 1;
      } else if (!isdigit(str[i])) {
        return 0;
      }
    }

    return i > 0;
}

int main() {
    char str[100];
    printf("Enter a number: ");
    scanf("%s", str);

    if (isValidNumber(str))
        printf("'%s' is a valid number.\n", str);
    else
        printf("'%s' is not a valid number.\n", str);

    return 0;
}
```

Output:

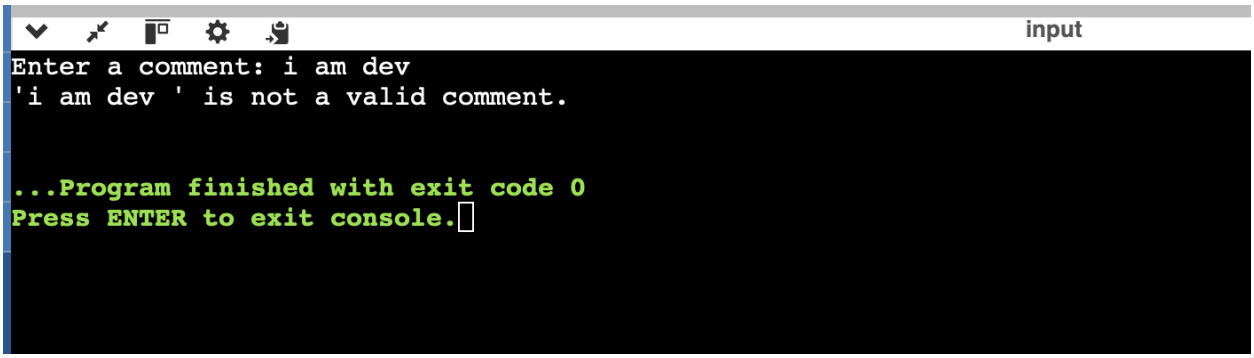


D).code:

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

int isValidComment(char *str) { int len = strlen(str);

if ((str[0] == '/' && str[1] == '/') ||
  (str[0] == '/' && str[1] == '*' && str[len-2] == '*' && str[len-1] == '/')) {
  return 1;
}

return 0;


}

int main() { char str[200]; printf("Enter a comment: "); scanf(" %[^\n]s", str);

if (isValidComment(str))
  printf("'%s' is a valid comment.\n", str);
else
  printf("'%s' is not a valid comment.\n", str);
```

```
return 0;

}
```

Output:



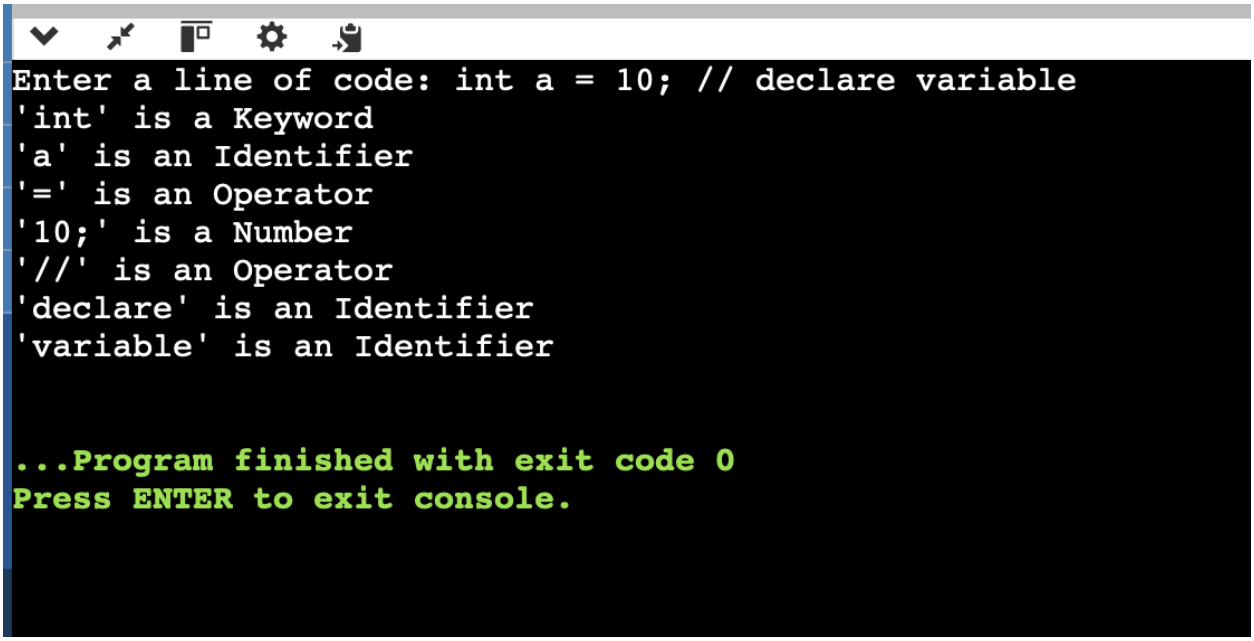
E).Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


int isKeyword(char *word) {
   char *keywords[] = {"int", "float", "if", "else", "while", "return", "char", "for", "do"};
   for (int i = 0; i < 9; i++)
      if (strcmp(word, keywords[i]) == 0)
         return 1;
   return 0;
}


int isOperator(char ch) {
   return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '=' || ch == '<' || ch == '>';
}
```

```
int main() {
    char input[200];
    printf("Enter a line of code: ");
    fgets(input, sizeof(input), stdin);

    char *token = strtok(input, " \t\n");

    while (token) {
        if (isKeyword(token)) {
            printf("'%s' is a Keyword\n", token);
        } else if (isalpha(token[0]) || token[0] == '_') {
            printf("'%s' is an Identifier\n", token);
        } else if (isdigit(token[0])) {
            printf("'%s' is a Number\n", token);
        } else if (isOperator(token[0])) {
            printf("'%s' is an Operator\n", token);
        } else if (token[0] == '/' && token[1] == '/') {
            printf("'%s' is a Single-line Comment\n", token);
        } else {
            printf("'%s' is Unknown Token\n", token);
        }

        token = strtok(NULL, " \t\n");
    }
```

```
    return 0;
}
```

Output:



3). To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)

What is a Lexical Analyzer?

A **Lexical Analyzer** (also called a **scanner** or **lexer**) is the **first phase of a compiler**. It processes the source code to break it into **tokens**, which are meaningful character sequences such as identifiers, keywords, numbers, operators, and symbols. These tokens are passed to the **parser** for syntax analysis.

Functions of a Lexical Analyzer:

- Removes whitespace and comments
- Recognizes tokens
- Provides token information to the parser
- Reports lexical errors

◇ Lexical Analyzer Generator:

A **Lexical Analyzer Generator** is a tool that **automatically generates** source code (usually in C or C++) to perform lexical analysis. Instead of manually writing a lexical analyzer, developers write regular expressions and actions in a specific format, and the generator creates the scanner code.

◇ LEX: Lexical Analyzer Generator

☑ Definition:

LEX is a tool for generating lexical analyzers. It was developed in the early 1970s by **Mike Lesk and Eric Schmidt** at **AT&T Bell Labs**.

☑ Purpose:

LEX helps in **automatically generating** C code for lexical analysis from regular expressions.

☑ Structure:

LEX programs are divided into **three sections**:

1. **Definition Section (%{ ... %})** – Contains header files or macros.
2. **Rules Section (%%)** – Each line defines a **pattern** (regular expression) and an **action** (C code).
3. **Code Section (%%)** – Optional user-defined C functions (e.g., main()).

☑ Output:

The LEX tool generates a file called lex.yy.c containing the C source code of the lexer.

◇ Flex: Fast Lexical Analyzer Generator

☑ Definition:

Flex (Fast Lex) is an **improved version of LEX**, created to be faster, more powerful, and more portable. It is open-source and commonly used in Unix/Linux environments.

☑ Features:

- **Compatible** with LEX
- **Faster and more efficient**
- **Open-source** and actively maintained
- Generates C code from .l files
- Provides better **error handling and debugging tools**

☑ Output:

Like LEX, Flex generates a lex.yy.c file, which is then compiled using a C compiler (e.g., gcc) to create the lexical analyzer.

◇ Working Process of LEX/Flex

1. **Write a Lex/Flex file** (e.g., scanner.l) with token patterns.
2. Run:
a. lex scanner.l or flex scanner.l
3. Compile:
a. gcc lex.yy.c -o scanner -lfl
4. Execute:
a. ./scanner
5. The analyzer reads input, matches patterns, and executes actions.

◇ Regular Expressions in LEX/Flex

- [a-z] – any lowercase letter
- [0-9]+ – one or more digits
- int|float – matches int or float
- [\t\n]+ – matches whitespace

◇ Advantages of Using LEX/Flex

- Simplifies scanner development
- Reduces manual errors
- Supports complex pattern matching
- Easily integrates with **YACC/Bison** (parser generators)

◇ Applications

- Compilers (tokenizing source code)
- Interpreters for custom scripting languages
- Syntax highlighting tools
- Static code analyzers
- Log file analyzers

◇ Conclusion

LEX and Flex are powerful tools in the domain of compiler design and language processing. They automate the process of building lexical analyzers using regular expressions and actions, drastically reducing development time and errors. While LEX is the historical tool, Flex is the modern, efficient alternative preferred in most practical applications today.

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.

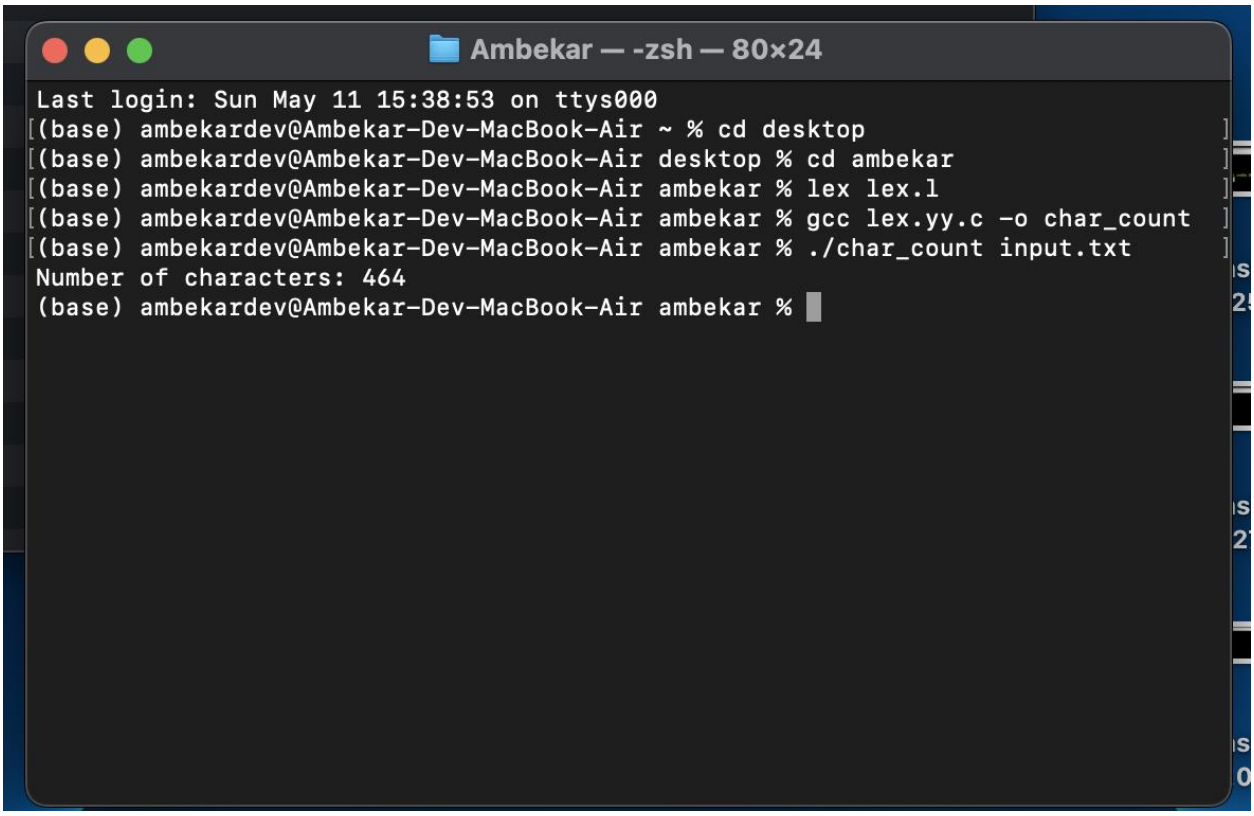
E). Write a Lex program to printout all markup tags and HTML comments in file

A).Code:

```
%{
#include <stdio.h>
int char_count = 0;
%}

%%

.|\n    { char_count++; }

%%

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if (!file) {
        printf("Error opening file %s\n", argv[1]);
        return 1;
```

```
    }

    yyin = file;
    yylex();
    fclose(file);

    printf("Number of characters: %d\n", char_count);
    return 0;
}

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

Output:



B). Code:

```
%{
#include <stdio.h>
int vowels = 0;
int consonants = 0;
%}

%%

[aAeEiIoOuU]    { vowels++; }
[bBcCdDfFgGhHjJkKlLmMnNpPqQrRsStTvVwWxXyYzZ]  { consonants++; }
.|\n           { }

%%
```

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if (!file) {
        printf("Error opening file %s\n", argv[1]);
        return 1;
    }

    yyin = file;
    yylex();
    fclose(file);

    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);
    return 0;
}

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

Output:

```
(base) ambekardev@Ambekar-Dev-MacBook-Air ambekar % lex lex.l
(base) ambekardev@Ambekar-Dev-MacBook-Air ambekar % gcc lex.yy.c -o vowel_conson
ant_count
(base) ambekardev@Ambekar-Dev-MacBook-Air ambekar % ./vowel_consonant_count inpu
t.txt
Number of vowels: 84
Number of consonants: 195
(base) ambekardev@Ambekar-Dev-MacBook-Air ambekar %
```

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

Code:

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

%%

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

%%

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }
```

```
    FILE *file = fopen(argv[1], "r");
    if (!file) {
        printf("Error opening file %s\n", argv[1]);
        return 1;
    }

    yyin = file;
    yylex();
    fclose(file);

    return 0;
}

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

Output:

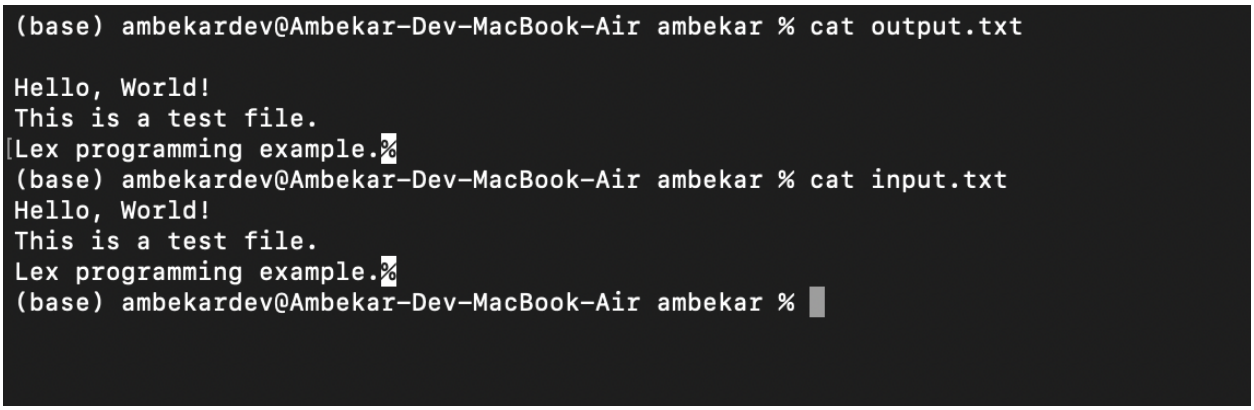
D). Write a Lex program which adds line numbers to the given file and display the same into different file.

Code:

```
%{ #include <stdio.h> int line_number = 1; FILE *output_file; %}

%%

\n { fprintf(output_file, "%d: %s", line_number++, yytext); }

. { fprintf(output_file, "%s", yytext); }

%%

int main(int argc, char *argv[]) { if (argc != 3) { printf("Usage: %s input_file output_file\n", argv[0]); return 1; }

FILE *input_file = fopen(argv[1], "r");
if (!input_file) {
   printf("Error opening input file\n");
   return 1;
}

output_file = fopen(argv[2], "w");
if (!output_file) {
   printf("Error opening output file\n");
   fclose(input_file);
   return 1;
}

yyin = input_file;
yylex();

fclose(input_file);
fclose(output_file);
return 0;


}
```

int yywrap() { return 1; }

Output:



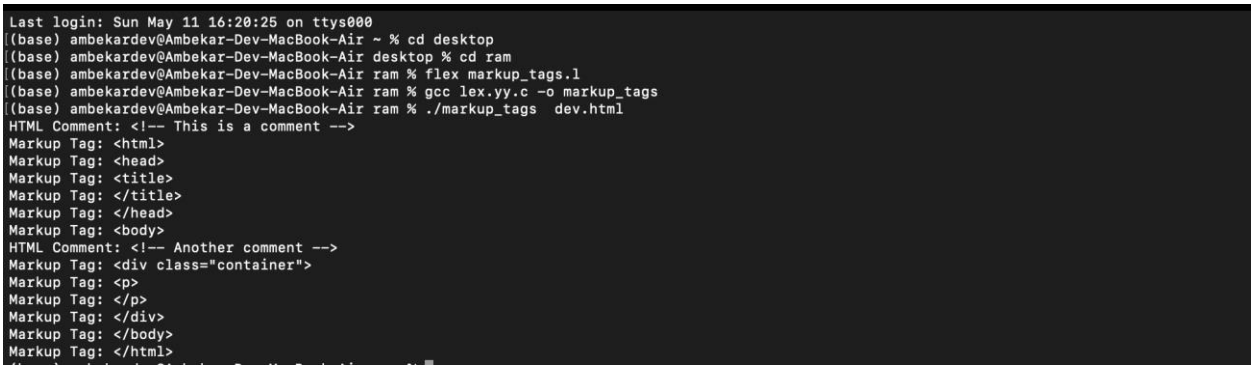
E). Write a Lex program to printout all markup tags and HTML comments in file

Code:

```
%option noyywrap

%{

#include <stdio.h>

%}


%%

"<!--"([^-\n]|"-"[^-]|"--"[^>])*"-->"    { printf("HTML Comment: %s\n", yytext); }

"<"[^>]+">"                              { printf("Markup Tag: %s\n", yytext); }


.|\n                                     ; // Skip all other characters

%%
```

```
int main(int argc, char **argv)
{
    if (argc > 1) {
        FILE *file = fopen(argv[1], "r");
        if (!file) {
            perror("Error opening file");
            return 1;
        }
        yyin = file;
    }
    yylex();
    return 0;
}
```

Output:



5).

A. Write a Lex program to count the number of C comment lines from a given
B. program. Also eliminate them and copy that program into separate file.
C. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

A) Lex code:

%{ #include <stdio.h> int commentCount = 0; %}

%% //.* { commentCount++; } /* Matches single-line comments / /*([^]|*+[^*/])*/ { *commentCount++; }* / Matches multi-line comments / .|\n { / Ignore everything else */ } %%

int main(int argc, char *argv[]) { if (argc != 2) { printf("Usage: %s \n", argv[0]); return 1; }

FILE *file = fopen(argv[1], "r");
if (!file) {
printf("Error opening file.\n");
return 1;
}

yyin = file;
yylex();
fclose(file);

printf("Total number of comment lines: %d\n", commentCount);
return 0;

}

Dev.c:

```
#include <stdio.h>

int main() {
    // This is a single-line comment

    /*
       This is a multi-line comment
       spanning multiple lines
    */

    printf("Hello, World!\n"); // Another comment

    /*
       Another multi-line comment
    */

    return 0;
}
```

Output:

B).program. Also eliminate them and copy that program into separate file.

Code:

```
%{ #include <stdio.h> int commentCount = 0; FILE *outFile; %}

%% //.* { commentCount++; } /* Single-line comments (ignored) / /*([^]|*+[^*/])*/ { commentCount++; } / Multi-line comments (ignored) / .|\n { fprintf(outFile, "%s", yytext); } / Copy non-comment content */ %%

int main(int argc, char *argv[]) { if (argc != 2) { printf("Usage: %s \n", argv[0]); return 1; }

FILE *file = fopen(argv[1], "r");
if (!file) {
   printf("Error opening file.\n");
   return 1;
}

outFile = fopen("output.c", "w");
if (!outFile) {
   printf("Error creating output file.\n");
   return 1;
}

yyin = file;
yylex();

fclose(file);
fclose(outFile);

printf("Total comment lines removed: %d\n", commentCount);
printf("Cleaned file saved as 'output.c'\n");

return 0;


}
```

Output:

```
(base) ambekardev@Ambekar-Dev-MacBook-Air ram % flex comments.l
(base) ambekardev@Ambekar-Dev-MacBook-Air ram % gcc lex.yy.c -o remove_comments -ll
ld: warning: object file (/Library/Developer/CommandLineTools/SDKs/MacOSX15.4.sdk/usr/lib/libl.a[arm64][3](libyywrap.o)) was built for newer 'macOS' version (15.4) than being linked (15.0)
(base) ambekardev@Ambekar-Dev-MacBook-Air ram % ./remove_comments dev.c
Total comment lines removed: 4
Cleaned file saved as 'output.c'
(base) ambekardev@Ambekar-Dev-MacBook-Air ram %
```

c). Write a Lex program to recognize keywords, identifiers, operators,

numbers, special symbols, literals from a given C program.

Code:

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

// Define a list of C keywords char *keywords[] = { "auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register", "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while" };

int is_keyword(char *word) { for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) { if (strcmp(word, keywords[i]) == 0) return 1; } return 0; } %}

%%

[ \t\n]+ ; // Ignore whitespace

"([^"\]|\.)*" { printf("Literal (string): %s\n", yytext); } '([^'\]|\.)*' { printf("Literal (char): %s\n", yytext); }

"==|!=|<=|>=|&&||||++|--" { printf("Operator: %s\n", yytext); } "[-+*/%=<>&|!^]" { printf("Operator: %s\n", yytext); }

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

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

[a-zA-Z_][a-zA-Z0-9_]* { if (is_keyword(yytext)) printf("Keyword: %s\n", yytext); else printf("Identifier: %s\n", yytext); }

. { printf("Unrecognized token: %s\n", yytext); }

%%

int main(int argc, char **argv) { printf("Lexical Analysis Output:\n\n"); yylex(); return 0; }

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

Main.c code:

```
#include <stdio.h>

int main() {
```

```
    int a = 5;
    if (a > 3) {
        printf("Hello, world!");
    }
    return 0;
}
```

Output:

6). Program to implement Recursive Descent Parsing in C.

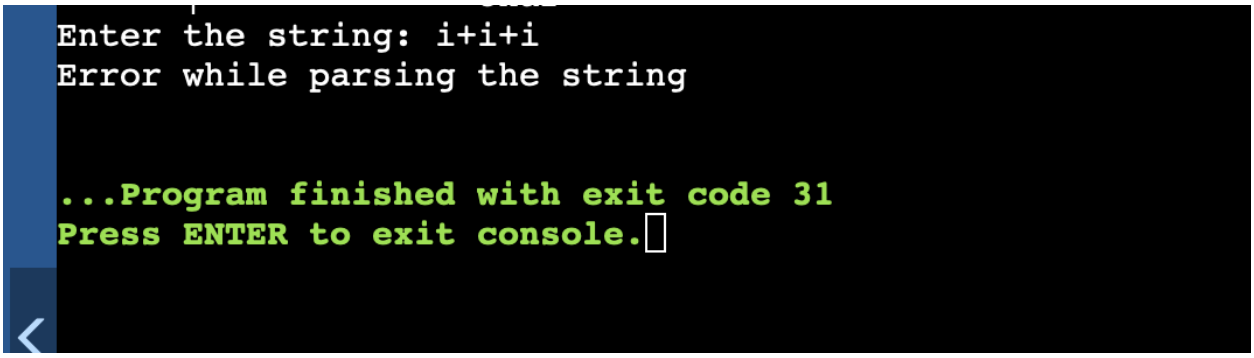
Code:

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

char s[20];
int i = 1;
char l;
int match(char l);
int E1();
int E()
{
   if (l == 'i')
   {
      match('i');
      E1();
   }
   else
   {
      printf("Error parsing string");
      exit(1);
   }
   return 0;
```

```
}

int E1()
{
   if (l == '+')
   {
      match('+');
      match('i');
      E1();
   }
   else
   {
      return 0;
   }
}

int match(char t)
{
   if (l == t)
   {
      l = s[i];
      i++;
   }
   else
```

```
    {
        printf("Syntax Error");
        exit(1);
    }
    return 0;
}
void main()
{
    printf("Enter the string: ");
    scanf("%s", &s);
    l = s[0];
    E();
    if (l == '$')
    {
        printf("parsing successful");
    }
    else
    {
        printf("Error while parsing the string\n");
    }
}
```

Output:



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.

A). Study of Yet Another Compiler-Compiler (YACC)

Study of Yet Another Compiler-Compiler (YACC)

YACC (Yet Another Compiler-Compiler) is a tool that generates parsers, specifically for context-free grammars, which are commonly used in the design and implementation of compilers and interpreters. It is a parser generator that takes a formal grammar description and produces a parser that can process input strings according to that grammar.

YACC is primarily used for creating syntax analyzers (parsers) in the context of compilers and other systems that require the interpretation or validation of structured text (e.g., programming languages, configuration files, etc.).

Introduction to YACC

- YACC is a tool that provides a high-level method for defining parsers. It is a LALR(1) parser generator, meaning it constructs parsers that follow Look-Ahead, Left-to-Right parsing strategy with a 1-symbol lookahead. The tool is often used with Lex, a tool that generates lexical analyzers (scanners), to build compilers and interpreters.

- A typical YACC input is a file that contains:
- A grammar specification (productions).
- Action code (usually C code) that is executed when a rule is matched.
- Optionally, semantic actions that are linked to the grammar rules.
- YACC is widely used in the development of compilers, interpreters, static analysis tools, and other applications that require parsing of structured text.

Components of YACC

- YACC operates in three main parts:
- Input Grammar: Defines the syntax of the language being parsed (often described using Backus-Naur Form or Context-Free Grammar).
- Semantic Actions: The code to be executed when a grammar rule is matched. These are typically written in C and can perform tasks such as creating a parse tree or checking for semantic errors.
- Parser: The parser generated by YACC based on the grammar and semantic actions.

YACC Syntax and Structure

- A YACC file typically has three sections:

- Declarations Section (Optional): Contains C code for including libraries, defining tokens, or defining external variables.
- Grammar Rules Section: Contains the grammar rules, with C code embedded to perform actions when rules are matched.
- Code Section: Contains any C code to be used for the parser's operations, such as helper functions and additional declarations.

Here is a simple YACC example that parses basic arithmetic expressions.

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

%token NUM

%%

expr: expr '+' term { printf("Add operation\n"); }
   | term { printf("Single term\n"); }
   ;

term: NUM { printf("Term: %d\n", $1); }
   ;

%%

int main(void) {
   printf("Enter an expression:\n");
   yyparse();
   return 0;
}

int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

In this example:

The grammar rules define an expression (expr) that can either be a term or a term followed by a + operator and another term.

The NUM token represents a number and is matched in the term rule.

The C code embedded inside {} gets executed whenever a rule is matched, such as printing messages when specific operations occur.

Steps to Use YACC

- The general steps for using YACC to create a parser are:
- Write the YACC file: Create a .y file that contains the grammar rules and C code for actions.
- Run YACC: Process the .y file using the YACC tool, which generates a C file (usually y.tab.c).

Example:

yacc -d example.y

- This command generates y.tab.c (the parser code) and y.tab.h (the header file with token definitions).
- Write a Lex file (optional): If your parser depends on a lexer (such as to handle tokens like NUM in the example), write a .l file using the Lex tool.
- Compile the generated C code: Compile the generated parser code along with any Lex-generated code (usually lex.yy.c) to create the executable.

Example:

gcc -o parser y.tab.c lex.yy.c -ll

Run the parser: Execute the compiled parser to analyze input according to the grammar.

Example:

./parser

5.

Advantages of YACC

- High-Level Specification: YACC allows defining complex grammars in a concise, readable manner, avoiding manual management of parsing tables or other low-level operations.
- Integration with Lex: YACC works seamlessly with Lex to generate both the lexical analyzer and parser for a compiler.
- Automation: YACC automates much of the process of parsing, allowing developers to focus on grammar rules and semantic actions.
- Flexibility: YACC can be customized with C code to perform various tasks such as building abstract syntax trees, symbol tables, or handling errors.

Applications of YACC

- YACC is commonly used in various domains, including:
- Compiler Design: For building parsers for programming languages.
- Interpreter Design: Used for building interpreters for custom languages or domain-specific languages (DSLs).
- Static Analysis Tools: To analyze source code for patterns, errors, or optimizations.
- Data Validation: For validating structured data formats (e.g., XML, JSON).
- Protocol Parsing: In network communication protocols where the data structure follows a well-defined grammar.

Common Errors in YACC

- Syntax Errors: Errors in the grammar rules, such as mismatched parentheses or incorrect rule definitions.
- Shift/Reduce Conflicts: Occurs when YACC cannot decide whether to shift a symbol or reduce it by a rule. This often happens in ambiguous grammars.
- Reduce/Reduce Conflicts: When there are multiple rules that could match the same input, YACC may encounter ambiguity in deciding which rule to reduce.

- To resolve conflicts, the user can manually adjust the grammar, use precedence and associativity rules, or employ techniques like rewriting ambiguous rules.

Conclusion

YACC is an essential tool for compiler construction, offering an efficient and automated way to generate parsers from formal grammar definitions. While YACC handles the parsing process, it is typically used in conjunction with Lex (a lexical analyzer generator) to handle tokenization of input strings. YACC simplifies complex grammar specifications, making it easier to develop compilers, interpreters, and other applications that require structured text parsing.

b).

Calc.y code:

```
%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex(void);

extern void yyerror(const char *s);

%}


%token NUMBER

%left '+' '-'

%left '*' '/'


%%

program:
```

```
    expr '\n' { printf("Valid arithmetic expression\n"); return 0; }
    ;

expr:
    NUMBER
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    ;

%%
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
    return yyparse();
}
```

calc.l

```
% {
#include "y.tab.h"
 %}
```

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; } "+" { return '+'; } "-" { return '-'; } "*" { return '*'; } "/" { return '/'; } [\t] ; /* skip whitespace */ \n { return '\n'; } . { fprintf(stderr, "Invalid character: %s\n", yytext); return 0; }

%%

int yywrap(void) {

return 1; }

Output:

```
5 + 3
Valid arithmetic expression
10 * 2 - 4
Valid arithmetic expression
3 + * 2
Error: syntax error
```

c). Create Yacc and Lex specification files are used to generate a calculator

which accepts integer type arguments.

Calc.y code:

%{ #include #include extern int yylex(void); extern void yyerror(const char *s); %}

%union { int ival; }

%token NUMBER %type expr %left '+' '-' %left '*' '/'

%% program: program expr '\n' { printf("Result: %d\n", $2); } | /* empty */ ;

expr: NUMBER { $$ = $1; } | expr '+' expr { $$ = $1 + $3; } | expr '-' expr { $$ = $1 - $3; } | expr '*' expr { $$ = $1 * $3; } | expr '/' expr { if ($3 == 0) { yyerror("Division by zero"); exit(1); } $$ = $1 / $3; } ;

%% void yyerror(const char *s) { fprintf(stderr, "Error: %s\n", s); }

int main(void) { return yyparse(); }

Calc.l :

```
%{
 #include "y.tab.h"
%}
%%
 [0-9]+ { yylval.ival = atoi(yytext); return NUMBER; } "+" { return '+'; } "-" { return '-'; } "″ { return ″'; } "/" { return '/'; } [ \t] ; /* skip whitespace */ \n { return '\n'; } . { fprintf(stderr, "Invalid character: %s\n", yytext); return 0; } %%
int yywrap(void) {
 return 1;
 }
```

Output:

```
5 + 3
Result: 8
10 * 2 - 4
Result: 16
6 / 2
Result: 3
2 + 3 * 4
Result: 14
6 / 0
Error: Division by zero
2 + a
Invalid character: a
3 + + 2
Error: syntax error
```

d. Create Yacc and Lex specification files are used to convert infix expression

to postfix expression.

infix_to_postfix.y

infix_to_postfix.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int yylex(void);
extern void yyerror(const char *s);
char postfix[100];
void append(char *str, char c);
%}

%union {
  int ival;
  char *str;
}

%token <ival> NUMBER
%type <str> expr
%left '+' '-'
%left '*' '/'

%%
program:
  program expr '\n' { printf("Postfix: %s\n", $2); free($2); }
  | /* empty */
  ;

expr:
  NUMBER { char *s = malloc(10); sprintf(s, "%d", $1); $$ = s; }
  | expr '+' expr { $$ = malloc(strlen($1) + strlen($3) + 2); strcpy($$, $1); strcat($$, $3); strcat($$,
"+"); free($1); free($3); }
  | expr '-' expr { $$ = malloc(strlen($1) + strlen($3) + 2); strcpy($$, $1); strcat($$, $3); strcat($$,
"-"); free($1); free($3); }
  | expr '*' expr { $$ = malloc(strlen($1) + strlen($3) + 2); strcpy($$, $1); strcat($$, $3); strcat($$,
"*"); free($1); free($3); }
  | expr '/' expr { $$ = malloc(strlen($1) + strlen($3) + 2); strcpy($$, $1); strcat($$, $3); strcat($$,
"/"); free($1); free($3); }
  ;
```

```
%%
void yyerror(const char *s) {
  fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
  return yyparse();
}
```

infix_to_postfix.l

```
%{
#include "y.tab.h"
%}

%%
[0-9]+ { yylval.ival = atoi(yytext); return NUMBER; }
"+" { return '+'; }
"-" { return '-'; }
"*" { return '*'; }
"/" { return '/'; }
[ \t] ; /* skip whitespace */
\n { return '\n'; }
. { fprintf(stderr, "Invalid character: %s\n", yytext); return 0; }
%%

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

Output:

```
2 + 3
Postfix: 2 3 +
2 + 3 * 4
Postfix: 2 3 4 * +
10 - 4 / 2
Postfix: 10 4 2 / -
3 * 4 + 5
Postfix: 3 4 * 5 +
2 + a
Invalid character: a
```