SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Laboratory Manual

Name: Shubham Pathak

Enrollment No: 22001011

Batch: A2

TABLE OF CONTENT

		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.

Practical-1

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

Source Code:

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

int startsWithA(char *str) {
    return str[0] == 'a';
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Failed to open input.txt\n");
        return 1;
    }

    char str[100];
    while (fscanf(file, "%s", str) != EOF) {
        printf("Processing string: %s\n", str);

        for (int i = 0; str[i] != '\0'; i++) {
            if (str[i] != 'a' && str[i] != 'b') {
                printf("Invalid string: contains characters other
than a or b\n");
```

```
                fclose(file);
                return 1;
            }
        }

        if (startsWithA(str)) {
            printf("String starts with 'a': Accepted\n");
        } else {
            printf("String does not start with 'a': Rejected\n");
        }
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p1\a\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p1\a\"main
Processing string: aab
String starts with 'a': Accepted
Processing string: baa
String does not start with 'a': Rejected
Processing string: ab
String starts with 'a': Accepted
Processing string: bba
String does not start with 'a': Rejected
Processing string: abbb
String starts with 'a': Accepted
```

1.b: Write a program to recognize strings end with ‘a’.

Source Code:

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

int endsWithA(char *str) {
    int len = strlen(str);
    return len > 0 && str[len - 1] == 'a';
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Failed to open input.txt\n");
        return 1;
    }

    char str[100];
    while (fscanf(file, "%s", str) != EOF) {
        printf("Processing string: %s\n", str);

        for (int i = 0; str[i] != '\0'; i++) {
            if (str[i] != 'a' && str[i] != 'b') {
                printf("Invalid string: contains characters other
than a or b\n");
                fclose(file);
                return 1;
            }
        }

        if (endsWithA(str)) {
            printf("String ends with 'a': Accepted\n");
        } else {
            printf("String does not end with 'a': Rejected\n");
        }
```

```
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p1\b\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p1\b\"main
Processing string: aab
String does not end with 'a': Rejected
Processing string: baa
String ends with 'a': Accepted
Processing string: ab
String does not end with 'a': Rejected
Processing string: bba
String ends with 'a': Accepted
Processing string: abb
String does not end with 'a': Rejected
```

1.c: Write a program to recognize strings end with ‘ab’. Take the input from text file.

Source Code:

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

int endsWithAB(char *str) {
    int len = strlen(str);
    return len >= 2 && str[len - 2] == 'a' && str[len - 1] == 'b';
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Cannot open input.txt\n");
        return 1;
    }

    char str[100];
    while (fgets(str, 100, file)) {
        str[strcspn(str, "\n")] = '\0'; // Remove newline
        int valid = 1;
        for (int i = 0; str[i] != '\0'; i++) {
            if (str[i] != 'a' && str[i] != 'b') {
                valid = 0;
```

```
                break;
            }
        }
        if (!valid) {
            printf("Invalid string '%s': contains characters
other than a or b\n", str);
            continue;
        }
        if (endsWithAB(str))
            printf("String '%s' ends with 'ab': Accepted\n",
str);
        else
            printf("String '%s' does not end with 'ab':
Rejected\n", str);
    }
    fclose(file);
    return 0;
}
```

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p1\c\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p1\c\"main
String 'aba' does not end with 'ab': Rejected
String 'aab' ends with 'ab': Accepted
Invalid string 'xyz': contains characters other than a or b
```

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

Source Code:

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

int containsAB(char *str) {
    for (int i = 0; str[i + 1] != '\0'; i++) {
        if (str[i] == 'a' && str[i + 1] == 'b')
            return 1;
    }
    return 0;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Cannot open input.txt\n");
        return 1;
    }

    char str[100];
    while (fgets(str, 100, file)) {
        str[strcspn(str, "\n")] = '\0';
        int valid = 1;
        for (int i = 0; str[i] != '\0'; i++) {
            if (str[i] != 'a' && str[i] != 'b') {
                valid = 0;
                break;
            }
        }
        if (!valid) {
            printf("Invalid string '%s': contains characters
other than a or b\n", str);
            continue;
        }
```

```
        if (containsAB(str))
            printf("String '%s' contains 'ab': Accepted\n", str);
        else
            printf("String '%s' does not contain 'ab':
Rejected\n", str);
    }
    fclose(file);
    return 0;
}
```

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p1\d\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p1\d\"main
String 'aba' contains 'ab': Accepted
String 'abb' contains 'ab': Accepted
String 'ba' does not contain 'ab': Rejected
String 'aab' contains 'ab': Accepted
Invalid string 'xyz': contains characters other than a or b
```

Practical - 2

2.a: Write a program to recognize the valid identifiers.

Source Code:

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

const char *keywords[] = {"int", "float", "if", "else", "while",
"for", "return", NULL};

int isKeyword(char *str) {
    for (int i = 0; keywords[i]; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1;
    }
    return 0;
}

int isIdentifier(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() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Failed to open input.txt\n");
        return 1;
    }
```

```
    char str[100];
    while (fscanf(file, "%s", str) != EOF) {
        if (isKeyword(str))
            printf("'%s' is a keyword\n", str);
        else if (isIdentifier(str))
            printf("'%s' is a valid identifier\n", str);
        else
            printf("'%s' is neither a valid identifier nor a
keyword\n", str);
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p2\a\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile && "d:\6th sem\cd\lbm\p2\a\"tempCodeRunnerFile
'int' is a keyword
'myVar' is a valid identifier
'123var' is neither a valid identifier nor a keyword
'float' is a keyword
'while' is a keyword
'forLoop' is a valid identifier
'else' is a keyword
'my_function' is a valid identifier
'return' is a keyword
'_underscore' is a valid identifier
```

2.b: Write a program to recognize the valid operators.

Source Code:

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

const char *operators[] = {"+", "-", "*", "/", "=", "==",
"!=", "<", ">", "<=", ">=", "&&", "||", NULL};

int isOperator(char *str) {
    for (int i = 0; operators[i]; i++) {
        if (strcmp(str, operators[i]) == 0)
            return 1;
    }
    return 0;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    char str[10];
    while (fscanf(file, "%s", str) != EOF) {
        if (isOperator(str))
            printf("'%s' is a valid operator\n", str);
        else
            printf("'%s' is not a valid operator\n", str);
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p2\b\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p2\b\"main
'+' 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
```

2.c: Write a program to recognize the valid number.

Source Code:

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

int isNumber(char *str) {
    int hasDigit = 0, hasDot = 0;
    if (str[0] == '-' || str[0] == '+') str++;
    for (int i = 0; str[i]; i++) {
        if (isdigit(str[i])) hasDigit = 1;
        else if (str[i] == '.' && !hasDot) hasDot = 1;
        else return 0;
    }
    return hasDigit;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    char str[100];
```

```
    while (fscanf(file, "%s", str) != EOF) {
        if (isNumber(str))
            printf("'%s' is a valid number\n", str);
        else
            printf("'%s' is not a valid number\n", str);
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p2\c\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p2\c\"main
'123.45' is a valid number
'-123.45' is a valid number
'+123.45' is a valid number
'123' is a valid number
'-123' is a valid number
'+123' is a valid number
'abc' is not a valid number
'12.34abc' is not a valid number
'.123' is a valid number
```

2.d: Write a program to recognize the valid comments.

Source Code:

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

int isComment(char *str) {
    if (strncmp(str, "//", 2) == 0)
        return 1;
    if (strncmp(str, "/*", 2) == 0 && str[strlen(str) - 2] == '*'
&& str[strlen(str) - 1] == '/')
        return 1;
    return 0;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    char str[1000];
    while (fgets(str, 1000, file)) {
        str[strcspn(str, "\n")] = '\0';

        if (isComment(str))
            printf("'%s' is a valid comment\n", str);
        else
            printf("'%s' is not a valid comment\n", str);
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p2\d\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p2\d\"main
'// This is a single-line comment' is a valid comment
'/* This is a multi-line comment */' is a valid comment
'int x = 10;  // This is a comment after code' is not a valid comment
'/*' is not a valid comment
'This is a multi-line comment without closing' is not a valid comment
'*/' is not a valid comment
```

2.e: Write a program to implement Lexical Analyzer

Source Code:

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

const char *keywords[] = {"int", "float", "if", "else", "while",
"for", "return", NULL};
const char *operators[] = {"+", "-", "*", "/", "=", "==", "!=",
"<", ">", "<=", ">=", "&&", "||", NULL};

int isKeyword(char *str) {
    for (int i = 0; keywords[i]; i++)
        if (strcmp(str, keywords[i]) == 0) return 1;
    return 0;
}

int isOperator(char *str) {
    for (int i = 0; operators[i]; i++)
        if (strcmp(str, operators[i]) == 0) return 1;
    return 0;
}

int isIdentifier(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 isNumber(char *str) {
    int hasDigit = 0, hasDot = 0;
    if (str[0] == '-' || str[0] == '+') str++;
    for (int i = 0; str[i]; i++) {
        if (isdigit(str[i])) hasDigit = 1;
        else if (str[i] == '.' && !hasDot) hasDot = 1;
```

```
        else return 0;
    }
    return hasDigit;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Cannot open input.txt\n");
        return 1;
    }

    char buffer[100], ch;
    int i = 0;
    while ((ch = fgetc(file)) != EOF) {
        if (isalnum(ch) || ch == '_' || ch == '.' || ch == '-' ||
ch == '+') {
            buffer[i++] = ch;
        } else {
            buffer[i] = '\0';
            if (i > 0) {
                if (isKeyword(buffer)) printf("Keyword: %s\n",
buffer);
                else if (isIdentifier(buffer))
printf("Identifier: %s\n", buffer);
                else if (isNumber(buffer)) printf("Number: %s\n",
buffer);
            }
            if (strchr("+-*/=<>!&|", ch)) {
                buffer[0] = ch;
                buffer[1] = '\0';
                if (isOperator(buffer)) printf("Operator: %s\n",
buffer);
            } else if (strchr(";,(){}", ch)) {
                printf("Symbol: %c\n", ch);
```

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

Output:

input.txt

```
int main() {
    float x = 3.14;
    if (x > 0) {
        x = x + 1;
    }
    return 0;
}
```

```
[Running] cd "d:\6th sem\cd\lbm\p2\e\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p2\e\"main
Keyword: int
Identifier: main
Symbol: (
Symbol: )
Symbol: {
Keyword: float
Identifier: x
Operator: =
Number: 3.14
Symbol: ;
Keyword: if
Symbol: (
Identifier: x
Operator: >
Number: 0
Symbol: )
Symbol: {
Identifier: x
Operator: =
Identifier: x
Number: 1
Symbol: ;
Symbol: }
Keyword: return
Number: 0
Symbol: ;
Symbol: }
```

Practical - 3

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

Lexical Analyzer Generator (LEX) and Flex (Fast Lexical Analyzer)

1. LEX:

- LEX is a tool for generating lexical analyzers (scanners) that tokenize input based on regular expressions.
- Input: A .l file containing regular expressions and corresponding actions in C.
- Output: A C program (lex.yy.c) that performs lexical analysis.
- Workflow: Write .l file -> Run lex -> Compile lex.yy.c -> Execute.
- Features: Pattern matching, token generation, integration with parsers like Yacc.
- Limitations: Slower than Flex, less maintained.

2. Flex:

- Flex is a faster, modern alternative to LEX, compatible with LEX specifications.
- Improvements: Faster scanning, better performance, more features like start conditions.
- Usage: Similar to LEX, takes a .l file and generates a scanner.
- Additional Features:
  - Reentrant scanners for thread safety.
  - Support for larger input files and complex patterns.
  - Better error reporting and debugging options.
- Compilation: flex file.l -> gcc lex.yy.c -lfl -> Run.

3. Common Applications:

- Compilers: Tokenizing source code (keywords, identifiers, operators).
- Text Processing: Counting words, lines, or specific patterns.
- Data Parsing: Extracting structured data from text.

4. Example LEX/Flex File Structure:

- Definitions: Macros, regular expressions.
- Rules: Patterns and actions (e.g., return tokens or print).
- User Code: Additional C functions or main().

5. Integration:

- Both LEX and Flex integrate with Yacc/Bison for building parsers.
- Scanners pass tokens to parsers for syntax analysis.

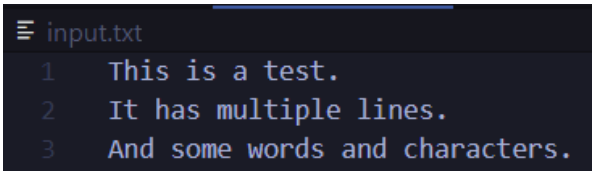
Practical - 4

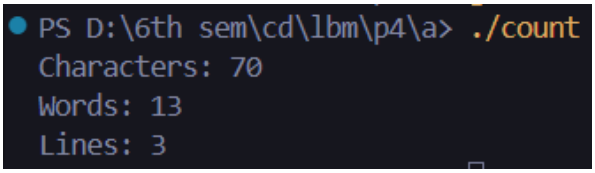
4.a: Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.

Source Code:

```
%{
#include <stdio.h>
int chars = 0, words = 0, lines = 0;
%}

%%

\n          { lines++; chars++; }
[ \t]+      { chars += yyleng; }
[a-zA-Z]+   { words++; chars += yyleng; }
.           { chars++; }

%%

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Cannot open input.txt\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    printf("Characters: %d\nWords: %d\nLines: %d\n", chars,
words, lines);
    return 0;
}

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

Output:


input.txt
1 This is a test.
2 It has multiple lines.
3 And some words and characters.

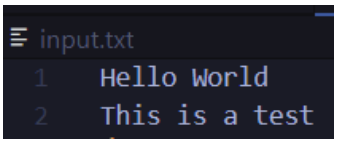

PS D:\6th sem\cd\lbm\p4\a> ./count
Characters: 70
Words: 13
Lines: 3

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

Source Code:

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

%%

[aeiouAEIOU]    { vowels++; }
[a-zA-Z]        { consonants++; }
.|\n            { /* Ignore */ }

%%

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Cannot open input.txt\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    printf("Vowels: %d\nConsonants: %d\n", vowels, consonants);
    return 0;
}

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

Output:




```
PS D:\6th sem\cd\lbm\p4\b> ./vowels_consonants
Vowels: 7
Consonants: 14
```

Aim3: Write a Lex program to print out all numbers from the given file.

Source Code:

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

%%

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

%%

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Cannot open input.txt\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    return 0;
}

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

Output:

input.txt

```
1   Price: 123.45
2   Quantity: 67
3   Invalid: abc
4   Total: 89.0
```

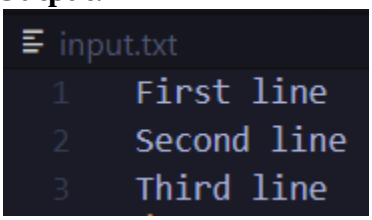
```
PS D:\6th sem\cd\lbm\p4\c> ./numbers
Number: 123.45
Number: 67
Number: 89.0
```

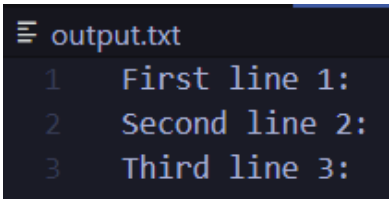
4.d: Write a Lex program which adds line numbers to the given file and display the same into different file.

Source Code:

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

%%

\n          { fprintf(out, "%d: %s", line++, yytext); }
.           { fprintf(out, "%s", yytext); }

%%

int main() {
    FILE *file = fopen("input.txt", "r");
    out = fopen("output.txt", "w");
    if (!file || !out) {
        printf("Cannot open files\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    fclose(out);
    return 0;
}

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

Output:


input.txt
1 First line
2 Second line
3 Third line


output.txt
1 First line 1:
2 Second line 2:
3 Third line 3:

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

Source Code:

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

%%

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

%%

int main() {
    FILE *file = fopen("input.html", "r");
    if (!file) {
        printf("Cannot open input.html\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    return 0;
}

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

Output:

input.html

```
<html>
<head>
    <title>Test</title>
    <!-- This is a comment -->
</head>
<body>
    <p>Hello</p>
</body>
</html>
```

```
PS D:\6th sem\cd\lbm\p4\e> ./html_tags_comments
Tag: <html>
Tag: <head>
Tag: <title>
Tag: </title>
Tag: <!-- This is a comment -->
Tag: </head>
Tag: <body>
Tag: <p>
Tag: </p>
Tag: </body>
Tag: </html>
```

Practical - 5

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.

Source Code:

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

%%

"//".*\n           { comments++; fprintf(out, "\n"); }
"/*"([^*]|\*+[^*/])*\*+"/" { comments++; /* Skip */ }
.|\n               { fprintf(out, "%s", yytext); }

%%

int main() {
    FILE *file = fopen("input.c", "r");
    out = fopen("output.c", "w");
    if (!file || !out) {
        printf("Cannot open files\n");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    fclose(out);
    printf("Number of comment lines: %d\n", comments);
    return 0;
}

int yywrap() {
    return 1;
```

```
}
```

Output:

input.c

```c
int main() {
    // Single line comment
    int x = 10;
    /* Multi-line
       comment */
    return x;
}
```

```
PS D:\6th sem\cd\lbm\p5\a> ./remove_comments
Number of comment lines: 2
```

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

Source Code:

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

%%

"int"|"float"|"if"|"else"|"while"|"for"|"return"   {
printf("Keyword: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*                             {
printf("Identifier: %s\n", yytext); }
[0-9]+(\.[0-9]+)?                                   {
printf("Number: %s\n", yytext); }
"+"|"-"|"*"|"/"|"="|"=="|"!="|"<"|">"|"<="|">="|"&&"|"||" {
printf("Operator: %s\n", yytext); }
"{"|"}"|"("|")"|";"|","                             {
printf("Symbol: %s\n", yytext); }
\"[^\"]*\"                                          {
printf("Literal: %s\n", yytext); }
[ \t\n]                                             { /* Ignore
*/ }
.                                                   {
printf("Unknown: %s\n", yytext); }

%%

int main() {
    FILE *file = fopen("input.c", "r");
    if (!file) {
        printf("Cannot open input.c\n");
        return 1;
    }
    yyin = file;
```

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

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

Output:

```c
int main() {
    float x = 3.14;
    if (x > 0) {
        printf("Hello");
    }
    return 0;
}
```

```
PS D:\6th sem\cd\lbm\p5\b> ./c_tokens
Keyword: int
Identifier: main
Symbol: (
Symbol: )
Symbol: {
Keyword: float
Identifier: x
Operator: =
Number: 3.14
Symbol: ;
Keyword: if
Symbol: (
Identifier: x
Operator: >
Number: 0
Symbol: )
Symbol: {
Identifier: printf
Symbol: (
Literal: "Hello"
Symbol: )
Symbol: ;
Symbol: }
Keyword: return
Number: 0
Symbol: ;
Symbol: }
```

Practical - 6

Aim: Program to implement Recursive Descent Parsing in C.

Source Code:

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

char *input;
int pos = 0;

void error();
void E();
void E_prime();
void T();
void T_prime();
void F();

void match(char c) {
    if (input[pos] == c) pos++;
    else error();
}

void E() {
    T();
    E_prime();
}

void E_prime() {
    if (input[pos] == '+') {
        match('+');
        T();
        E_prime();
    }
}
```

```
void T() {
    F();
    T_prime();
}

void T_prime() {
    if (input[pos] == '*') {
        match('*');
        F();
        T_prime();
    }
}

void F() {
    if (input[pos] == '(') {
        match('(');
        E();
        match(')');
    } else if (input[pos] == 'i') {
        match('i');
    } else {
        error();
    }
}

void error() {
    printf("Syntax error at position %d\n", pos);
    exit(1);
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error: Could not open input.txt\n");
        return 1;
```

```
    }

    char buffer[100];
    while (fgets(buffer, sizeof(buffer), file)) {
        buffer[strcspn(buffer, "\n")] = '\0';
        input = buffer;
        pos = 0;

        printf("Processing expression: %s\n", input);
        E();
        if (input[pos] == '\0')
            printf("Expression is valid\n\n");
        else
            printf("Invalid expression: extra characters\n\n");
    }

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

Output:

```
[Running] cd "d:\6th sem\cd\lbm\p6\" && gcc main.c -o main && "d:\6th sem\cd\lbm\p6\"main
Processing expression: i+i
Expression is valid

Processing expression: i*i+i
Expression is valid

Processing expression: i+(i*i)
Expression is valid

Processing expression: i+i*i
Expression is valid

Processing expression: i*(i+i)
Expression is valid
```

Practical - 7

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

Yet Another Compiler-Compiler (YACC)

1. **Overview:**

- YACC is a tool for generating parsers based on context-free grammars.
- Input: A .y file with grammar rules and actions in C.
- Output: A C program (y.tab.c) that parses input according to the grammar.
- Workflow: Write .y file -> Run yacc -> Compile y.tab.c -> Execute.
- Often used with Lex to tokenize input before parsing.

2. **Features:**

- Supports LALR(1) parsing.
- Generates parsers for programming languages, calculators, etc.
- Allows embedding C code for semantic actions.
- Handles ambiguous grammars with precedence rules.

3. **YACC File Structure:**

- Definitions: Token declarations, C includes.
- Rules: Grammar rules with actions.
- User Code: Main function, additional routines.

4. **Integration with Lex:**

- Lex generates a scanner (yylex) that returns tokens.
- YACC uses these tokens to build a parse tree.
- Tokens are defined in YACC (%token) and used in Lex.

5. **Applications:**

- Compilers: Syntax analysis of source code.
- Interpreters: Parsing command languages.
- Expression Evaluators: Arithmetic, logical expressions.

6. **Example Workflow:**

- Lex file defines tokens (e.g., NUMBER, PLUS).
- YACC file defines grammar (e.g., expr: expr PLUS expr).
- Compile: lex file.l; yacc -d file.y; gcc lex.yy.c y.tab.c -o parser.

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

Source Code:

expr.l:

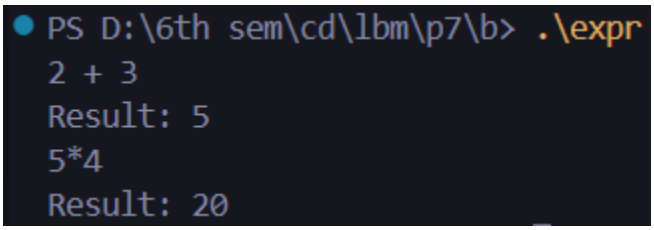
```
%{
#include "expr.tab.h"
%}

%%

[0-9]+      { yylval = atoi(yytext); return NUMBER; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MULT; }
"/"         { return DIV; }
[ \t]       { /* Ignore spaces and tabs */ }
\n          { return NEWLINE; }
.           { return yytext[0]; }

%%

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

expr.y:

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

%token NUMBER PLUS MINUS MULT DIV NEWLINE
```

```
%left PLUS MINUS
%left MULT DIV

%%

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

expr: NUMBER        { $$ = $1; }
    | expr PLUS expr { $$ = $1 + $3; }
    | expr MINUS expr { $$ = $1 - $3; }
    | expr MULT expr { $$ = $1 * $3; }
    | expr DIV expr  { if ($3 != 0) $$ = $1 / $3; else {
yyerror("Division by zero"); $$ = 0; } }
    ;

%%

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

int main() {
    yyparse();
    return 0;
}
```

Output:



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

Source Code:

calc.l :

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

%%

[0-9]+      { yylval = atoi(yytext); return NUMBER; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MULT; }
"/"         { return DIV; }
[ \t]       { /* Ignore spaces and tabs */ }
\n          { return NEWLINE; }
.           { return yytext[0]; }

%%

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

calc.y:

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

%token NUMBER PLUS MINUS MULT DIV NEWLINE
%left PLUS MINUS
```

```
%left MULT DIV

%%

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

expr: NUMBER        { $$ = $1; }
    | expr PLUS expr { $$ = $1 + $3; }
    | expr MINUS expr { $$ = $1 - $3; }
    | expr MULT expr { $$ = $1 * $3; }
    | expr DIV expr  { if ($3 != 0) $$ = $1 / $3; else {
yyerror("Division by zero"); $$ = 0; } }
    ;

%%

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

int main() {
    yyparse();
    return 0;
}
```

Output:

```
PS D:\6th sem\cd\lbm\p7\c> .\calc
2+3
Result: 5
1+3
Result: 4
0/0
Error: Division by zero
Result: 0
```

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

Source Code:

infix_postfix.l

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

%%

[0-9]+      { yylval = atoi(yytext); return NUMBER; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MULT; }
"/"         { return DIV; }
"("         { return LPAREN; }
")"         { return RPAREN; }
[ \t]       { /* Ignore spaces and tabs */ }
\n          { return NEWLINE; }
.           { return yytext[0]; }

%%

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

infix_postfix.y

```
 %{
#include <stdio.h>
#include <string.h>
int yylex();
void yyerror(char *s);
char result[1000] = "";
```

```
void append(char *s);
%}

%token NUMBER PLUS MINUS MULT DIV LPAREN RPAREN NEWLINE
%left PLUS MINUS
%left MULT DIV

%%

program: /* Empty */
       | program expr NEWLINE { printf("Postfix: %s\n", result);
result[0] = '\0'; }
       | program NEWLINE      { /* Ignore empty lines */ }
       ;

expr: NUMBER        { char buf[10]; sprintf(buf, "%d ", $1);
append(buf); }
    | expr PLUS expr { append("+ "); }
    | expr MINUS expr { append("- "); }
    | expr MULT expr { append("* "); }
    | expr DIV expr  { append("/ "); }
    | LPAREN expr RPAREN { /* No action */ }
    ;

%%

void append(char *s) {
    strcat(result, s);
}

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

int main() {
```

```
    yyparse();
    return 0;
}
```

Output: