LAB MANUAL

of

COMPILER DESIGN LABORATORY

(CSE606)

Bachelor of Technology (CSE)

By

Vaidik Patel (22000842)

Third Year, Semester 6

Course In-charge: Prof. Vaibhavi Patel

NAVRACHANA UNIVERSITY | Accredited with Grade 'A' by NAAC

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester

(2024-2025)

List of Experiments

Experiment - 1

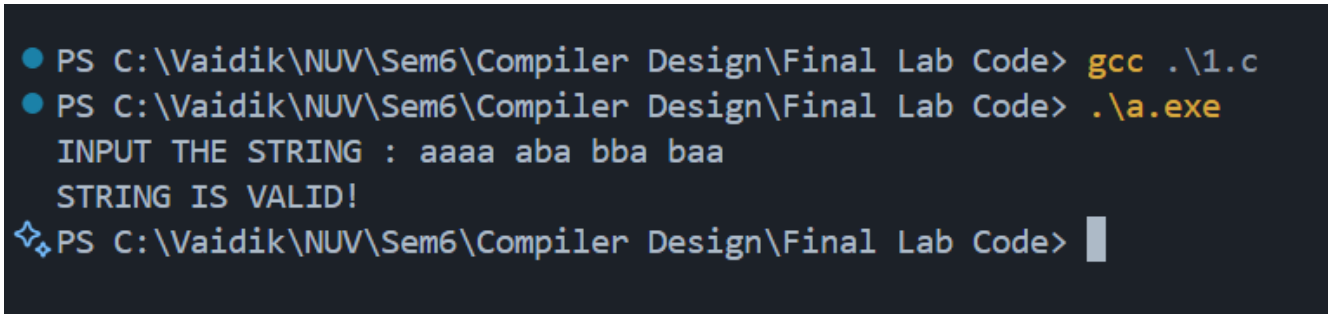
Aim : Write a C program for the following :

A. Write a program to recognise strings starts with ‘a’ over {a, b}.

Code :

```c
#include <stdio.h>
int main(){
  char input[10];
  int i=0, state=0;
  printf("INPUT THE STRING : ");
  scanf("%s", input);

  while(input[i]!='\0'){
    switch(state) {
    case 0:
    if(input[i]=='a') state = 1;
    else if (input[i] == 'b') state = 2;
    else state = 3;
    break;

    case 1:
    if (input[i] == 'a' || input[i] == 'b') state = 1;
    else state = 3;
    break;

    case 2:
    if (input[i] == 'a' || input[i] == 'b') state = 2;
    else state = 3;
    break;

    case 3:
    state = 3;
    break;
    }

    i++;
  }

  if (state==1) printf("STRING IS VALID!");
  else if (state==2 || state==0) {printf("STRING IS INVALID!");}
  else if (state==3) printf("STRING IS NOT RECOGNISED!");
  return 0;
}
```

Output :


PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> gcc .\1.c
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> .\a.exe
INPUT THE STRING : aaaa aba bba baa
STRING IS VALID!
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code>

B. Write a program to recognise strings end with ‘a’.

Code :

```c
#include <stdio.h>
int main(){
  char input[10];
  int i=0, state=0;
  printf("INPUT THE 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' || input[i] == 'b') state = 0;
    else state = 0;
    break;

    case 2:
    if (input[i] == 'a' || input[i] == 'b') state = 1;
    else state = 1;
    break;
    }

    i++;
  }

  if (state==1) printf("STRING IS VALID!");
  else if (state==2 || state==0) {printf("STRING IS INVALID!");}
  else if (state==3) printf("STRING IS NOT RECOGNISED!");
  return 0;
}
```

Output :

```
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> gcc .\2.c
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> .\a.exe
INPUT THE STRING : aba bbb bab aaaa
STRING IS VALID!
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code>
```

C. Write a program to recognise strings end with ‘ab’. Take the input from text file.

Code :

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

int ends_with_ab(const char *str) {
    int len = strlen(str);

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

    if (len >= 2 && str[len - 2] == 'a' && str[len - 1] == 'b') {
    return 1;
    }
    return 0;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
    perror("CANNOT OPEN INPUT FILE!");
    return 1;
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
    if (ends_with_ab(line)) {
    printf("STRING ENDS WITH 'ab' : %s", line);
    }
    }

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

Output :

input.txt

```
helloab
thisisatest
grab
abcab
about
tab
cab
labcoat
ab
```

```
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> gcc .\1c.c
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> .\a.exe
STRING ENDS WITH 'ab' : helloab
STRING ENDS WITH 'ab' : grab
STRING ENDS WITH 'ab' : abcab
STRING ENDS WITH 'ab' : tab
STRING ENDS WITH 'ab' : cab
STRING ENDS WITH 'ab' : ab
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code>
```

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

Code :

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

int main() {
    FILE *file;
    char input[100];
    int i, state;

    file = fopen("input.txt", "r");
    if (file == NULL) {
    printf("ERROR : FILE COULD NOT BE FOUND!\n");
    return 1;
    }

    while (fgets(input, sizeof(input), file)) {
    input[strcspn(input, "\n")] = 0;
    i = 0;
    state = 0;




    if (state == 2) printf("THE STRING \"%s\" CONTAINS 'ab'.
\n", input);
    else printf("THE STRING \"%s\" DOES NOT CONTAIN 'ab'.\n",  input);
    }

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

Output :

input.txt

```
helloab
thisisatest
grab
abcab
about
tab
cab
labcoat
ab
```

```
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> gcc .\1d.c
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code> .\a.exe
THE STRING "helloab" DOES NOT CONTAIN 'ab'.
THE STRING "thisisatest" DOES NOT CONTAIN 'ab'.
THE STRING "grab" DOES NOT CONTAIN 'ab'.
THE STRING "abcab" DOES NOT CONTAIN 'ab'.
THE STRING "about" DOES NOT CONTAIN 'ab'.
THE STRING "tab" DOES NOT CONTAIN 'ab'.
THE STRING "cab" DOES NOT CONTAIN 'ab'.
THE STRING "labcoat" DOES NOT CONTAIN 'ab'.
THE STRING "ab" DOES NOT CONTAIN 'ab'.
PS C:\Vaidik\NUV\Sem6\Compiler Design\Final Lab Code>
```

Experiment - 2

Aim : Write a Python program for the following :

A. Write a program to recognise the valid identifiers.

Code :

```
def is_valid_identifier(token):
    state = 0
    for char in token:
    if state == 0:
    if char.isalpha() or char == '_':
    state = 1
    else:
    return False
    elif state == 1:
    if char.isalnum() or char == '_':
    state = 1
    else:
    return False
    return state == 1

def is_keyword(token):
    keywords = {"if", "else", "while", "return", "for", "def",
"class", "import", "from", "as", "with", "try", "except",
"finally", "raise", "lambda", "pass", "break", "continue", "in",
"not", "or", "and", "is", "None", "True", "False", "global",  "nonlocal", "assert",
"yield"}
    return token in keywords

def tokenize_and_check(input_string):
    tokens = input_string.split()
    results = []
    for token in tokens:
    identifier = is_valid_identifier(token)
    keyword_check = is_keyword(token)
    status = "Both Identifier and Keyword" if identifier and keyword_check else \
    "Valid Identifier" if identifier else \
    "Keyword" if keyword_check else "Invalid"
    results.append((token, status))

    return results

if  name    == " main ":
    with open("input.txt", "r") as file:
    input_string = file.read().strip()

    results = tokenize_and_check(input_string)


    for token, status in results:
    file.write(f"Token: '{token}', Status: {status}\n")
```

```python
print("\nTokenized Output:")
for token, status in results:
print(f"Token: '{token}', Status: {status}")
```

Output :

input.txt

```
if
else
my_var
1variable
_underscore
class
def
function_name
99bottles
helloWorld
int
float
__init__
try
except
lambda
whileTrue
True
False
None
```

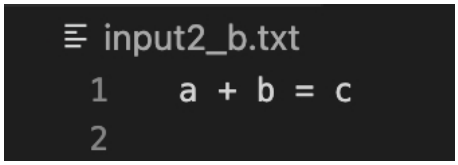
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

/opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P2-(a).py"
ompandit@Oms-MacBook-Pro COMPILER DESIGN % /opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P2-(a).py"

Tokenized Output:
Token: 'if', Status: Both Identifier and Keyword
Token: 'else', Status: Both Identifier and Keyword
Token: 'my_var', Status: Valid Identifier
Token: '1variable', Status: Invalid
Token: '_underscore', Status: Valid Identifier
Token: 'class', Status: Both Identifier and Keyword
Token: 'def', Status: Both Identifier and Keyword
Token: 'function_name', Status: Valid Identifier
Token: '99bottles', Status: Invalid
Token: 'helloWorld', Status: Valid Identifier
Token: 'int', Status: Valid Identifier
Token: 'float', Status: Valid Identifier
Token: '__init__', Status: Valid Identifier
Token: 'try', Status: Both Identifier and Keyword
Token: 'except', Status: Both Identifier and Keyword
Token: 'lambda', Status: Both Identifier and Keyword
Token: 'whileTrue', Status: Valid Identifier
Token: 'True', Status: Both Identifier and Keyword
Token: 'False', Status: Both Identifier and Keyword
Token: 'None', Status: Both Identifier and Keyword
ompandit@Oms-MacBook-Pro COMPILER DESIGN %
```

B. Write a program to recognise the valid operators.

Code :

```
with open("input2_b.txt", "r") as file:
    input_str = file.read().split()

operators = {"+", "-", "%", "*", "/", "<", ">", "--", "!=", "+="}
i = 0 tokens = []
while i < len(input_str):
    for char in input_str:
    if char in operators:
    print(f"{char} IS AN OPERATOR")
    tokens.append((char, "operator"))
    else:
    print(f"{char} IS AN IDENTIFIER")
    tokens.append((char, "identifier"))
    i += 1
```

Output :



C. Write a program to recognise the valid number.

Code :

```
def is_valid_number_fsm(number: str) -> bool: state = 'a'
for char in number: if state == 'a':
if char in '+-': state = 'h'
elif char.isdigit(): state = 'b'
else:
return False elif state == 'h':
if char.isdigit(): state = 'b'
else:
return False elif state == 'b':
if char.isdigit(): state = 'b'
elif char == '.': state = 'c'
elif char in 'Ee': state = 'e'
else:
return False elif state == 'c':
if char.isdigit(): state = 'd'
else:
return False elif state == 'd':
if char.isdigit(): state = 'd'
elif char in 'Ee': state = 'e'
else:
return False elif state == 'e':
if char in '+-': state = 'f'
elif char.isdigit(): state = 'g'
else:
return False elif state == 'f':
if char.isdigit(): state = 'g'
else:
return False

    elif state == 'g':
    if char.isdigit():
    state = 'g'
    else:
    return False
    return state in {'b', 'd', 'g'}

if  name    == " main ":
    try:
    with open("numbers.txt", "r") as file:
    for line in file:
    number = line.strip()
    print(f"'{number}' IS A VALID NUMBER :
{is_valid_number_fsm(number)}")
    except FileNotFoundError:
    print("ERROR : 'numbers.txt' FILE NOT FOUND.")
```

Output :

numbers.txt

```
1    123
2    -456.78
3    3.14159
4    1E10
5    -2.5e-3
6    +100
7    abc
8    12.34.56
9    E45
10   1.2.3
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

/opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P3-(c).py"
ompandit@Oms-MacBook-Pro COMPILER DESIGN % /opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P3-(c).py"
'123' IS A VALID NUMBER : True
'-456.78' IS A VALID NUMBER : True
'3.14159' IS A VALID NUMBER : True
'1E10' IS A VALID NUMBER : True
'-2.5e-3' IS A VALID NUMBER : True
'+100' IS A VALID NUMBER : True
'abc' IS A VALID NUMBER : False
'12.34.56' IS A VALID NUMBER : False
'E45' IS A VALID NUMBER : False
'1.2.3' IS A VALID NUMBER : False
'' IS A VALID NUMBER : False
'' IS A VALID NUMBER : False
ompandit@Oms-MacBook-Pro COMPILER DESIGN %
```

D. Write a program to recognise the valid comments.

Code :

```
def is_valid_comment(line: str) -> bool:
    state = 'start'
    i = 0
    while i < len(line):
    char = line[i]



    elif state == 'multi_line':
    if char == '*':
    state = 'multi_line_end'

    elif state == 'multi_line_end':
    if char == '/':


    state = 'multi_line'

    i += 1
    return state == 'multi_line_end'  if        name    == " main ":
    try:
    with open("comments.txt", "r") as file:
    for line in file:
    line = line.strip()
    print(f"'{line}' is a valid comment:
{is_valid_comment(line)}")
    except FileNotFoundError:
    print("Error: 'comments.txt' file not found.")
```

Output :

comments.txt

```
1   Hello World
2   // This is a single-line comment
3   /* This is a multi-line comment */
4   Not a comment
5   /* Unclosed comment
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
/opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P2-(d).py"
ompandit@Oms-MacBook-Pro COMPILER DESIGN % /opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P2-(d).py"
'Hello World' is a valid comment: False
'// This is a single-line comment' is a valid comment: True
'/* This is a multi-line comment */' is a valid comment: True
'Not a comment' is a valid comment: False
'/* Unclosed comment' is a valid comment: False
ompandit@Oms-MacBook-Pro COMPILER DESIGN %
```

E. Program to implement Lexical Analyser.

Code :

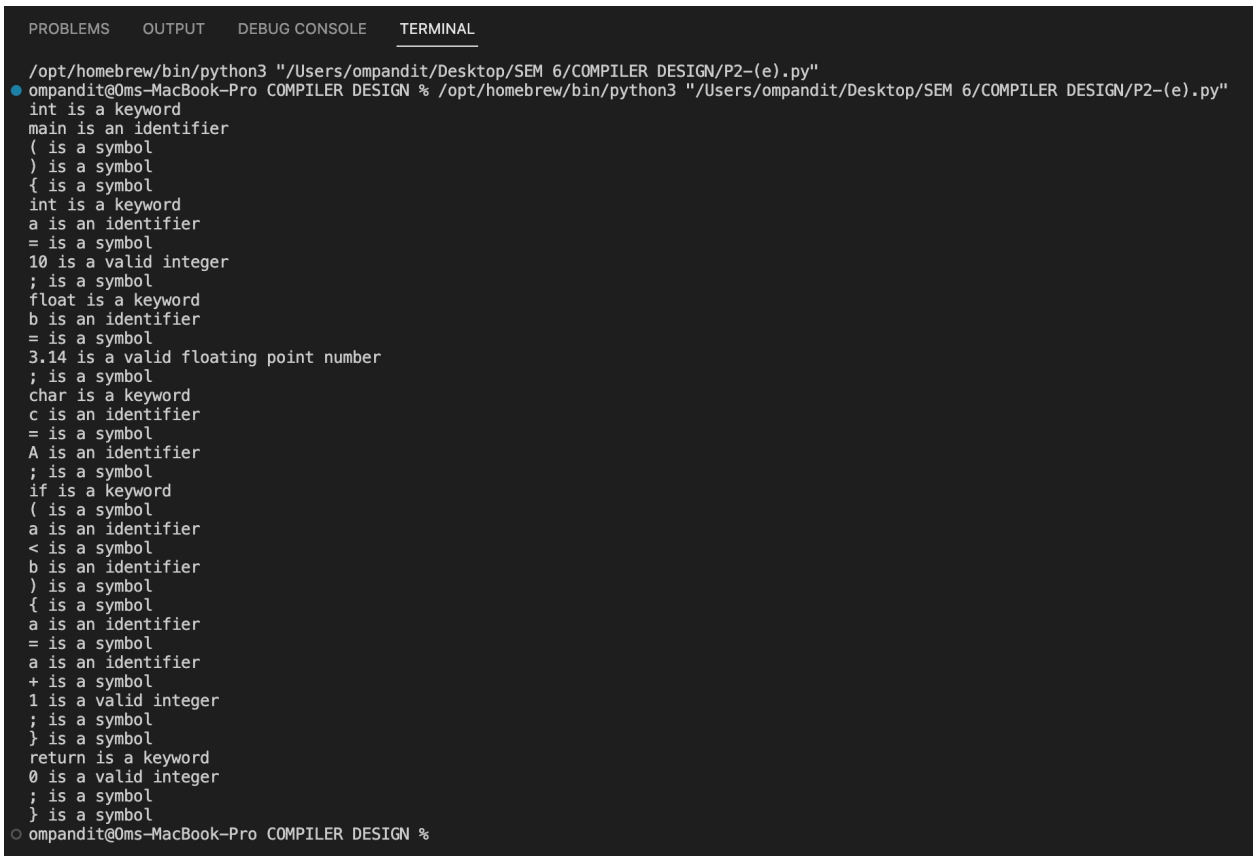
```
def check(lexeme):
    keywords = {"auto", "break", "case", "char", "const",  "continue", "default",
"do",
    "double", "else", "enum", "extern", "float",  "for", "goto", "if",
    "inline", "int", "long", "register", "restrict",  "return", "short", "signed",
    "sizeof", "static", "struct", "switch", "typedef",
"union", "unsigned", "void", "volatile", "while"}
    if lexeme in keywords:
    print(f"{lexeme} is a keyword")
    else:
    print(f"{lexeme} is an identifier")

def lexer(filename):
    try:
    with open(filename, "r") as f:

    print("Error opening file")
    return


lexeme = "" state = 0
f -= 1
elif state == 11: if c == '/':
while f < len(buffer) and buffer[f] != '\n': f += 1
state = 0 elif c == '*':
f += 1
while f < len(buffer) - 1 and not (buffer[f] == '*' and buffer[f + 1] == '/'):
f += 1
f += 2
state = 0 else:
print("/ is an operator") state = 0
f -= 1
elif state == 13: if c.isdigit():
lexeme += c elif c == '.':
state = 14 lexeme += c
elif c in "Ee": state = 16 lexeme += c
else:
print(f"{lexeme} is a valid integer") lexeme = ""
state = 0
f -= 1
elif state == 14: if c.isdigit():
lexeme += c state = 15
else:
print("Error: Invalid floating point format") lexeme = ""
state = 0 elif state == 15:
if c.isdigit(): lexeme += c elif c in "Ee":
state = 16 lexeme += c
```

```
else:
print(f"{lexeme} is a valid floating point

number")


lexeme = "" state = 0




lexer("input2.txt")
```

Output :

input2.txt

```
1   // This is a single-line comment
2   /* This is
3      a multi-line comment */
4   int main() {
5       int a = 10;
6       float b = 3.14;
7       char c = 'A';
8       if (a < b) {
9           a = a + 1;
10      }
11      return 0;
12  }
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
/opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P2-(e).py"
ompandit@Oms-MacBook-Pro COMPILER DESIGN % /opt/homebrew/bin/python3 "/Users/ompandit/Desktop/SEM 6/COMPILER DESIGN/P2-(e).py"
int is a keyword
main is an identifier
(is a symbol
) is a symbol
{ is a symbol
int is a keyword
a is an identifier
= is a symbol
10 is a valid integer
; is a symbol
float is a keyword
b is an identifier
= is a symbol
3.14 is a valid floating point number
; is a symbol
char is a keyword
c is an identifier
= is a symbol
A is an identifier
; is a symbol
if is a keyword
(is a symbol
a is an identifier
< is a symbol
b is an identifier
) is a symbol
{ is a symbol
a is an identifier
= is a symbol
a is an identifier
+ is a symbol
1 is a valid integer
; is a symbol
} is a symbol
return is a keyword
0 is a valid integer
; is a symbol
} is a symbol
ompandit@Oms-MacBook-Pro COMPILER DESIGN %

Experiment - 3

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

DESCRIPTION :

Lexical analysis is the first phase of a compiler, responsible for converting source code into tokens. This phase is automated using Lexical Analyzer Generators like LEX and Flex.

LEX (Lexical Analyzer Generator)

LEX is a tool used for generating lexical analyzers in compiler design. It helps in pattern recognition and tokenising input text using regular expressions. LEX works by defining patterns and corresponding actions in a .l file, which is then processed to generate a C-based scanner.

Key Features of LEX :

- Uses regular expressions to match patterns in input text.
- Generates lex.yy.c, a C program implementing the scanner.
- Can be compiled using a C compiler to produce an executable lexer.
- Works with YACC (Yet Another Compiler Compiler) to build full-fledged compilers.

Working of LEX :

- Specification: The user writes a .l file containing regular expressions and C actions.
- Processing: The lex command processes the .l file and generates lex.yy.c.
- Compilation: The lex.yy.c is compiled with gcc to create an executable scanner.
- Execution: The scanner reads input, matches patterns, and executes the corresponding actions.

Flex (Fast Lexical Analyzer)

Flex is an enhanced and faster version of LEX, designed for improved performance and portability. It follows the same working mechanism as LEX but generates more efficient and optimized C code.

Key Features of Flex :

- Faster and more efficient than LEX.
- Uses longest match rule over first match rule.
- Generates lex.yy.c, similar to LEX but optimized for better performance.
- Works seamlessly on Linux, Unix, and Windows with the required dependencies.

Working of Flex :

- Write a `` file with pattern definitions and C-based actions.
- Use the `` command to generate lex.yy.c.
- Compile the file using gcc.
- Run the executable, which scans the input and processes tokens.

Differences Between LEX and Flex

Feature	*LEX*	*Flex*
Speed	Slower	Faster
Portability	Limited	Widely used in Linux & Unix
Memory Usage	Higher	Optimized
Output File	lex.yy.c	lex.yy.c
Default Action	Returns first match	Returns longest match

Procedure

- Create a .l file (e.g., lexer.l) containing regular expressions and C code.
- Use the flex command to generate lex.yy.c.
- Compile the generated C file using GCC.
- Run the executable and provide input for analysis.

Example Code (LEX/Flex Program)

```
%{
#include <stdio.h>
%}
%%
[0-9]+        { printf("NUMBER\n"); }
[a-zA-Z]+   { printf("IDENTIFIER\n"); }
.               { printf("SPECIAL CHARACTER\n"); }
%%
int main() {
   yylex();
   return 0;
}
```

Conclusion

LEX and Flex are powerful tools for lexical analysis in compilers. They help automate tokenisation using regular expressions and C functions, making lexical analysis efficient.

Experiment - 4

Aim : Implement the following using LEX :

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

Code :

P4 > input.txt

```
1   Hello
2   Good Morning
3   This is my lex program
4   123456 677 34.676
5   56e56
```

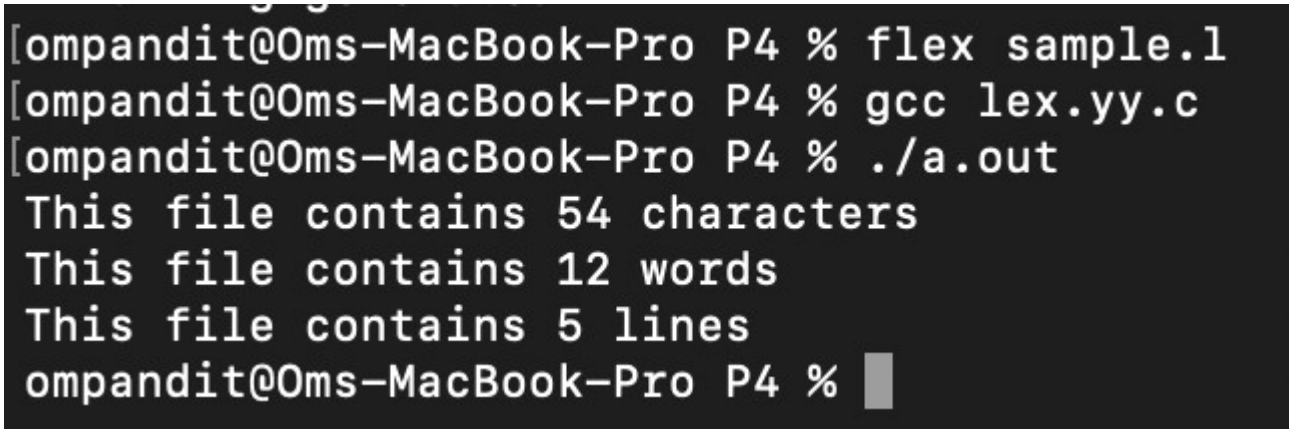
P4 > sample.l

```
1   %{
2   #include<stdio.h>
3   int char_count=0, word_count=0, line_count=0;
4   %}
5   %%
6   \n  {line_count++;word_count++;}
7   [\t ]+  word_count++;
8   . char_count++;
9   %%
10  void main(){
11  yyin=fopen("input.txt","r");
12  yylex();
13  printf("This file contains %d characters\n", char_count);
14  printf("This file contains %d words\n", word_count);
15  printf("This file contains %d lines\n", line_count);
16  }
17  int yywrap(){ return(1);}
```

Output :

P4

Name	Date Modified	Size	Kind
a.out	Today at 3:13 PM	53 KB	Document
input.txt	Today at 3:05 PM	66 bytes	Plain Text
lex.yy.c	Today at 3:13 PM	45 KB	text document
sample.l	Today at 3:10 PM	393 bytes	text document


[ompandit@Oms-MacBook-Pro P4 % flex sample.l
[ompandit@Oms-MacBook-Pro P4 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P4 % ./a.out
This file contains 54 characters
This file contains 12 words
This file contains 5 lines
ompandit@Oms-MacBook-Pro P4 %

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

Code :

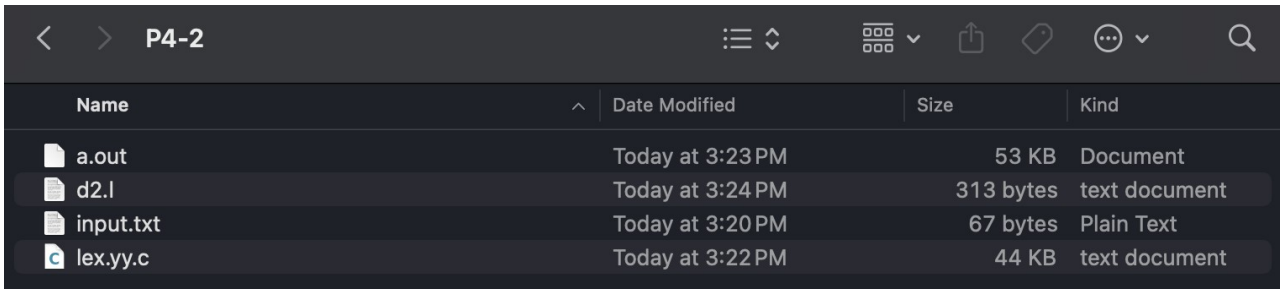
P4-2 > input.txt

```
Hello
Good Morning
This is my lex program
123456 677 34.676
56e56
```

P4-2 > d2.l

```
%{
#include<stdio.h>
int consonants=0, vowels=0;
%}
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z] {consonants++;}
. ;
%%
int main(){
yyin=fopen("input.txt","r");
yylex();
printf("This file contains .....");
printf("\n\t%d vowels ",vowels);
printf("\n\t%d consonants ",consonants);
return 0;
}
int yywrap(){ return(1);}
```

Output :




```
[ompandit@Oms-MacBook-Pro ~ % cd Desktop
[ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
[ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
[ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd P4-2
[ompandit@Oms-MacBook-Pro P4-2 % flex d2.l
[ompandit@Oms-MacBook-Pro P4-2 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P4-2 % ./a.out




 This file contains .....
         12 vowels
[        23 consonants %
 ompandit@Oms-MacBook-Pro P4-2 %
```

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

Code :

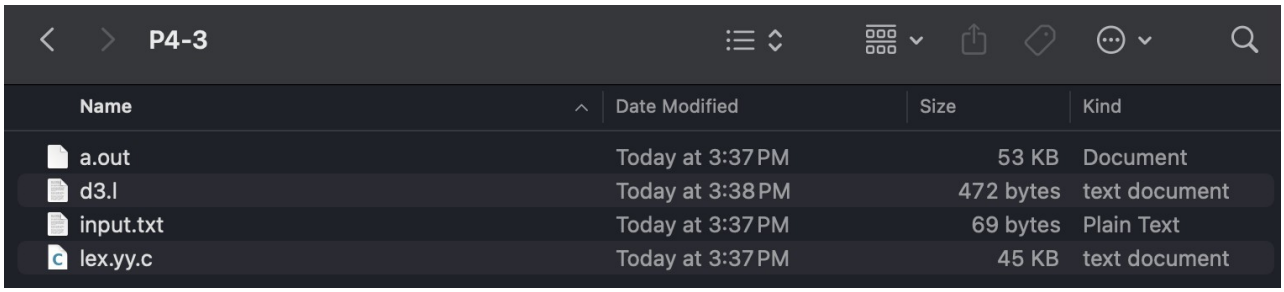
P4-3 > input.txt

```
1   Hello
2   Good Morning
3   This is my lex program
4   123456
5   677
6   34.676
7   56e56
```

P4-3 > d3.l

```
1   %{
2   #include <stdio.h>
3   #include <stdlib.h>
4   %}
5
6   digits     [0-9]+
7
8   %%
9   {digits}(\.{digits})?([eE][+-]?{digits})?     { printf("%s is valid number\n", yytext); }
10  \n                                            ;
11  .                                             ;
12  %%
13
14  int main() {
15      yyin = fopen("input.txt", "r");
16      if (!yyin) {
17          perror("Error opening input.txt");
18          return 1;
19      }
20      yylex();
21      fclose(yyin);
22      return 0;
23  }
24
25  int yywrap() {
26      return 1;
27  }
```

Output :



```
[ompandit@Oms-MacBook-Pro ~ % cd Desktop
[ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
[ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
[ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd P4-3
[ompandit@Oms-MacBook-Pro P4-3 % flex d3.l
[ompandit@Oms-MacBook-Pro P4-3 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P4-3 % ./a.out
 123456 is valid number
 677 is valid number
 34.676 is valid number
 56e56 is valid number
 ompandit@Oms-MacBook-Pro P4-3 %
```

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

Code :

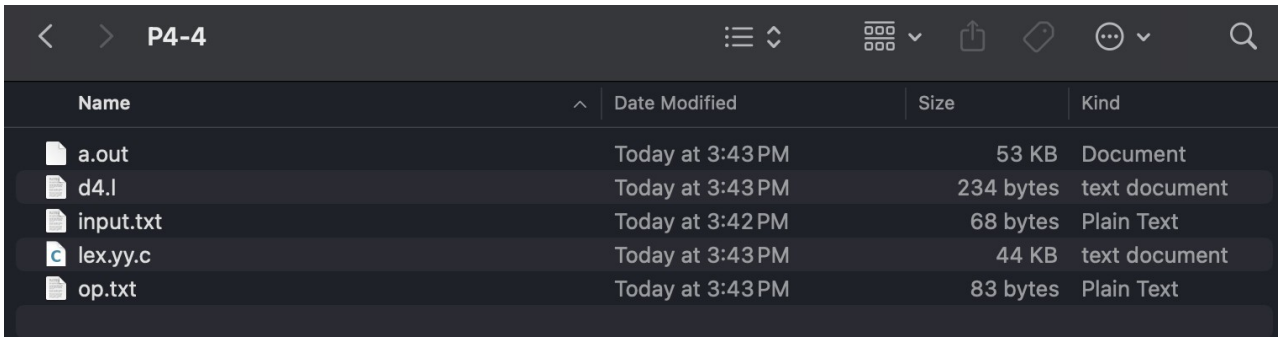
P4-4 > input.txt

```
1   Hello
2   Good Morning
3   This is my lex program
4   123456 677 34.676
5   56e56
```

P4-4 > d4.l

```
1   %{
2   int line_number = 1;
3   %}
4   %%
5   .* { fprintf(yyout, "%d: %s",line_number,yytext);line_number++; }
6
7   %%
8   int main(){
9   yyin=fopen("input.txt","r");
10  yyout=fopen("op.txt","w");
11  yylex();
12  printf("done");
13  return 0;
14  }
15  int yywrap(){ return(1);}
```

Output :



```
[ompandit@Oms-MacBook-Pro ~ % cd Desktop
[ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
[ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
[ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd P4-4
[ompandit@Oms-MacBook-Pro P4-4 % flex d4.l
[ompandit@Oms-MacBook-Pro P4-4 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P4-4 % ./a.out
 done%
 ompandit@Oms-MacBook-Pro P4-4 %
```

```
P4-4 > op.txt
   1    1: Hello
   2    2: Good Morning
   3    3: This is my lex program
   4    4: 123456 677 34.676
   5    5: 56e56
```

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

Code :

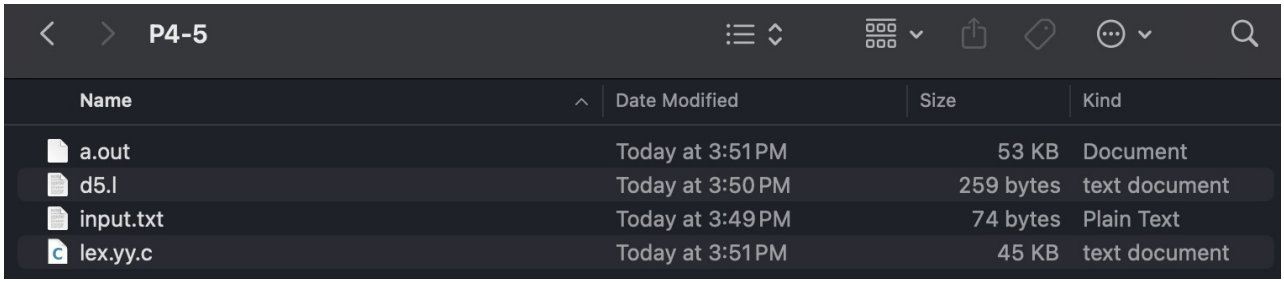
P4-5 > input.txt

```
<html>
<head> Heer </head>
<body>
<!-- iehhfjs 122 -->
</body>
</html>

```

P4-5 > d5.l

```
%{
#include<stdio.h>
int num = 0;
%}
%%
\<[a-zA-z0-9]+">" printf("%s is valid markup tag \n",yytext);
"<!--"(.|\n)*"-->" num++;
\n ;
. ;
%%
int main(){
yyin=fopen("input.txt","r");
yylex();
printf("%d comment", num);
return 0;
}
int yywrap(){ return(1);}
```

Output :



```
[ompandit@Oms-MacBook-Pro ~ % cd Desktop
[ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
[ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
[ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd P4-5
[ompandit@Oms-MacBook-Pro P4-5 % flex d5.l
[ompandit@Oms-MacBook-Pro P4-5 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P4-5 % ./a.out
 <html> is valid markup tag
 <head> is valid markup tag
 <body> is valid markup tag
 ompandit@Oms-MacBook-Pro P4-5 %
```

Experiment - 5

Aim : Perform the following :

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 :

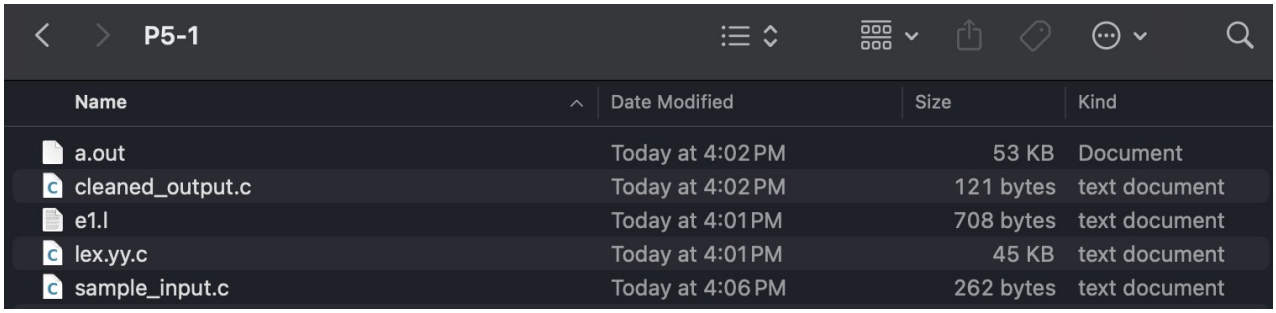
P5-1 > C sample_input.c > ...

```c
#include <stdio.h>

// This is a single-line comment
int main() {
    int a = 5; /* Inline multi-line
                  comment */
    int b = 10; // Another comment
    printf("Sum = %d\n", a + b);
    /* Entire
       block
       comment */
    return 0;
}
```

P5-1 > e1.l

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

%%
"//".*                          { comment_count++; /* skip single-line comment */ }
"/*"([^*]|\*+[^*/])*\*+"/"      { comment_count++; /* skip multi-line comment */ }
.|\n                           { fputc(yytext[0], out); }
%%

int main() {
    FILE *in = fopen("sample_input.c", "r");
    if (!in) {
        perror("Input file error");
        return 1;
    }
    out = fopen("cleaned_output.c", "w");
    if (!out) {
        perror("Output file error");
        return 1;
    }
    yyin = in;
    yylex();
    fclose(in);
    fclose(out);
    printf("Total comments removed: %d\n", comment_count);
    return 0;
}

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

Output :



```
[ompandit@Oms-MacBook-Pro ~ % cd Desktop
[ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
[ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
[ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd P5-1
[ompandit@Oms-MacBook-Pro P5-1 % flex e1.l
[ompandit@Oms-MacBook-Pro P5-1 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P5-1 % ./a.out
[ompandit@Oms-MacBook-Pro P5-1 % flex e1.l
[ompandit@Oms-MacBook-Pro P5-1 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P5-1 % ./a.out
[ompandit@Oms-MacBook-Pro P5-1 % flex e1.l
[ompandit@Oms-MacBook-Pro P5-1 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P5-1 % ./a.out
Total comments removed: 4
ompandit@Oms-MacBook-Pro P5-1 %
```

P5-1 > C cleaned_output.c > ...

```c
#include <stdio.h>


int main() {
    int a = 5;
    int b = 10;
    printf("Sum = %d\n", a + b);

    return 0;
}
```

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

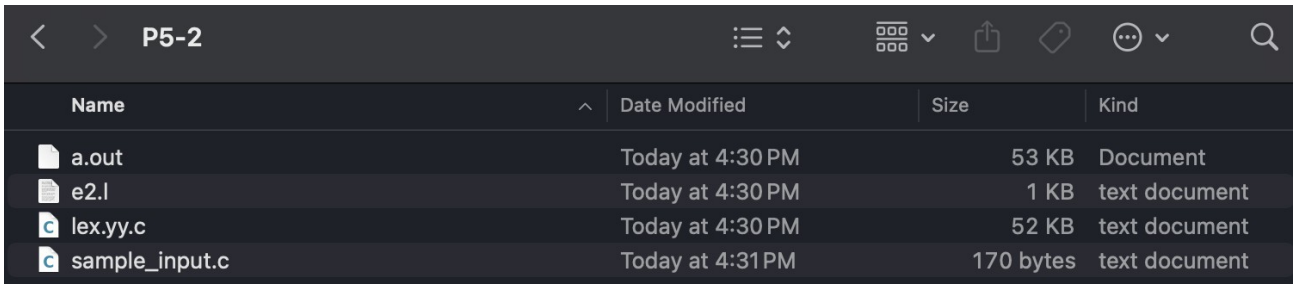
Code :

P5-2 > C sample_input.c > ...

```c
#include <stdio.h>

int main() {
    int a = 10;
    float b = 3.14;
    char c = 'x';
    printf("Hello, World!");
    if (a > b) {
        a++;
    }
    return 0;
}
```

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

digit       [0-9]
alpha       [a-zA-Z]
id          {alpha}({alpha}|{digit})*
int_const   {digit}+
float_const {digit}+"."{digit}+
string_lit  \"([^\\\"]|\\.)*\"
char_lit    \'([^\\\']|\\.)\'

%%
"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"|"volatil
e"|"while"   { printf("Keyword: %s\n", yytext); }

{id}                { printf("Identifier: %s\n", yytext); }
"==" | "!=" | "<=" | ">=" | "&&" | "||" | "++" | "--" | "+" | "-" | "*" | "/" | "=" | "<" | ">"
{ printf("Operator: %s\n", yytext); }
{float_const}       { printf("Float Number: %s\n", yytext); }
{int_const}         { printf("Integer Number: %s\n", yytext); }
{string_lit}        { printf("String Literal: %s\n", yytext); }
{char_lit}          { printf("Character Literal: %s\n", yytext); }
[{}();,]            { printf("Special Symbol: %s\n", yytext); }
[ \t\n]+            { /* skip whitespace */ }
.                   { printf("Unrecognized token: %s\n", yytext); }
%%

int main() {
   yyin = fopen("sample_input.c", "r");
   if (!yyin) {
      perror("Could not open file");
      return 1;
   }
   yylex();
   fclose(yyin);
   return 0;
}

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

Output :




```
[ompandit@Oms-MacBook-Pro ~ % cd Desktop
[ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
[ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
[ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd P5-2
[ompandit@Oms-MacBook-Pro P5-2 % flex e2.l
[ompandit@Oms-MacBook-Pro P5-2 % gcc lex.yy.c
[ompandit@Oms-MacBook-Pro P5-2 % ./a.out
Unrecognized token: #
Identifier: include
Operator: <
Identifier: stdio
Unrecognized token: .
Identifier: h
Operator: >
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Integer Number: 10
Special Symbol: ;
Keyword: float
Identifier: b
Operator: =
Float Number: 3.14
Special Symbol: ;
Keyword: char
Identifier: c
Operator: =
Character Literal: 'x'
Special Symbol: ;
Identifier: printf
Special Symbol: (
String Literal: "Hello, World!"
Special Symbol: )
Special Symbol: ;
Keyword: if
Special Symbol: (
Identifier: a
Operator: >
Identifier: b
Special Symbol: )
Special Symbol: {
Identifier: a
Operator: ++
Special Symbol: ;
Special Symbol: }
Keyword: return
Integer Number: 0
Special Symbol: ;
Special Symbol: }
ompandit@Oms-MacBook-Pro P5-2 %
```

Experiment - 6

Aim : Program to implement Recursive Descent Parsing in C.

Code :

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

char ip[100];
int l = 0;

void match(char t);
void E();
void T();
void F();
void T_dash();
void E_dash();

void match(char t) {
    if (ip[l] == t) {
        l++;
    } else {
        printf("Error\n", ip[l]);
        exit(1);
    }
}

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

void T() {
    F();
    T_dash();
}

void F() {
    if (ip[l] == '(') {
        match('(');
        E();
        match(')');
    } else if (ip[l] == 'i') {
        match('i');
    } else if (ip[l] == 'n') {
        match('n');
    } else {
        printf("Error\n", ip[l]);
        exit(1);
    }
}
```

```c
void E_dash() {
    if (ip[l] == '+') {
        match('+');
        T();
        E_dash();
    } else if (ip[l] == '-') {
        match('-');
        T();
        E_dash();
    }
}

void T_dash() {
    if (ip[l] == '*') {
        match('*');
        F();
        T_dash();
    } else if (ip[l] == '/') {
        match('/');
        F();
        T_dash();
    }
}

int main() {
    printf("Enter the input ending with '$':\n");
    scanf("%s", ip);

    if (ip[strlen(ip) - 1] != '$') {
        printf("Error: Input must end with '$'\n");
        return 1;
    }

    E();

    if (ip[l] == '$') {
        printf("Parsing successful\n");
    } else {
        printf("Error: Parsing incomplete at '%c'\n", ip[l]);
    }

    return 0;
}
```

Output :

```
C:\Vaidik\NUV\Sem6\Compiler Design>a.exe
Enter the input ending with '$':
i+i$
Parsing successful

C:\Vaidik\NUV\Sem6\Compiler Design>a.exe
Enter the input ending with '$':
i+i i$
Error: Input must end with '$'

C:\Vaidik\NUV\Sem6\Compiler Design>
```

Experiment - 7

Aim : Perform the following :

A. Create Yacc and Lex specification files to recognises arithmetic expressions involving +, -, * and / .

Code :

lex.l

```
%{
#include "yacc.tab.h"
#include <stdlib.h>
#include <string.h>
%}

%%
[0-9]+                  { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { return IDENTIFIER; }
[+\-*/]                 { return yytext[0]; }
\n                      { return '\n'; }
[ \t]                   { /* skip whitespace */ }
.                       { return 0; }  // Invalid character triggers error in yacc
%%

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

yacc.y

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

int yylex(void);
int yyerror(char *s);
%}

%token NUMBER IDENTIFIER

%left '+' '-'
%left '*' '/'

%%
stmt: expr '\n' { printf("Valid\n"); }
    ;

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

int main() {
    return yyparse();
}

int yyerror(char *s) {
    printf("Invalid\n");
    return 0;
}
```

Output :

```
ompandit@Oms-MacBook-Pro ~ % cd Desktop
ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd YACC
ompandit@Oms-MacBook-Pro YACC % cd 1
ompandit@Oms-MacBook-Pro 1 % flex lex.l
ompandit@Oms-MacBook-Pro 1 % bison -d yacc.y
ompandit@Oms-MacBook-Pro 1 % gcc lex.yy.c yacc.tab.c
ompandit@Oms-MacBook-Pro 1 % ./a.out
a+b
Valid
```

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

Code :

2 > lex.l

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

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[+\-*/\n]   { return yytext[0]; }
[ \t]       { /* ignore whitespace */ }
%%

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

2 > yacc.y

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

int yylex(void);
int yyerror(char *s);
%}

%token NUMBER

// Precedence and associativity rules
%left '+' '-'
%left '*' '/'

%%
stmt: expr '\n' { printf("Result = %d\n", $1); }
    ;

expr: expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr {
        if ($3 == 0) {
            printf("Error: Divide by zero\n");
            exit(1);
        }
        $$ = $1 / $3;
      }
    | NUMBER
    ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    printf("Error: %s\n", s);
    return 0;
}
```

Output :

```
ompandit@Oms-MacBook-Pro ~ % cd Desktop
ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd YACC
ompandit@Oms-MacBook-Pro YACC % cd 2
ompandit@Oms-MacBook-Pro 2 % flex lex.l
ompandit@Oms-MacBook-Pro 2 % bison -d yacc.y
ompandit@Oms-MacBook-Pro 2 % gcc lex.yy.c yacc.tab.c
ompandit@Oms-MacBook-Pro 2 % ./a.out
3+4*2-1
Result = 10
```

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

Code :

3 > lex.l

```
%{
#include "yacc.tab.h"
#include <stdlib.h>
#include <string.h>
%}

%%
[0-9]+                  { yylval.num = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.id = strdup(yytext); return IDENTIFIER; }
[+\-*/()\n]             { return yytext[0]; }
[ \t]+                  { /* ignore whitespace */ }
.                       { printf("Invalid character: %s\n", yytext); return -1; }
%%

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

3 > yacc.y

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

int yylex(void);
int yyerror(char *s);
%}

%union {
    int num;
    char* id;
}

%token <num> NUMBER
%token <id> IDENTIFIER

%%
stmt: expr '\n' { printf("\n"); }
    ;

expr: expr '+' term   { printf("+ "); }
    | expr '-' term   { printf("- "); }
    | term
    ;

term: term '*' factor { printf("* "); }
    | term '/' factor { printf("/ "); }
    | factor
    ;

factor: '(' expr ')'
      | NUMBER        { printf("%d ", $1); }
      | IDENTIFIER    { printf("%s ", $1); free($1); }
      ;
%%

int main() {
    return yyparse();
}

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

Output :

```
ompandit@Oms-MacBook-Pro ~ % cd Desktop
ompandit@Oms-MacBook-Pro Desktop % cd SEM\ 6
ompandit@Oms-MacBook-Pro SEM 6 % cd COMPILER\ DESIGN
ompandit@Oms-MacBook-Pro COMPILER DESIGN % cd YACC
ompandit@Oms-MacBook-Pro YACC % cd 3
ompandit@Oms-MacBook-Pro 3 % flex lex.l
ompandit@Oms-MacBook-Pro 3 % bison -d yacc.y
ompandit@Oms-MacBook-Pro 3 % gcc lex.yy.c yacc.tab.c
ompandit@Oms-MacBook-Pro 3 % ./a.out
a+5-b
a 5 + b -
```