System Programming and Compiler Design Laboratory
(CSPC-421)

B.Tech VIIth Semester
(August – December 2024)

Submitted by

Ashutosh Jha (21103029)
Group-G2

Submitted to

Mr. Rahul Kumar

Department of Computer Science & Engineering
Dr. B.R. Ambedkar National Institute of Technology Jalandhar
-144008, Punjab, India

Table of Content

Sr. No.	Practical Name	Date	Page No.	Remarks
1.	a) Write a program to simulate the behaviour of DFA for recognizing valid C++ identifier. b) Write a program to simulate the behaviour of DFA for recognizing valid signed integer.	29-07-2024	1-4	
2.	a) Write a program to simulate the behaviour of DFA for recognizing valid signed real number. b) Write a program to simulate the behaviour of DFA for recognizing the following valid C++ keywords- if, else, int, float, void, char, do, while. c) Write a program to simulate the behaviour of DFA for recognizing the constants. d) Write a program to simulate the behaviour of DFA for recognizing the operators of the mini language. e) Write a program to simulate the behaviour of DFA for recognizing string under ‘a’, ‘a*b+’, ‘abb’.	05-08-2024	5-13	
3.	a) Write a lex code to print "Hello" message on matching a string "Hi" and print "Wrong" message otherwise. b) Write a lex code to match ‘odd’ or ‘even’ numbers from the input.	12-08-2024	14-15	
4.	a) Write a Lex code to count and print the number of lines, tabs. Space and characters in given input stream. b) Desing a Lex code to count and print the number of total characters, word, white spaces in given “Input.txt” file.	05-09-2024	16-19	
5.	a) Design a Lex code to remove the comments from any C-program given at run-time and store into “out.c” file. b) Design a Lex code to extract all html tags in the given HTML file at run time and store into html.text file given at run time.	19-09-2024	20-24	

Practical#1

Objective:

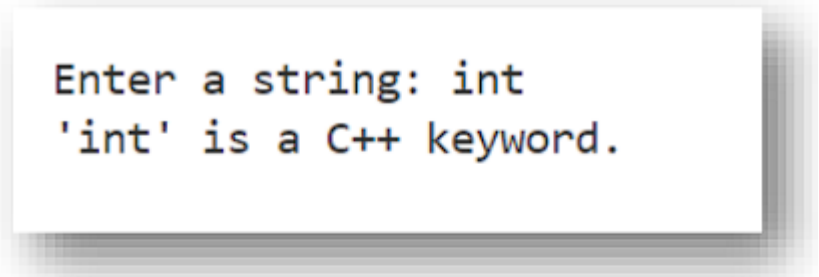
1. Write a program to simulate the behaviour of DFA for recognizing valid C++ identifier

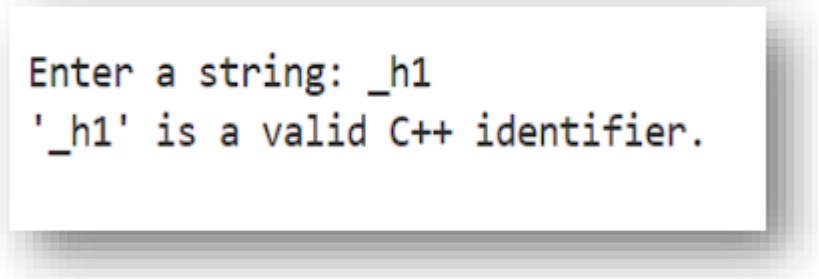
Code:

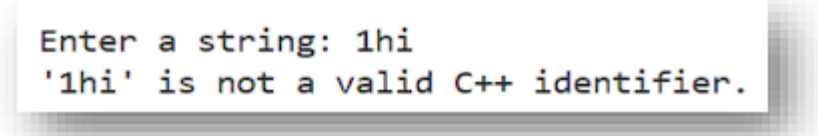
```
#include <bits/stdc++.h>
using namespace std;
enum State
{
    START,
    IDENTIFIER,
    DEAD
};
unordered_set<string> keywords = {
    "auto", "break", "case", "char", "const", "continue", "default",
"do", "double",
    "else", "enum", "extern", "float", "for", "goto", "if", "int",
"long",
    "register", "return", "short", "signed", "sizeof", "static",
"struct", "switch",
    "typedef", "union", "unsigned", "void", "volatile", "while"};
State transition(State state, char c)
{
    switch (state)
    {
    case START:
        if (isalpha(c) || c == '_')
            return IDENTIFIER;
        return DEAD;
    case IDENTIFIER:
        if (isalnum(c) || c == '_')
            return IDENTIFIER;
        return DEAD;
    case DEAD:
        return DEAD;
    }
    return DEAD;
}
bool isValidIdentifier(const string &str)
{
    State state = START;
    for (char c : str)
    {
        state = transition(state, c);
        if (state == DEAD)
        {

            return false;
        }
```

```
    }
    return state == IDENTIFIER && keywords.find(str) == keywords.end();
}
bool isKeyword(const string &str)
{
    return keywords.find(str) != keywords.end();
}
int main()
{
    string input;
    cout << "Enter a string: ";
    cin >> input;
    if (isKeyword(input))
    {
        cout << "'" << input << "' is a C++ keyword." << endl;
    }
    else if (isValidIdentifier(input))
    {
        cout << "'" << input << "' is a valid C++ identifier." << endl;
    }
    else
    {
        cout << "'" << input << "' is not a valid C++ identifier." <<
endl;
    }
    return 0;
}
```

Output:







2. Write a program to simulate the behaviour of DFA for recognizing valid signed integer.

Code:

```
#include <bits/stdc++.h>
using namespace std;
enum State
{
    START,
    SIGNED,
    INTEGER,
    DEAD
};
bool SignedInteger(const string &str)
{
    State state = START;
    for (char c : str)
    {
        switch (state)
        {
        case START:
            if (c == '+' || c == '-')
            {
                state = SIGNED;
            }
            else if (isdigit(c))
            {
                state = INTEGER;
            }
            else
            {
                state = DEAD;
            }
            break;
        case SIGNED:
            if (isdigit(c))
            {
                state = INTEGER;
            }
            else
            {
                state = DEAD;
            }
            break;
        case INTEGER:
            if (isdigit(c))
            {
                state = INTEGER;
            }
            else
            {
                state = DEAD;
```

```
            }
            break;
        case DEAD:
            return false;
        }
    }
    return state == INTEGER;
}
int main()
{
    string input;
    cout << "Enter a string: ";
    cin >> input;
    if (SignedInteger(input))
    {
        cout << "'" << input << "' is a signed integer." << endl;
    }
    else
    {
        cout << "'" << input << "' is not a signed integer." << endl;
    }
    return 0;
}
```

Output:

```
Enter a string: -98
'-98' is a signed integer.
```

```
Enter a string: --98
'--98' is not a signed integer.
```

```
Enter a string: -9.8
'-9.8' is not a signed integer.
```

Practical#2

Objective:

1. Write a program to simulate the behaviour of DFA for recognizing valid signed real number.

Code:

```
#include <bits/stdc++.h>
using namespace std;
enum State
{
    START,
    SIGNED,
    INTEGER,
    DECIMAL,
    FRACTION,
    DEAD
};
bool SignedRealNumber(const string &str)
{
    State state = START;
    for (char c : str)
    {
        switch (state)
        {
        case START:
            if (c == '+' || c == '-')
            {
                state = SIGNED;
            }
            else if (isdigit(c))
            {
                state = INTEGER;
            }
            else
            {
                state = DEAD;
            }
            break;
        case SIGNED:
            if (isdigit(c))

            {
                state = INTEGER;
            }
            else
            {
                state = DEAD;
            }
            break;
        case INTEGER:
```

```
            if (isdigit(c))
            {
                state = INTEGER;
            }
            else if (c == '.')
            {
                state = DECIMAL;
            }
            else
            {
                state = DEAD;
            }
            break;
        case DECIMAL:
            if (isdigit(c))
            {
                state = FRACTION;
            }
            else
            {
                state = DEAD;
            }
            break;
        case FRACTION:
            if (isdigit(c))
            {
                state = FRACTION;
            }
            else
            {
                state = DEAD;
            }

            break;
        case DEAD:
            return false;
        }
    }
    return state == FRACTION;
}
int main()
{
    string input;
    cout << "Enter a number: ";
    cin >> input;
    if (SignedRealNumber(input))
    {
        cout << "'" << input << "' is a signed real number." << endl;
    }
    else
    {
        cout << "'" << input << "' is not a signed real number." <<
endl;
    }
```

```
    return 0;
}
```

Output:

```
Enter a number: -9.87
'-9.87' is a signed real number.
```

```
Enter a number: +-9.87
'+-9.87' is not a signed real number.
```

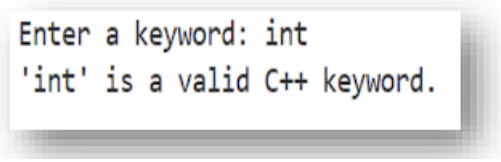
2. Write a program to simulate the behaviour of DFA for recognizing the following valid C++ keywords- if, else, int, float, void, char, do, while.

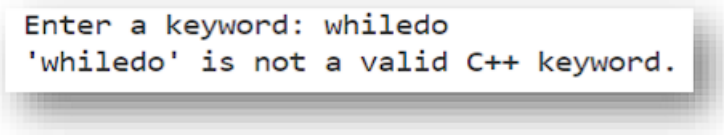
Code:

```
#include <bits/stdc++.h>
using namespace std;
enum State
{
    START,
    DEAD
};
unordered_set<string> keywords = {
    "if", "else", "int", "float", "void", "char", "do", "while"};
State transition(State state, char c)
{
    switch (state)
    {
    case START:
        if (isalpha(c))
            return START;
        return DEAD;
    case DEAD:
        return DEAD;
    }
    return DEAD;
}
bool isKeyword(const string &str)
{
    return keywords.find(str) != keywords.end();
}
int main()
{
    string input;
    cout << "Enter a keyword: ";
    cin >> input;
    if (isKeyword(input))
    {
```

```
        cout << "'" << input << "' is a valid C++ keyword." << endl;
    }
    else
    {
        cout << "'" << input << "' is not a valid C++ keyword." <<
endl;
    }
    return 0;
}
```

Output:





3. **Write a program to simulate the behaviour of DFA for recognizing the constants.**

Code:

```
#include <iostream>
#include <string>
using namespace std;
enum State
{
    START,       // Starting state
    SIGN,        // State after a sign (+/-)
    INTEGER,     // State after reading an integer part
    DOT,         // State after reading a decimal point
    FRACTION,    // State after reading the fractional part
    EXPONENT,    // State after reading 'e' or 'E'
    EXP_SIGN,    // State after a sign in the exponent
    EXP_NUMBER, // State after reading the exponent part
    DEAD         // Dead state for invalid input
};
State transition(State state, char c)
{
    switch (state)
    {
    case START:
        if (isdigit(c))
            return INTEGER;
        if (c == '+' || c == '-')
            return SIGN;
        return DEAD;
    case SIGN:
        if (isdigit(c))
            return INTEGER;
        return DEAD;
```

```
    case INTEGER:
        if (isdigit(c))
            return INTEGER;
        if (c == '.')
            return DOT;
        if (c == 'e' || c == 'E')
            return EXPONENT;
        return DEAD;
    case DOT:
        if (isdigit(c))
            return FRACTION;
        return DEAD;

    case FRACTION:
        if (isdigit(c))
            return FRACTION;
        if (c == 'e' || c == 'E')
            return EXPONENT;
        return DEAD;
    case EXPONENT:
        if (isdigit(c))
            return EXP_NUMBER;
        if (c == '+' || c == '-')
            return EXP_SIGN;
        return DEAD;
    case EXP_SIGN:
        if (isdigit(c))
            return EXP_NUMBER;
        return DEAD;
    case EXP_NUMBER:
        if (isdigit(c))
            return EXP_NUMBER;
        return DEAD;
    case DEAD:
        return DEAD;
    }
    return DEAD;
}
bool isConstant(const string &str)
{
    State state = START;
    for (char c : str)
    {
        state = transition(state, c);
        if (state == DEAD)
        {
            return false;
        }
    }
    return state == INTEGER || state == FRACTION || state ==
EXP_NUMBER;
}
int main()
{
```

```
    string input;
    cout << "Enter a constant: ";
    cin >> input;
    if (isConstant(input))
    {

        cout << "'" << input << "' is a valid constant." << endl;
    }
    else
    {
        cout << "'" << input << "' is not a valid constant." << endl;
    }
    return 0;
}
```

Output:

```
Enter a constant: 98
'98' is a valid constant.
```

```
Enter a constant: -9.8
'-9.8' is a valid constant.
```

```
Enter a constant: h1
'h1' is not a valid constant.
```

4. Write a program to simulate the behaviour of DFA for the operators of the mini language.

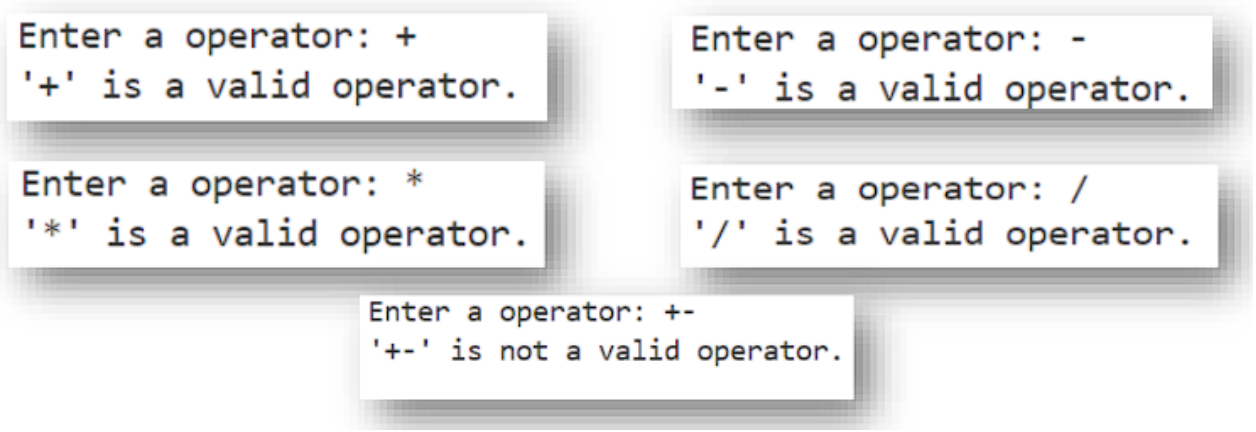
Code:

```
#include <bits/stdc++.h>
using namespace std;
enum State
{
    START, // Starting state
    PLUS,  // State for +
    MINUS, // State for -
    MULT,  // State for *
    DIV,   // State for /
    EQ,    // State for =
    LT,    // State for <
    GT,    // State for >
    AND,   // State for &
    OR,    // State for |
    NOT,   // State for !
    DEAD   // Dead state for invalid input
};
State transition(State state, char c)
{
    switch (state)
    {
```

```
    case START:
        if (c == '+')
            return PLUS;
        if (c == '-')
            return MINUS;
        if (c == '*')
            return MULT;
        if (c == '/')
            return DIV;
        if (c == '=')
            return EQ;
        if (c == '<')
            return LT;
        if (c == '>')
            return GT;
        if (c == '&')
            return AND;
        if (c == '|')
            return OR;
        if (c == '!')
            return NOT;
        return DEAD;
    case PLUS:
    case MINUS:
    case MULT:
    case DIV:
    case EQ:
    case LT:

    case GT:
    case AND:
    case OR:
    case NOT:
        return DEAD;
    case DEAD:
        return DEAD;
    }
    return DEAD;
}
bool isOperator(const string &str)
{
    State state = START;
    for (char c : str)
    {
        state = transition(state, c);
        if (state == DEAD)
        {
            return false;
        }
    }
    return state == PLUS || state == MINUS || state == MULT || state ==
DIV ||
           state == EQ || state == LT || state == GT || state == AND ||
           state == OR || state == NOT;
```

```
}
int main()
{
    string input;
    cout << "Enter a operator: ";
    cin >> input;
    if (isOperator(input))
    {
        cout << "'" << input << "' is a valid operator." << endl;
    }
    else
    {
        cout << "'" << input << "' is not a valid operator." << endl;
    }
    return 0;
}
```

Output:



5. Write a program to simulate the behaviour of DFA for recognizing string under ‘a’, ‘a*b+’, ‘abb’.

Code:

```
#include <bits/stdc++.h>
class DFA
{
public:
    DFA()
    {
        transitions = {
            {{'a', 1}},            // State 0
            {{'a', 1}, {'b', 2}}, // State 1
            {{'b', 3}},            // State 2
            {{'b', 3}}             // State 3
        };
        // Define accepting states
        acceptingStates = {1, 2, 3};
    }
```

```
    bool accepts(const string &input)
    {
        if (input.find_first_not_of('b') == string::npos)
        {
            return true;
        }
        int currentState = 0;
        for (char c : input)
        {
            if (transitions[currentState].find(c) !=
transitions[currentState].end())
            {
                currentState = transitions[currentState][c];
            }
            else
            {
                return false;
            }
        }
        return acceptingStates.find(currentState) !=
acceptingStates.end();
    }

private:
    vector<unordered_map<char, int>> transitions;
    unordered_set<int> acceptingStates;
};
int main()
{

    DFA dfa;
    string input;
    cout << "Enter a string to test with the DFA (type 'exit' to quit):
";
    while (cin >> input && input != "exit")
    {
        if (dfa.accepts(input))
        {
            cout << "The string \"" << input << "\" is accepted by the
DFA.\n";
        }
        else
        {
            cout << "The string \"" << input << "\" is not accepted by
the DFA.\n";
        }
        cout << "\nEnter another string to test (type 'exit' to
quit):";
    }
    cout << "\nCODE BY: ABHAY MISHRA\n";
    cout << "ROLL NO: 21103001";
    return 0;
}
```

Output:

```
Enter a string to test with the DFA (type 'exit' to quit): a
The string "a" is accepted by the DFA.

Enter another string to test (type 'exit' to quit):b
The string "b" is accepted by the DFA.

Enter another string to test (type 'exit' to quit):abb
The string "abb" is accepted by the DFA.

Enter another string to test (type 'exit' to quit):aaaabbbbb
The string "aaaabbbbb" is accepted by the DFA.

Enter another string to test (type 'exit' to quit):aba
The string "aba" is not accepted by the DFA.

Enter another string to test (type 'exit' to quit):ba
The string "ba" is not accepted by the DFA.

Enter another string to test (type 'exit' to quit):exit
```

Practical#3

Objective:

1. Write a lex code to print "Hello" message on matching a string "Hi" and print "Wrong" message otherwise.

Code:

```
% {
#include <stdio.h>
#include <string.h>
      void check_input(const char *input){
          if (strcmp(input, "Hi") == 0){
              printf("Hello\n");
}
else
{
    printf("Wrong\n");
}
}
%
}
% %
\n{/* Ignore newlines */}
            [^\n] +
{ check_input(yytext); }
% %
    int main(void)
{
    yylex();
    return 0;
}
```

Output:

```
Hi
Hello

What
Wrong
```

2. Write a lex code to match ‘odd’ or ‘even’ numbers from the input.

Code:

```
# % {
#include <stdio.h>
      % } %
    % [0 - 9] *[02468]
{
```

```
    printf("Even number: %s\n", yytext);
}
[0 - 9] *[13579] { printf("Odd number: %s\n", yytext); }
.|\n { ; /* Ignore other input */ }
% %
    int main(void)
{
    yylex();
    return 0;
}
```

Output:

```
234
Even number: 234

297
Odd number: 297
```

Practical#4

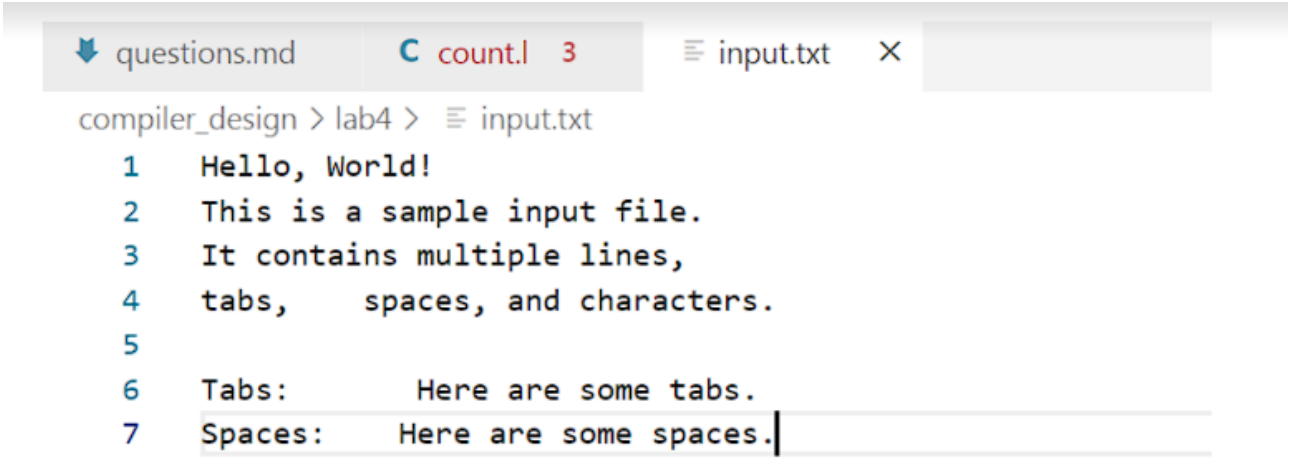
Objective:

1. Write a Lex code to count and print the number of lines, tabs. Space and characters in given input stream.

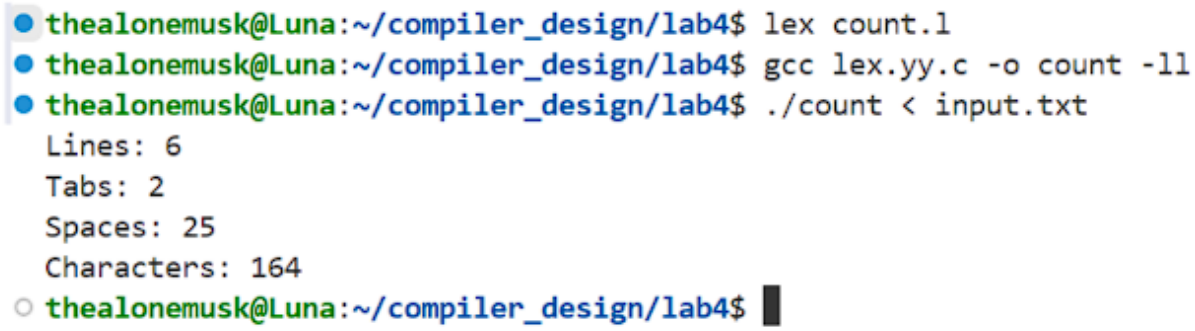
Code:

```
%
{
#include <stdio.h>
    int line_count = 0;
    int tab_count = 0;
    int space_count = 0;
    int char_count = 0;
    %
}
% %
\n
{
    line_count++;
    char_count++;
}
\t
{
    tab_count++;
    char_count++;
}
" "
{
    space_count++;
    char_count++;
}
. { char_count++; }
% %
    int main(int argc, char **argv)
{
    yylex();
    printf("Lines: %d\n", line_count);
    printf("Tabs: %d\n", tab_count);
    printf("Spaces: %d\n", space_count);
    printf("Characters: %d\n", char_count);
    return 0;
}
int yywrap()
{
    return 1;
}
```

Input:



Output:



2. Desing a Lex code to count and print the number of total characters, word, white spaces in given “Input.txt” file.input.

Code:

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

% %
        [a - zA -
            Z] +
{
    char_count += yyleng;
    word_count++;
    in_word = 1;
```

```
}
[0 - 9] +
{
    char_count += yyleng;
    if (!in_word)
    {
        word_count++;
        in_word = 1;
    }
}
[ \t] +
{
    char_count += yyleng;
    whitespace_count += yyleng;
    in_word = 0;
}
\n
{
    char_count++;
    whitespace_count++;
    in_word = 0;
}
.
{
    char_count++;
    in_word = 0;
}
% %
    int main(int argc, char **argv)
{
    yylex();
    printf("Characters: %d\n", char_count);
    printf("Words: %d\n", word_count);
    printf("White Spaces: %d\n", whitespace_count);
    return 0;
}
int yywrap()
{
    return 1;
}
```

Input:

questions.md | count.l 3 | input.txt

compiler_design > lab4 > input.txt

```
1   Hello, World!
2   This is a sample input file.
3   It contains multiple lines,
4   tabs,    spaces, and characters.
5
6   Tabs:       Here are some tabs.
7   Spaces:    Here are some spaces.
```

Output:

```
thealonemusk@Luna:~/compiler_design/lab4$ lex count2.l
thealonemusk@Luna:~/compiler_design/lab4$ gcc lex.yy.c -o count2 -ll
thealonemusk@Luna:~/compiler_design/lab4$ ./count2 < input.txt
Characters: 164
Words: 26
White Spaces: 33
thealonemusk@Luna:~/compiler_design/lab4$
```

Practical#5

Objective:

1. Design a Lex code to remove the comments from any C-program given run-time and store into “out.c” file.

Code:

```
%
%
{
#include <stdio.h>
#include <stdlib.h>
    FILE *outFile;
    %
}
% %
\/\/ [^\n] *
{ /* Ignore single-line comments */ }
\/\*([^*] |\* +[^* / ]) *\*\/ {/* Ignore multi-line comments */}.
{
    fputc(yytext[0], outFile);
}
\n { fputc('\n', outFile); }
% %
    int main(int argc, char **argv)
{
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <input-file>\n", argv[0]);
        exit(1);
    }
    FILE *inFile = fopen(argv[1], "r");
    if (!inFile)
    {
        perror("Error opening input file");
        exit(1);
    }
    outFile = fopen("out.c", "w");
    if (!outFile)
    {
        perror("Error opening output file");
        fclose(inFile);
        exit(1);
    }
    yyin = inFile;
    yylex();
    fclose(inFile);

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

Input:

input.c

compiler_design > input.c

```c
#include <stdio.h>

// This is a single-line comment

/*
 * This is a multi-line comment
 * that spans multiple lines.
 */

int main() {
    printf("Hello, World!\n"); // Print a message to the console

    /*
     * Another multi-line comment
     * inside the main function.
     */

    return 0; // Return 0 to indicate successful execution
}
```

Output:

out.c

compiler_design > out.c

```c
#include <stdio.h>



int main() {
    printf("Hello, World!\n");



    return 0;
}
```

2. Design a LeX code to extract all HTML tags in the given HTML file at run time and store into text file given at run time

Code:

```
 %
{
#include <stdio.h>
#include <string.h>
FILE *output_file;
%
}
```

```
% option noyywrap % %
    "<"[^ > ] * ">" { fprintf(output_file, "%s\n", yytext); }
.|\n{/* Ignore all other characters */} % %
        int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <input_html_file>
<output_text_file>\n", argv[0]);
        return 1;
    }
    FILE *input_file = fopen(argv[1], "r");
    if (!input_file)
    {
        fprintf(stderr, "Error: Cannot open input file %s\n", argv[1]);
        return 1;
    }
    output_file = fopen(argv[2], "w");
    if (!output_file)
    {
        fprintf(stderr, "Error: Cannot open output file %s\n",
argv[2]);
        fclose(input_file);
        return 1;
    }
    yyin = input_file;
    yylex();
    fclose(input_file);
    fclose(output_file);
    return 0;
}
```

Input:

input.html

compiler_design > lab5 > input.html > html > body

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sample HTML for Tag Extraction</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            line-height: 1.6;
            color: #333;
        }
    </style>
</head>
<body>
    <header>
        <h1>Welcome to Our Website</h1>
        <nav>
            <ul>
                <li><a href="#home">Home</a></li>
                <li><a href="#about">About</a></li>
                <li><a href="#contact">Contact</a></li>
            </ul>
        </nav>
    </header>

    <main>
        <section id="home">
            <h2>Home</h2>
```

Output:

input.html | output.txt

compiler_design > lab5 > output.txt

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>
</title>
<style>
</style>
</head>
<body>
<header>
<h1>
</h1>
<nav>
<ul>
<li>
<a href="#home">
</a>
</li>
<li>
<a href="#about">
</a>
</li>
<li>
<a href="#contact">
```