Lab Manual
Of
Compiler Design Laboratory
(CSE606)

Bachelor of Technology (CSE)

By
Het Patel (22000934)
Second Year, Semester 6
Course In-charge: Prof. Vaibhavi Patel

NAVRACHANA UNIVERSITY
a UGC recognized University

Department of Computer Science and Engineering
School Engineering and Technology
Navrachana University, Vadodara
Spring Semester
2024 – 2025

1

a) **Write a program to recognize strings starts with ‘a’ over {a, b}.**
Code:

```
class StringAutomaton:
    """Automaton to recognize strings starting with 'a' over the alphabet {a, b}."""
    def __init__(self):
        # States: 0 (Start), 1 (Valid state after 'a'), 2 (Dead state if first char is 'b')
        self.states = {
            0: {"a": 1, "b": 2},  # Transition from start state
            1: {"a": 1, "b": 1},  # Remains in valid state if 'a' or 'b' appears
            2: {"a": 2, "b": 2}   # Dead state if starts with 'b'
        }
        self.final_states = {1}  # Accepting state

    def accepts(self, string):
        state = 0
        for char in string:
            if char in self.states[state]:
                state = self.states[state][char]
            else:
                return False
        return state in self.final_states

# Taking user input
user_string = input("Enter a string consisting of 'a' and 'b': ").strip()
automaton = StringAutomaton()
print(f"'{user_string}' starts with 'a': {automaton.accepts(user_string)}")
```

Output:

```
PS C:\Users\USER\Desktop\SEM 6>  & 'c:\Users\USER\AppData
debugpy-2025.0.0-win32-x64\bundled\libs\debugpy\launcher'
Enter a string consisting of 'a' and 'b': abbabb
'abbabb' starts with 'a': True
```

```
PS C:\Users\USER\Desktop\SEM 6> & C:/Users/USER/AppData/
Enter a string consisting of 'a' and 'b': baabbaa
'baabbaa' starts with 'a': False
```

b) **Write a program to recognize strings end with ‘a’.**
Code:

```python
class StringAutomaton:
    """Automaton to recognize strings ending with 'a' over the alphabet {a, b}."""
    def __init__(self):
        # States: 0 (Start), 1 (Accepting if last char is 'a'), 2 (Non-accepting if last char is 'b')
        self.states = {
            0: {"a": 1, "b": 2},  # Transition from start state
            1: {"a": 1, "b": 2},  # Moves to state 2 if 'b' is encountered
            2: {"a": 1, "b": 2}   # Moves to state 1 if 'a' is encountered
        }
        self.final_states = {1}  # Accepting state if the last character is 'a'

    def accepts(self, string):
        state = 0
        for char in string:
            if char in self.states[state]:
                state = self.states[state][char]
            else:
                return False
        return state in self.final_states

# Taking user input
user_string = input("Enter a string consisting of 'a' and 'b': ").strip()
automaton = StringAutomaton()
print(f"'{user_string}' ends with 'a': {automaton.accepts(user_string)}")
```

Output:

```
PS C:\Users\USER\Desktop\SEM 6> & C:/Users/USER/AppData/
Enter a string consisting of 'a' and 'b': abba
'abba' ends with 'a': True
```

```
PS C:\Users\USER\Desktop\SEM 6> & C:/Users/USER/AppData/
Enter a string consisting of 'a' and 'b': abbaabb
'abbaabb' ends with 'a': False
```

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

Code:

```python
class StringAutomaton:
    """Automaton to recognize strings ending with 'ab' over the alphabet {a, b}."""
```

```
    def __init__(self):
        # States: 0 (Start), 1 (Seen 'a'), 2 (Accepting state if 'ab' is found)
        self.states = {
            0: {"a": 1, "b": 0},  # Start state: move to 1 if 'a', stay in 0 if 'b'
            1: {"a": 1, "b": 2},  # If 'a' again, stay in 1; if 'b', move to accepting state 2
            2: {"a": 1, "b": 0}   # If 'a' appears after 'ab', move to 1 again
        }
        self.final_states = {2}  # Accepting state when last two characters are 'ab'

    def accepts(self, string):
        state = 0
        for char in string:
            if char in self.states[state]:
                state = self.states[state][char]
            else:
                return False
        return state in self.final_states

# Write sample input to a text file
input_filename = "input_strings.txt"
sample_strings = ["aab", "abb", "bba", "aaa", "bab", "a", "b", "ba", "ab", "aab"]
with open(input_filename, "w") as file:
    for s in sample_strings:
        file.write(s + "\n")

# Read input from the text file
try:
    with open(input_filename, "r") as file:
        strings = [line.strip() for line in file if line.strip()]

    automaton = StringAutomaton()
    for s in strings:
        print(f"'{s}' ends with 'ab': {automaton.accepts(s)}")
except FileNotFoundError:
    print(f"Error: The file '{input_filename}' was not found.")
```

Input.txt:

File Edit View

```
aab
abb
bba
aaa
bab
a
b
ba
ab
aab
```

Output:

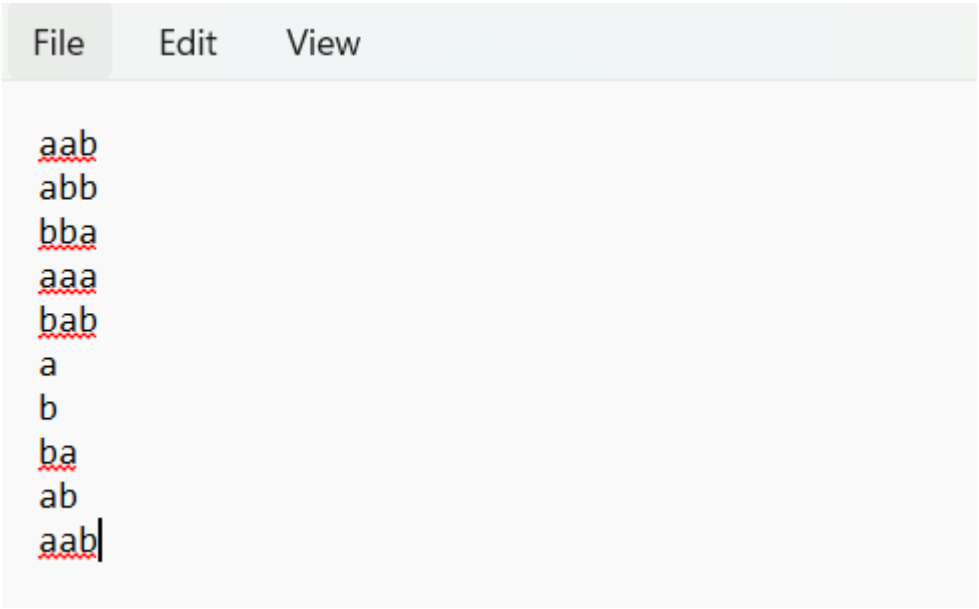
```
PS C:\Users\USER\Desktop\SEM 6\CD Lab>  & 'c:\Users\USER\AppData\Local\Progra
25.8.0-win32-x64\bundled\libs\debugpy\launcher' '56169' '--' 'c:\Users\USER\D
'aab' ends with 'ab': True
'abb' ends with 'ab': False
'bba' ends with 'ab': False
'aaa' ends with 'ab': False
'bab' ends with 'ab': True
'a' ends with 'ab': False
'b' ends with 'ab': False
'ba' ends with 'ab': False
'ab' ends with 'ab': True
'aab' ends with 'ab': True
PS C:\Users\USER\Desktop\SEM 6\CD Lab>
```

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

CODE:

```
def contains_ab(string):
    return 'ab' in string

def check_strings_from_file(filename):
    try:
        with open(filename, 'r') as file:
            lines = file.readlines()
            for line in lines:
                line = line.strip()  # Remove whitespace/newline characters
                if contains_ab(line):
                    print(f"'{line}' contains 'ab'")
                else:
                    print(f"'{line}' does NOT contain 'ab'")
    except FileNotFoundError:
        print(f"File '{filename}' not found.")

# Change the filename if needed
check_strings_from_file('input_strings.txt')
```

Input.txt:



OUTPUT:

```
PS C:\Users\USER\Desktop\SEM 6\CD Lab>  & 'c:\Users\USER\AppData\Local\P
25.8.0-win32-x64\bundled\libs\debugpy\launcher' '56464' '--' 'c:\Users\U
'aab' contains 'ab'
'abb' contains 'ab'
'bba' does NOT contain 'ab'
'aaa' does NOT contain 'ab'
'bab' contains 'ab'
'a' does NOT contain 'ab'
'b' does NOT contain 'ab'
'ba' does NOT contain 'ab'
'ab' contains 'ab'
'aab' contains 'ab'
PS C:\Users\USER\Desktop\SEM 6\CD Lab>
```

2

a) Write a program to recognize the valid identifiers.

Code:

```
def is_valid_identifier(token):
    state = 0  # Initial state
    for char in token:
        if state == 0:  # Start state
            if char.isalpha() or char == '_':
                state = 1  # Move to valid identifier state
            else:
                return False
        elif state == 1:  # Identifier continuation state
            if char.isalnum() or char == '_':
                state = 1  # Remain in valid state
            else:
                return False
    return state == 1  # Accepting state

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__":
    # Read input from file
    with open("input.txt", "r") as file:
        input_string = file.read().strip()

    results = tokenize_and_check(input_string)

    # Write output to file
    with open("output.txt", "w") as file:
        file.write("Tokenized Output:\n")
        for token, status in results:
            file.write(f"Token: '{token}', Status: {status}\n")

    # Print output to console
    print("\nTokenized Output:")
    for token, status in results:
        print(f"Token: '{token}', Status: {status}")
```

Output:

```
Token: 'here.', Status: Invalid
Token: '/*', Status: Invalid
Token: 'This', Status: Valid Identifier
Token: 'is', Status: Both Identifier and Keyword
Token: 'a', Status: Valid Identifier
Token: 'multi-line', Status: Invalid
Token: 'comment', Status: Valid Identifier
Token: 'spanning', Status: Valid Identifier
Token: 'multiple', Status: Valid Identifier
Token: 'lines', Status: Valid Identifier
Token: '*/', Status: Invalid
Token: 'Final', Status: Valid Identifier
Token: 'line', Status: Valid Identifier
Token: 'of', Status: Valid Identifier
Token: 'code.', Status: Invalid

[Done] exited with code=0 in 0.209 seconds
```

b) Write a program to recognize the valid operators.

Code:

```
class OperatorAutomaton:
    def __init__(self):
        # Define the valid operators (all operators used in Python)
        self.valid_operators = {
            '+', '-', '*', '/', '%', '**', '//',  # Arithmetic operators
            '==', '!=', '>', '<', '>=', '<=',  # Comparison operators
            'and', 'or', 'not',  # Logical operators
            '&', '|', '^', '~', '<<', '>>',  # Bitwise operators
            '=', '+=', '-=', '*=', '/=', '%=', '**=', '//=',  # Assignment operators
            'is', 'is not', 'in', 'not in',  # Identity and membership operators
            ',', '.', ':', '(', ')', '[', ']', '{', '}'  # Punctuation used in expressions
        }

    def is_identifier(self, part):
        """Check if a part is a valid identifier (variable name)."""
        if not part:
            return False
        # First character must be a letter or underscore
        if not (part[0].isalpha() or part[0] == '_'):
            return False
        # The rest must be letters, numbers, or underscores
        for char in part[1:]:
            if not (char.isalnum() or char == '_'):
                return False
        return True

    def process(self, part):
        """Determine if the part is an operator or identifier."""
        if part in self.valid_operators:
            return 'operator'
        elif self.is_identifier(part):
            return 'identifier'
        else:
            return 'invalid'

def tokenize(expression):
    """Manually split input into identifiers and operators."""
    tokens = []
    current_token = ""

    i = 0
    while i < len(expression):
        char = expression[i]
```

```
        if char.isalnum() or char == '_':
            # Part of an identifier
            current_token += char
        else:
            # If there's an identifier before, add it
            if current_token:
                tokens.append(current_token)
                current_token = ""

            # Handle multi-character operators (like **, //, +=, etc.)
            if i + 1 < len(expression) and (char + expression[i + 1]) in
OperatorAutomaton().valid_operators:
                tokens.append(char + expression[i + 1])
                i += 1  # Skip the next character since it's part of the operator
            elif char in OperatorAutomaton().valid_operators:
                tokens.append(char)

        i += 1

    # Add the last token if there is one
    if current_token:
        tokens.append(current_token)

    return tokens

def main():
    # Initialize the automaton
    automaton = OperatorAutomaton()

    # Take user input (e.g., a+b)
    input_string = input("Enter a string: ")

    # Tokenize the input string without using `re`
    tokens = tokenize(input_string)

    # Process each token (operator or identifier)
    for token in tokens:
        result = automaton.process(token)
        if result == 'operator':
            print(f"{token} is an operator.")
        elif result == 'identifier':
            print(f"{token} is an identifier.")
        else:
            print(f"{token} is invalid.")

if __name__ == "__main__":
```

```
main()
```

Output:

```
debugpy-2025.0.0-win32-x64\bundled\libs\debugpy\launcher' '51
Enter a string: a+b=c
a is an identifier.
+ is an operator.
b is an identifier.
= is an operator.
c is an identifier.
```

c) Write a program to recognize the valid number.

Code:

```
class NumberAutomaton:
    def __init__(self):
        self.state = 0  # Initial state
        self.final_states = {2, 4, 7}  # Accepting states

    def transition(self, char):
        if self.state == 0:
            if char in "+-":
                self.state = 1  # Sign detected
            elif char.isdigit():
                self.state = 2  # Integer part
            elif char == '.':
                self.state = 3  # Decimal point without leading digit (invalid alone)
            else:
                self.state = -1  # Invalid state
        elif self.state == 1:
            if char.isdigit():
                self.state = 2  # Integer part after sign
            elif char == '.':
                self.state = 3  # Decimal point after sign
            else:
                self.state = -1
        elif self.state == 2:
            if char.isdigit():
                self.state = 2  # Continue integer part
            elif char == '.':
                self.state = 4  # Decimal point after integer
            elif char in "eE":
                self.state = 5  # Exponential notation
            else:
```

```
            self.state = -1
        elif self.state == 3:
            if char.isdigit():
                self.state = 4  # Decimal fraction part
            else:
                self.state = -1
        elif self.state == 4:
            if char.isdigit():
                self.state = 4  # Continue fraction part
            elif char in "eE":
                self.state = 5  # Exponential notation
            else:
                self.state = -1
        elif self.state == 5:
            if char in "+-":
                self.state = 6  # Sign after exponent
            elif char.isdigit():
                self.state = 7  # Exponent part
            else:
                self.state = -1
        elif self.state == 6:
            if char.isdigit():
                self.state = 7  # Exponent part after sign
            else:
                self.state = -1
        elif self.state == 7:
            if char.isdigit():
                self.state = 7  # Continue exponent part
            else:
                self.state = -1
        else:
            self.state = -1

    def is_valid_number(self, s):
        self.state = 0  # Reset state
        for char in s.strip():
            self.transition(char)
            if self.state == -1:
                return False
        return self.state in self.final_states

# Example usage
num_automaton = NumberAutomaton()
user_input = input("Enter a number: ")
print(f"'{user_input}' is valid: {num_automaton.is_valid_number(user_input)}")
```

Output:

```
PS C:\Users\USER\Desktop\SEM 6> 
Enter a number: 44e884
'44e884' is valid: True
PS C:\Users\USER\Desktop\SEM 6>
```

```
PS C:\Users\USER\Desktop\SEM 6>
Enter a number: 455e22.78
'455e22.78' is valid: False
PS C:\Users\USER\Desktop\SEM 6>
```

d) Write a program to recognize the valid comments.

Code:

```
def create_input_file():
    # Create the input file with sample content
    try:
        with open("input.txt", "w") as file:
            file.write("""This is a line of code.
// This is a single-line comment
More code here.
/* This is a multi-line comment
spanning multiple lines */
Final line of code.""")
        print("input.txt file has been created with sample content.")
    except Exception as e:
        print(f"An error occurred while creating the file: {e}")


def main():
    # Create the input file if it doesn't exist
    create_input_file()

    try:
        # Read the input from the file
        with open("input.txt", "r") as file:
            input_string = file.read()  # Read the entire content
```

```
except FileNotFoundError:
    print("Error: input.txt not found!")
    return
except Exception as e:
    print(f"An error occurred: {e}")
    return

i = 0
state = 0  # Initial state

while i < len(input_string):
    char = input_string[i]

    if state == 0:
        # Looking for the start of a comment
        if char == '/':
            state = 1  # First '/' found, move to state 1
        else:
            state = 0  # Continue checking for comments

    elif state == 1:
        # Looking for the second '/' (single-line) or '*' (multi-line)
        if char == '/':
            state = 2  # Found '//', move to state 2 (single-line comment)
            print("Valid for single-line comment")  # Print for single-line comment
        elif char == '*':
            state = 3  # Found '/*', move to state 3 (multi-line comment)
            print("Valid for multi-line comment start")  # Print for multi-line comment start
        else:
            state = 5  # Invalid transition, set to invalid state
            print("Invalid state at position:", i)

    elif state == 2:
        # Inside a single-line comment, look for newline to end it
        if char == '\n':
            state = 0  # End of single-line comment, reset to state 0
        else:
            state = 2  # Continue inside single-line comment

    elif state == 3:
        # Inside a multi-line comment, looking for the closing '*/'
        if char == '*':
            state = 4  # Found '*', move to state 4
        elif char == '\n':
            state = 3  # Continue inside multi-line comment on new lines
        else:
            state = 3  # Continue in multi-line comment
```

```
    elif state == 4:
        # After '*' in multi-line comment, looking for '/' to close
        if char == '/':
            state = 0  # Found '*/', closing multi-line comment, reset to state 0
            print("Valid for multi-line comment end")  # Print for multi-line comment end
        else:
            state = 3  # If not '/', stay in multi-line comment

    elif state == 5:
        # Invalid state encountered, print invalid
        print("Invalid state at position:", i)
        state = 5  # Stay in invalid state

    i += 1

  # After processing, check if the comment structure is valid
  if state == 0:
      print("String is valid")
  else:
      print("String is invalid")


if __name__ == "__main__":
  main()
```

Output:

```
[Running] python -u "c:\Users\USER\Desktop\SEM 6\CD Lab\2(d).py"
input.txt file has been created with sample content.
Valid for single-line comment
Valid for multi-line comment start
Valid for multi-line comment end
String is valid

[Done] exited with code=0 in 0.165 seconds
```

e) Program to implement Lexical Analyzer.

CODE:

```
import re

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

    if lexeme in keywords:
        print(f"{lexeme} is a keyword")
    elif re.fullmatch(r'\d+(\.\d+)?', lexeme):
        print(f"{lexeme} is a valid number")
    elif re.fullmatch(r'\d+[a-zA-Z_]+', lexeme) or
re.fullmatch(r'\d+\.\d+\.\d+', lexeme):
        print(f"{lexeme} is an invalid number")
        invalid_tokens.append(lexeme)
    elif re.fullmatch(r'[^a-zA-Z0-9_]', lexeme):
        print(f"{lexeme} is an invalid identifier")
        invalid_tokens.append(lexeme)
    else:
        print(f"{lexeme} is an identifier")

def lexical_analysis(filename):
    with open(filename, 'r') as file:
        buffer = file.read()

    i = 0
    length = len(buffer)
    lexeme = ""
    invalid_tokens = []
    operators = {'+', '-', '*', '/', '%', '//', '**', '=', '==', '!=', '<',
'>', '<=', '>=', 'and', 'or', 'not'}
    special_chars = {'(', ')', '[', ']', '{', '}', ',', ':', '.', ';'}

    while i < length:
        c = buffer[i]

        if c.isalpha() or c == '_':  # Identifier or keyword
            lexeme += c
            i += 1
            while i < length and (buffer[i].isalnum() or buffer[i] == '_'):
                lexeme += buffer[i]
                i += 1
```

```
            check(lexeme, invalid_tokens)
            lexeme = ""

        elif c.isdigit():  # Number detection
            lexeme += c
            i += 1
            while i < length and buffer[i].isdigit():
                lexeme += buffer[i]
                i += 1
            if i < length and buffer[i] == '.':
                lexeme += '.'
                i += 1
                while i < length and buffer[i].isdigit():
                    lexeme += buffer[i]
                    i += 1
            check(lexeme, invalid_tokens)
            lexeme = ""

        elif c in operators:  # Operator detection
            lexeme += c
            i += 1
            if i < length and buffer[i] in operators:
                lexeme += buffer[i]
                i += 1
            print(f"{lexeme} is an operator")
            lexeme = ""

        elif c in special_chars:  # Special character detection
            print(f"{c} is a special character")
            i += 1

        elif c == '#':  # Handling comments
            while i < length and buffer[i] != '\n':
                i += 1

        elif c.isspace():  # Ignore whitespace
            i += 1

        else:
            invalid_tokens.append(c)
            print(f"{c} is an invalid token")
            i += 1

    print("\nInvalid Tokens List:", invalid_tokens)

if __name__ == "__main__":
    lexical_analysis("input.txt")
```

Input.txt:

File Edit View

```
int a = 10;
float b = 20.5;
if (a < b) {
    a = a + 1;
}
```

OUTPUT:

```
PS C:\Users\USER\Desktop\SEM 6\CD Lab>  & 'c:\Users\USER\AppData\Local\Programs\Python\Pyth
25.8.0-win32-x64\bundled\libs\debugpy\launcher' '56803' '--' 'c:\Users\USER\Desktop\SEM 6\C
int is an identifier
a is an identifier
= is an operator
10 is a valid number
; is a special character
float is an identifier
b is an identifier
= is an operator
20.5 is a valid number
; is a special character
if is a keyword
( is a special character
a is an identifier
< is an operator
b is an identifier
) is a special character
{ is a special character
a is an identifier
= is an operator
a is an identifier
+ is an operator
1 is a valid number
; is a special character
} is a special character

Invalid Tokens List: []
```

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

4. Implement following programs using Lex.

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

Code:

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

  %%
  \n     { line_count++; char_count++; }
  [ \t]  { char_count++; }
  \w+    { word_count++; char_count += yyleng; }
  .      { char_count++; }
  %%

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

     printf("Number of characters: %d\n", char_count);
     printf("Number of words: %d\n", word_count);
     printf("Number of lines: %d\n", line_count);

     return 0;
  }

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

Input.txt:

```
Hello, this is a sample text file.
It contains multiple lines and words.

Lex is used for lexical analysis.
Counting characters, words, and lines is useful.
```

Output:

```
Number of characters: 162
Number of words: 2
Number of lines: 5


...Program finished with exit code 0
Press ENTER to exit console.
```

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

Code:

```
%{
  #include<stdio.h>
  extern FILE *yyin;
  int consonants=0, vowels=0;
  %}
  %%
  [aeiouAEIOU] { vowels++; }
  [a-zA-Z] { consonants++; }
  . ;
  %%
  int main() {
     FILE *file = fopen("input.txt", "r");
     if (!file) {
        perror("Error opening file");
        return 1;
     }
```

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

    printf("This File contains ...");
    printf("\n\t%d vowels", vowels);
    printf("\n\t%d consonants", consonants);

    return 0;
}

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

Input.txt:

```
Hello, this is a sample text file.
It contains multiple lines and words.

Lex is used for lexical analysis.
Counting characters, words, and lines is useful.
```

Output:

```
This File contains ...
        46 vowels
        77 consonants

...Program finished with exit code 0
Press ENTER to exit console.
```

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

Code:

```
%{
  #include<stdio.h>
  extern FILE *yyin;
  %}
  %%
  -?[0-9]+(\.[0-9]+)? { printf("%s\n", yytext); }
  . ;
  %%
  int main() {
     FILE *file = fopen("input.txt", "r");
     if (!file) {
        perror("Error opening file");
        return 1;
     }

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

     return 0;
  }

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

Input.txt:

```
42
-789
12345
56.78
-908.172
300
-0.99
3.14159
-2.718
```

Output:

```
42

-789

12345

56.78

-908.172

300

-0.99

3.14159

-2.718


...Program finished with exit code 0
Press ENTER to exit console.
```

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

Code:

```
%{
  #include<stdio.h>
  extern FILE *yyin;
  int line_number = 1;
  FILE *output;
  %}
  %%
  ^.* { fprintf(output, "%d: %s\n", line_number++, yytext); }
  . ;
  %%
  int main() {
     FILE *file = fopen("input.txt", "r");
     if (!file) {
        perror("Error opening input file");
        return 1;
     }

     output = fopen("output.txt", "w");
     if (!output) {
        perror("Error opening output file");
        fclose(file);
        return 1;
     }

     yyin = file;
```

```
    yylex();
    fclose(file);
    fclose(output);

    return 0;
}

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

Input.txt:

```
Hello, this is a test file.
It contains multiple lines.
Each line will be numbered.
```

Output:

```
main.c    input.txt    output.txt
1  1: Hello, this is a test file.
2  2: It contains multiple lines.
3  3: Each line will be numbered.
4
```

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

CODE:

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

%%
"<!--"([^-\n]|"-"[^-\n])*"-->"     { printf("HTML COMMENT: %s\n", yytext); }
"<"[^>"]+">"                       { printf("MARKUP TAG: %s\n", yytext); }
.|\n                               { /* ignore other characters */ }
%%

int yywrap() {
    return 1;
```

```
}

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

Input.txt:

File Edit View

```
<html>
  <head>
    <!-- This is a comment -->
    <title>Page</title>
  </head>
</html>
```

OUTPUT:

```
C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\4.e>flex code.l

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\4.e>gcc lex.yy.c

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\4.e>a.exe
<html>
MARKUP TAG: <html>
  <head>
MARKUP TAG: <head>
    <!-- This is a comment -->
HTML COMMENT: <!-- This is a comment -->
    <title>Page</title>
MARKUP TAG: <title>
MARKUP TAG: </title>
  </head>
MARKUP TAG: </head>
</html>
MARKUP TAG: </html>
```

5)

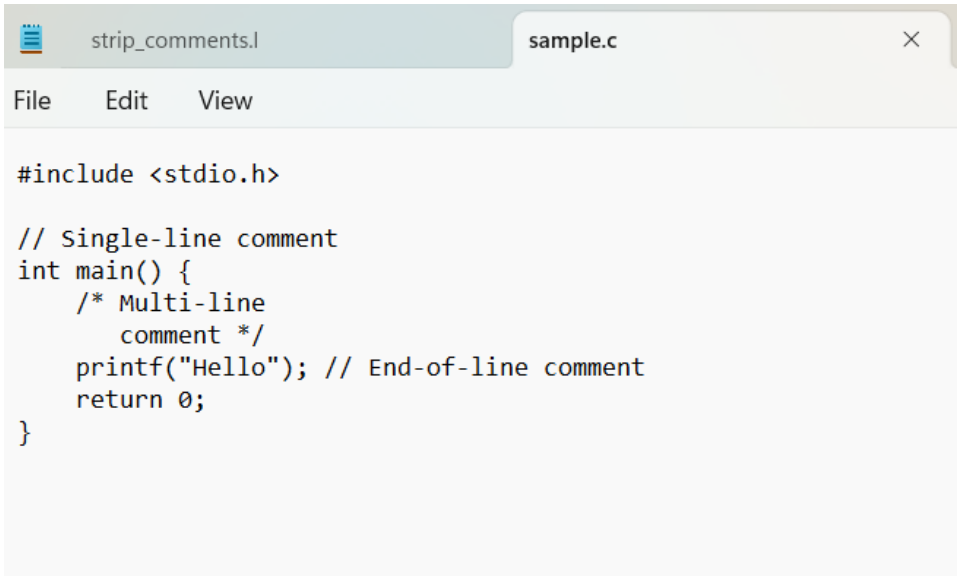
a) Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.

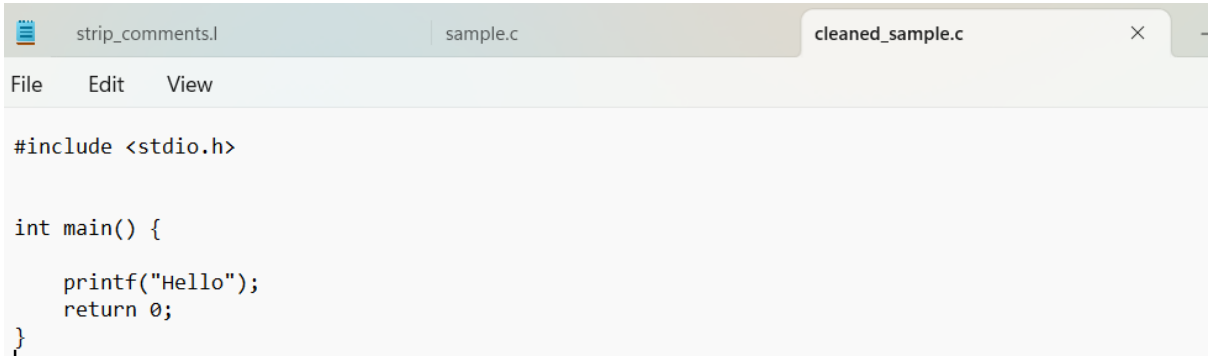
CODE:

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

%%
"//".* {
    comment_line_count++;
    /* Ignore the comment, don't write it to the output file */
}

"/*"([^*]|\*+[^*/])*"*"+"/" {
    // Count the number of lines in the multiline comment
    for (int i = 0; yytext[i] != '\0'; i++) {
        if (yytext[i] == '\n') {
            comment_line_count++;
        }
    }
    /* Do not write to output */
}

.|\n {
    // Write all other characters (including newlines) to the output file
    fputc(yytext[0], output);
}
%%

int yywrap() {
    return 1;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input.c output.c\n", argv[0]);
        return 1;
    }

    FILE *input = fopen(argv[1], "r");
    if (!input) {
        perror("Error opening input file");
```

```
        return 1;
    }

    output = fopen(argv[2], "w");
    if (!output) {
        perror("Error opening output file");
        fclose(input);
        return 1;
    }

    yyin = input;
    yylex();

    fclose(input);
    fclose(output);

    printf("Total comment lines removed: %d\n", comment_line_count);
    return 0;
}
```

Input.txt:





OUTPUT:

```
Microsoft Windows [Version 10.0.26100.3915]
(c) Microsoft Corporation. All rights reserved.

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.a>flex strip_comments.l

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.a>gcc lex.yy.c -o strip_comments

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.a>strip_comments sample.c cleaned_sample.c
Total comment lines removed: 3

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.a>
```

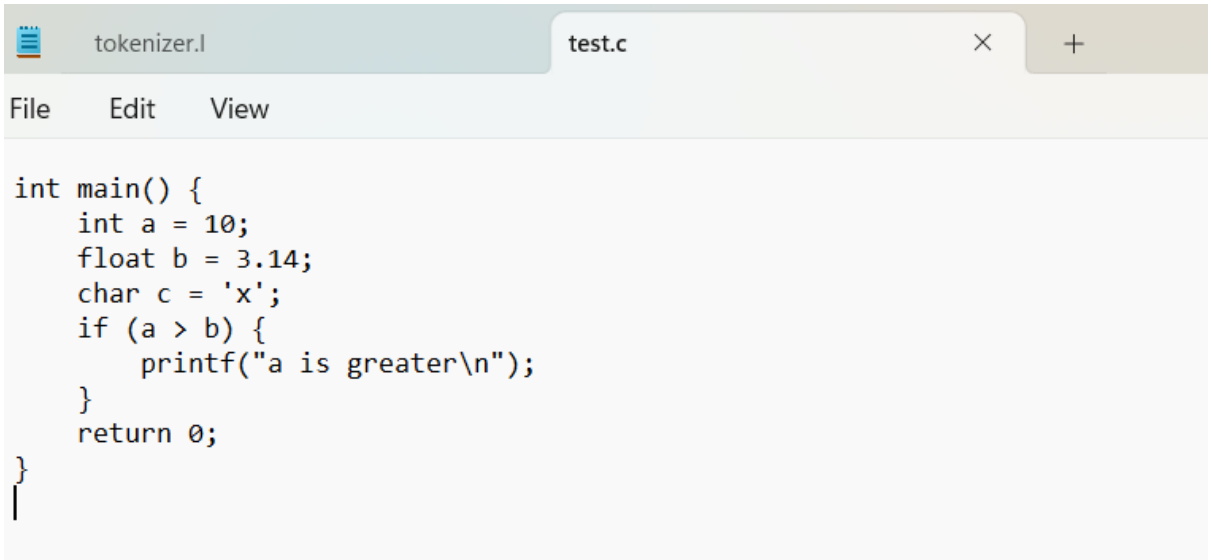
b) Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

CODE:

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

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", NULL
};

int isKeyword(const char *word) {
    for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(keywords[i], word) == 0)
            return 1;
    }
    return 0;
}
%}

IDENTIFIER    [a-zA-Z_][a-zA-Z0-9_]*
NUMBER        [0-9]+(\.[0-9]+)?
OPERATOR      (\+|\-|\*|\/|%|==|=|<=|>=|!=|<|>|&&|\|\||\+\+|\-\-)
SPECIAL       [\[\]{}\(\),;:.]
STRING        \"([^\\\"]|\\.)*\"
CHAR          \'([^\\\']|\\.)\'

%%
{STRING}      { printf("STRING LITERAL: %s\n", yytext); }
{CHAR}        { printf("CHARACTER LITERAL: %s\n", yytext); }
```

```
{NUMBER}      { printf("NUMBER: %s\n", yytext); }
{OPERATOR}    { printf("OPERATOR: %s\n", yytext); }
{SPECIAL}     { printf("SPECIAL SYMBOL: %s\n", yytext); }
{IDENTIFIER}  {
                if (isKeyword(yytext))
                    printf("KEYWORD: %s\n", yytext);
                else
                    printf("IDENTIFIER: %s\n", yytext);
              }
[ \t\n]+      { /* Ignore whitespace */ }
.             { printf("UNKNOWN TOKEN: %s\n", yytext); }
%%

int yywrap() {
    return 1;
}

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

Input.txt:



OUTPUT:

```
(c) Microsoft Corporation. All rights reserved.

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.b>flex tokenizer.l

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.b>gcc lex.yy.c -o tokenizer

C:\Users\USER\Desktop\SEM 6\CD Lab\Lexical\5.b>tokenizer test.c
KEYWORD: int
IDENTIFIER: main
SPECIAL SYMBOL: (
SPECIAL SYMBOL: )
SPECIAL SYMBOL: {
KEYWORD: int
IDENTIFIER: a
OPERATOR: =
NUMBER: 10
SPECIAL SYMBOL: ;
KEYWORD: float
IDENTIFIER: b
OPERATOR: =
NUMBER: 3.14
SPECIAL SYMBOL: ;
KEYWORD: char
IDENTIFIER: c
OPERATOR: =
CHARACTER LITERAL: 'x'
SPECIAL SYMBOL: ;
KEYWORD: if
SPECIAL SYMBOL: (
IDENTIFIER: a
OPERATOR: >
IDENTIFIER: b
SPECIAL SYMBOL: )
SPECIAL SYMBOL: {
IDENTIFIER: printf
SPECIAL SYMBOL: (
STRING LITERAL: "a is greater\n"
SPECIAL SYMBOL: )
SPECIAL SYMBOL: ;
SPECIAL SYMBOL: }
KEYWORD: return
NUMBER: 0
SPECIAL SYMBOL: ;
SPECIAL SYMBOL: }
```

6) Program to implement Recursive Descent Parsing in C.

CODE:

```
i = 0
l = ''
S = ""  # Input string

def match(t):
    global i, l
    if l == t:
        i += 1
        if i < len(S):
            l = S[i]
        else:
            l = '\0'  # End of input
    else:
        print("error")
        exit(1)

def T():
    global l
    while l == '+' or l == '-':
        op = l
        match(op)
        if l.isalpha():
            match(l)
        else:
            print("error")
            exit(1)

def E():
    global l
    if l.isalpha():  # Accept any alphabetic character as an identifier
        match(l)
        T()
    else:
        print("error")
        exit(1)

def main():
    global i, l, S
    S = input("\nEnter the string: ")
    S += '\0'  # End-of-string marker
    l = S[0]

    E()

    if l == '\0':  # End of input
```

```
        print("Success")
    else:
        print("syntax error")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\USER\Desktop\SEM 6\CD Lab>  & 'c:\Users\USER\AppData\Local\Programs\Pytho
25.8.0-win32-x64\bundled\libs\debugpy\launcher' '58192' '--' 'c:\Users\USER\Desktop\S

Enter the string: a+b-c
Success
PS C:\Users\USER\Desktop\SEM 6\CD Lab>
```

```
PS C:\Users\USER\Desktop\SEM 6\CD Lab>  & 'c:\Users\USER\AppData\Local'
25.8.0-win32-x64\bundled\libs\debugpy\launcher' '58211' '--' 'c:\Users'

Enter the string: i+
error
PS C:\Users\USER\Desktop\SEM 6\CD Lab>
```

7)

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

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

CODE:

Parser.y:

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

%union {
    double dval;
}

%token <dval> NUMBER
%token LPAREN RPAREN
%left '+' '-' /* Lowest precedence */
%left '*' '/' /* Higher precedence */
%type <dval> expr term primary

%%
program:
    lines { /* Process lines */ }
    ;

lines:
    lines line
    | /* empty */
    ;

line:
    expr opt_newline { printf("Result: %.2f\n", $1); }
    | error '\n' { yyerror("Syntax Error In Expression"); yyerrok; }
    ;

opt_newline:
    '\n'
    | /* empty */
    ;
```

```
expr:
    expr '+' term     { $$ = $1 + $3; }
    | expr '-' term   { $$ = $1 - $3; }
    | term
    ;

term:
    term '*' primary  { $$ = $1 * $3; }
    | term '/' primary {
        if ($3 == 0) {
            yyerror("Division By Zero");
            $$ = 0;
        } else {
            $$ = $1 / $3;
        }
    }
    | primary
    ;

primary:
    NUMBER            { $$ = $1; }
    | LPAREN expr RPAREN { $$ = $2; }
    ;

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

int main(int argc, char *argv[]) {
    yyin = fopen("input.txt", "r");
    if (!yyin) {
        fprintf(stderr, "Cannot Open Input.txt\n");
        return 1;
    }
    yyparse();
    fclose(yyin);
    return 0;
}
```

Lexer.l:

```
%{
#include "parser.tab.h"
%}
%option noyywrap
%%
```

```
[0-9]+(\.[0-9]+)? { yylval.dval = atof(yytext); return NUMBER; }
"+"               { return '+'; }
"-"               { return '-'; }
"*"               { return '*'; }
"/"               { return '/'; }
"("               { return LPAREN; }
")"               { return RPAREN; }
\n                { return '\n'; }
[ \t]             { /* Ignore whitespace */ }
.                 { fprintf(stderr, "Invalid character: %s\n", yytext); return
0; }
%%
```

Input.txt:

File Edit View

```
2 + 3 * 4
10 / 2 - 1
5.5 + 2.2
10 / 0
(2 + 3) * 4
```

OUTPUT:

```
Enter arithmetic expressions (CTRL+D to end):
3 + 4 * 2
Result: 11

10 / 2 - 1
Result: 4
```

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

CODE:

Parser.y:

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

%union {
    int ival; /* For integer arguments */
}

%token <ival> NUMBER
%token LPAREN RPAREN
%left '+' '-' /* Lowest precedence */
%left '*' '/' /* Higher precedence */
%type <ival> expr term primary

%%
program:
    lines { /* Process lines */ }
    ;

lines:
    lines line
    | /* empty */
    ;

line:
    expr opt_newline { printf("Result: %d\n", $1); }
    | error '\n' { yyerror("Syntax Error In Expression"); yyerrok; }
    ;

opt_newline:
    '\n'
    | /* empty */
    ;

expr:
    expr '+' term { $$ = $1 + $3; }
    | expr '-' term { $$ = $1 - $3; }
    | term
```

```
    ;

term:
    term '*' primary { $$ = $1 * $3; }
    | term '/' primary {
        if ($3 == 0) {
            yyerror("Division By Zero");
            $$ = 0;
        } else {
            $$ = $1 / $3;
        }
    }
    | primary
    ;

primary:
    NUMBER { $$ = $1; }
    | LPAREN expr RPAREN { $$ = $2; }
    ;

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

int main(int argc, char *argv[]) {
    yyin = fopen("input.txt", "r");
    if (!yyin) {
        fprintf(stderr, "Cannot Open Input.txt\n");
        return 1;
    }
    yyparse();
    fclose(yyin);
    return 0;
}
```

Lexer.l:

```
%{
#include "parser.tab.h"
%}
%option noyywrap
%%
[0-9]+            { yylval.ival = atoi(yytext); return NUMBER; }
"+"               { return '+'; }
"-"               { return '-'; }
```

```
"*"             { return '*'; }
"/"             { return '/'; }
"("             { return LPAREN; }
")"             { return RPAREN; }
\n              { return '\n'; }
[ \t]           { /* Ignore whitespace */ }
.               { fprintf(stderr, "Invalid character: %s\n", yytext); return
0; }
%%
```

Input.txt:

File Edit View

```
2 + 3 * 4
10 / 2 - 1
15 + 25
10 / 0
(2 + 3) * 4
```

OUTPUT:

```
Input:  10 + 5 * 2
Output: Result: 20

Input:  25 / 5 + 3
Output: Result: 8
```

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

CODE:

Parser.y:

```
%{
#include <stdio.h>
#include <string.h>
char postfix[100]; /* Buffer for postfix expression */
int pos = 0;       /* Current position in postfix buffer */

void append(char *s) {
```

```
    strcpy(postfix + pos, s);
    pos += strlen(s);
}

void reset_postfix() {
    postfix[0] = '\0';
    pos = 0;
}

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

int yylex(void);
extern FILE *yyin;
%}

%token NUMBER PLUS MINUS TIMES DIVIDE LPAREN RPAREN NEWLINE
%left PLUS MINUS
%left TIMES DIVIDE
%right UMINUS

%%
program:
    program expr NEWLINE     { printf("Postfix: %s\n", postfix);
reset_postfix(); }
    | program NEWLINE        { /* Skip empty lines */ }
    | /* empty */            { /* Allow empty program */ }
    ;

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

%%
int main() {
    yyin = fopen("input.txt", "r");
    if (!yyin) {
        fprintf(stderr, "Could not open input.txt\n");
        return 1;
    }
```

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

Lexer.l:

```
%{
#include <stdio.h>
#include "parser.tab.h"
%}

%%
[0-9]+          { yylval = atoi(yytext); return NUMBER; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return TIMES; }
"/"             { return DIVIDE; }
"("             { return LPAREN; }
")"             { return RPAREN; }
[ \t]           ; /* Skip whitespace */
\n              { return NEWLINE; }
.               { printf("Invalid character: %s\n", yytext); }
%%

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

Input.txt:

File Edit View

```
2 + 3 * 4
(5 - 2) * 3
-2 + 4
10 / (2 + 3)
```

OUTPUT:

```
Postfix: 5 3 2 * +
```