CSM 3202: Compiler Lab

Lab Report 1: Lexical Analysis and Token Classification

Name: MD. TAMIM AHMED FAHIM

ID: 2209024

Level: 3, Semester: 2

Bioinformatics Engineering

Bangladesh Agricultural University

Report submitted: December 2, 2025

This is a documentation to the Lab X done in the Compiler Lab Course. The report is submitted to Md. Saif Uddin, Lecturer, Department of Computer Science and Mathematics, Bangladesh Agricultural University.

Contents

ListofFigures

ListofTables

Listings

1 Introduction

1.1 Background of the Experiment

You will explain what lexical analysis is, why it is used in compiler design, and the overall purpose of token identification.

1.2 Objectives of the Lab

You will list 3–5 clear objectives describing what you aim to learn, such as writing lex programs, identifying tokens, and classifying input streams.

1.3 Tools and Requirements

Y*ou will list the software used (e.g.IDE, C Compiler, Windows/Ubuntu OS, terminal/command prompt).*

2 Tasks and Implementations

2.1 Task 1: Characters, Words, and Lines Count

2.1.1 Task Description

This program reads a source file sequentially and counts the total number of characters, words, and lines. It correctly handles spaces, tabs, and newline characters while scanning the file.

2.1.2 Source Code

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

int main()
{

    FILE *fp = NULL;
    char ch;
    int character = 0;
    int word = 0;
    int line = 0;
    int word_flag = 0;

    fp = fopen("task1.txt", "r");
    if (fp == NULL)
    {
        printf("file␣not␣found");
        exit(1);
    }

    // reading character by character
    while ((ch = fgetc(fp)) != EOF)
    {
        character++;
        if (ch == '\n')
        {
            line++;
        }

        if (isspace(ch))
        {
            word_flag = 1;
        }
        else if (word_flag == 1)
        {
            word_flag = 0;
            word++;
        }

        // printf("%c ",ch);
    }

    word++;

    fclose(fp);
    printf("Characters:␣%d\n", character);
    printf("Words:␣%d\n", word);
    printf("Lines:␣%d\n", line);

    return 0;
}
```

Listing 1: A C program to count characters, words, and lines

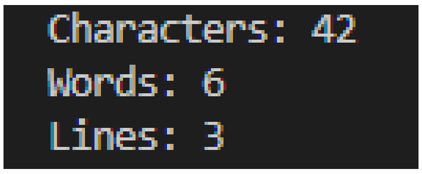
2.1.3 Input & Output

You will write the input sample you provided to the program exactly as executed and insert the screenshot of your output terminal here. Make sure the image is clear and readable.

Input:

```
int main() {
    printf("Hello World");
}

```

Output:



```
Characters: 42
Words: 6
Lines: 3
```

2.1.4 Explanation of Code

The program read each character from file and calculate character, whhen it find ' ' or '\t' or '\n' or other whitespace it count word and when it fine '\n' it count new line.

2.2 Task 2: Identify Keywords

2.2.1 Task Description

This program takes input line of C code, identify and print all the keywords present in the line. Non-keyword identifiers should not be printed.

2.2.2 Source Code

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

int is_keyword(const char *word)
{
    const char *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"};

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

int main()
{

    FILE *fp = NULL;
    char ch;
    char new_word[40];
    int index = 0;

    fp = fopen("task2.txt", "r");
    if (fp == NULL)
    {
        printf("file␣not␣found");
        exit(1);
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        if (isalpha(ch))
        {
            index = 0;
            new_word[index] = ch;
            index++;

            while ((ch = fgetc(fp)) != EOF && isalpha(
               ch))
            {
                new_word[index] = ch;
                index++;
            }

            new_word[index] = '\0';

            // printf("%s@ ", new_word);

            if (is_keyword(new_word))
            {
                printf("%s␣:␣keyword␣\n", new_word);
            }
        }
    }

    fclose(fp);

    return 0;
}
```

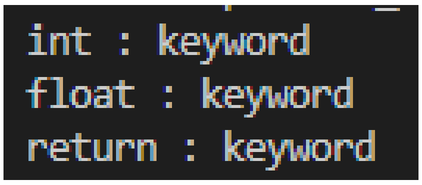
Listing 2: A C program to Identify Keywords

2.2.3 Input & Output

Input:

```c
int main() {
    float num = 2.5;
    return 0;
}
```

Output:



```
int : keyword
float : keyword
return : keyword
```

2.2.4 Explanation of Code

Firstly each word is identified and a function is used to check if that word in keyword array.

2.3 Task 3: Identification of Valid and Invalid Identifiers

2.3.1 Task Description

A program to extract all identifiers from a given line and classify them as valid or invalid.

2.3.2 Source Code

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

int is_identifier(const char *word)
{
    const char *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 (isdigit(word[0]))
    {
        return 0;
    }

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

```

```c
int main()
{

    FILE *fp = NULL;
    char ch;
    char new_word[40];
    int index = 0;

    fp = fopen("task3.txt", "r");
    if (fp == NULL)
    {
        printf("file␣not␣found");
        exit(1);
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        if (isalnum(ch) || ch == '_')
        {
            index = 0;
            new_word[index] = ch;
            index++;

            while ((ch = fgetc(fp)) != EOF && (isalnum
               (ch) || ch == '_'))
            {
                new_word[index] = ch;
                index++;
            }

            new_word[index] = '\0';

            // printf("%s@ ", new_word);

            if (is_identifier(new_word) == 1)
            {
                printf("%s␣:␣Valid␣Indentifier␣\n",
                   new_word);
            }
            else
            {
                    printf("%s␣:␣Invalid␣Indentifier␣\n",
                        new_word);
               }
          }
     }

     fclose(fp);

     return 0;
}
```

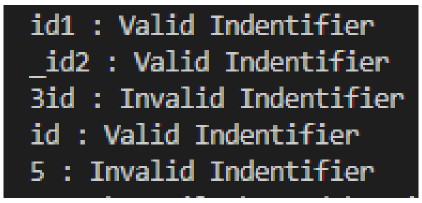
Listing 3: A C program to Identification of Valid and Invalid Identifiers

2.3.3 Input & Output

Input:

```
id1 , _id2 , 3id , id 5
```

Output:



```
id1 : Valid Indentifier
_id2 : Valid Indentifier
3id : Invalid Indentifier
id : Valid Indentifier
5 : Invalid Indentifier
```

2.3.4 Explanation of Code

Firstly identified each word and check with conditions of identifier in a function.

2.4 Task 4: Identification of Numeric Constants

2.4.1 Task Description

A program to detect and classify numeric constants into integer and floating point.

2.4.2 Source Code

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

int is_number(const char *word)
{
    int flag = 0;
    int i = 0;
    while (word[i] != '\0')
    {
        if (word[i] == '.')
        {
            flag++;
        }
        i++;
    }
    if (flag > 1)
    {
        return 0;
    }
    else if (flag == 1)
    {
        return 1;
    }
    else
    {
        return 2;
    }
}

int main()
{

    FILE *fp = NULL;
    char ch;
    char new_word[40];
    int index = 0;

    fp = fopen("task4.txt", "r");
    if (fp == NULL)
    {
        printf("file␣not␣found");
        exit(1);
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        if (isdigit(ch) || ch == '.')
        {
            index = 0;
            new_word[index] = ch;
            index++;

            while ((ch = fgetc(fp)) != EOF && (isdigit
               (ch) || ch == '.'))
            {
                new_word[index] = ch;
                index++;
            }

            new_word[index] = '\0';

            // printf("%s@ ", new_word);

            if (is_number(new_word) == 1)
            {
                printf("%s␣:␣Float␣\n", new_word);
            }
            else if (is_number(new_word) == 2)
            {
                printf("%s␣:␣Integer␣\n", new_word);
            }
            else
```

```
                {
                    printf("%s␣:␣Not␣a␣number␣\n",
                       new_word);
                }
            }
        }

        fclose(fp);

        return 0;
}
```

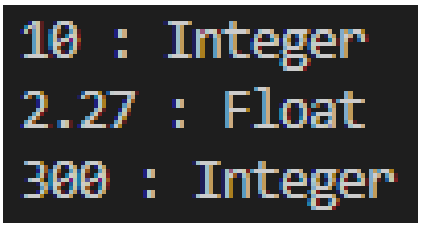
Listing 4: A C program to identification of Numeric Constants

2.4.3 Input & Output

Input:

```
a = 10 ; b = 2.27 ; c = 300 ;
```

Output:



```
10 : Integer
2.27 : Float
300 : Integer
```

2.4.4 Explanation of Code

Firstly all word which are similar like digit is seperated then the are check throuch a user defined function.

2.5 Task 5: Identification of Operators

2.5.1 Task Description

A programm to identify all assignment, arithmetic, relational, and logical operators.

2.5.2 Source Code

```c
#include <stdio.h>

int main()
{
    FILE *fp = fopen("task5.txt", "r");
    if (fp == NULL)
    {
        printf("File␣not␣found\n");
        return 1;
    }

    char ch, next_ch;

    while ((ch = fgetc(fp)) != EOF)
    {
        int next_ch = fgetc(fp);
        if (next_ch != EOF)
        {
            // Check 2-character operators
            if (ch == '+' && next_ch == '+')
            {
                printf("++␣:␣arithmetic␣Operator\n");
                continue;
            }
            else if (ch == '-' && next_ch == '-')
            {
                printf("--␣:␣arithmetic␣Operator\n");
                continue;
            }
            else if (ch == '=' && next_ch == '=')
            {
                printf("==␣:␣relational␣Operator\n");
                continue;
            }
            else if (ch == '!' && next_ch == '=')
            {
                printf("!=␣:␣relational␣Operator\n");
                continue;
            }
            else if (ch == '>' && next_ch == '=')
            {
                printf(">=␣:␣relational␣Operator\n");
                continue;
            }
            else if (ch == '<' && next_ch == '=')
            {
                printf("<=␣:␣relational␣Operator\n");
                continue;
            }
            else if (ch == '&' && next_ch == '&')
            {
                printf("&&␣:␣logical␣Operator\n");
                continue;
            }
            else if (ch == '|' && next_ch == '|')
            {
                printf("||␣:␣logical␣Operator\n");
                continue;
            }
            else if (ch == '+' && next_ch == '=')
            {
                printf("+=␣:␣assignment␣Operator\n");
                continue;
            }
            else if (ch == '-' && next_ch == '=')
            {
                printf("-=␣:␣assignment␣Operator\n");
                continue;
            }
            else if (ch == '*' && next_ch == '=')
            {
                printf("*=␣:␣assignment␣Operator\n");
                continue;
            }
```

```c
            else if (ch == '/' && next_ch == '=')
            {
                printf("/=␣:␣assignment␣Operator\n");
                continue;
            }
            else
            {
                ungetc(next_ch, fp);
            }
        }

        // Single-character operators
        if (ch == '=')
            printf("=␣:␣assignment␣Operator\n");
        else if (ch == '+')
            printf("+␣:␣arithmetic␣Operator\n");
        else if (ch == '-')
            printf("-␣:␣arithmetic␣Operator\n");
        else if (ch == '*')
            printf("*␣:␣arithmetic␣Operator\n");
        else if (ch == '/')
            printf("/␣:␣arithmetic␣Operator\n");
        else if (ch == '%')
            printf("%%␣:␣arithmetic␣Operator\n");
        else if (ch == '>')
            printf(">␣:␣relational␣Operator\n");
        else if (ch == '<')
            printf("<␣:␣relational␣Operator\n");
        else if (ch == '!')
            printf("!␣:␣logical␣Operator\n");
    }

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

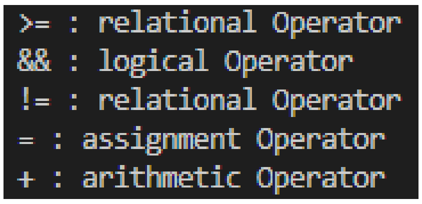
Listing 5: A C program to Identification of Operators

2.5.3 Input & Output

Input:

```
if (a >= 10 && b != 5 ) a = a + 1 ;
```

Output:



```
>= : relational Operator
&& : logical Operator
!= : relational Operator
= : assignment Operator
+ : arithmetic Operator
```

2.5.4 Explanation of Code

Using simply if condition for character match.

2.6 Task 6: Identification of Punctuation and Special Symbols

2.6.1 Task Description

A program to Identification of Punctuation and Special Symbols.

2.6.2 Source Code

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

int main()
{

    FILE *fp = NULL;
    char ch;
    char new_word[40];
    int index = 0;

    fp = fopen("task2.txt", "r");
    if (fp == NULL)
    {
        printf("file not found");
        exit(1);
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        if (ispunct(ch))
        {
            printf("%c : Punctuation Symbol\n", ch);
        }
        else if (!isdigit(ch) && !isalpha(ch) && !
           isspace(ch))
        {
            printf("%c : special Symbol\n", ch);
        }
    }

    fclose(fp);

    return 0;
}
```

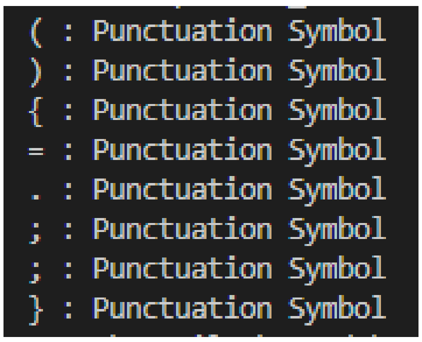
Listing 6: A C program to Identification of Punctuation and Special Symbols

2.6.3 Input & Output

Input:

```c
if(a%2==0) printf("%d is even", a);
```

Output:



```
( : Punctuation Symbol
) : Punctuation Symbol
{ : Punctuation Symbol
= : Punctuation Symbol
. : Punctuation Symbol
; : Punctuation Symbol
; : Punctuation Symbol
} : Punctuation Symbol
```

2.6.4 Explanation of Code

Using buiidin function to Identification of Punctuation and Special Symbols.

2.7 Task 7: Identification of Single-line and Multi-line Comments

2.7.1 Task Description

A program that detects and prints single-line comments (//...) and multi line comments (/...*/). Handle multiple comments in the same file and report unterminated comments as lexical errors.*

2.7.2 Source Code

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

int main()
{

    FILE *fp = NULL;
    char ch, next_ch;

    fp = fopen("task7.txt", "r");
    if (fp == NULL)
    {
        printf("file not found");
        exit(1);
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        if (ch == '/')
        {
            next_ch = fgetc(fp);
            if (next_ch == '/')
            {
                printf("Single-line comments\n");
                while ((ch = fgetc(fp)) != EOF && ch
                   != '\n')
                    ;
            }
            else if (next_ch == '*')
            {
                int flag = 0;
                while ((ch = fgetc(fp)) != EOF)
                {
                    next_ch = fgetc(fp);
                    // printf("%c\n",ch);
                    // printf("%c\n", next_ch);
                    if (ch == '*' && next_ch == '/')
                    {
                        flag = 1;
                        break;
                    }
                    ungetc(next_ch, fp);
                }

                if (flag == 1)
                {
                    printf("Multi-line␣comments\n");
                }
                else
                {
                    printf("Unterminated␣Multi-line␣
                       comments,␣Lexical␣error\n");
                }
            }
        }
    }

    fclose(fp);

    return 0;
}
```

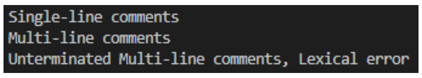
Listing 7: A C program to Identification of Single-line and Multi-line Comments

2.7.3 Input & Output

Input:

```
 // this is comment;
 /* multi line comment
 */
/* this is
```

Output:



```
Single-line comments
Multi-line comments
Unterminated Multi-line comments, Lexical error
```

2.7.4 Explanation of Code

Using a flag variable and loop for comment condition .

2.8 Task 8: Identification of String Literals and Character Constants

2.8.1 Task Description

A program to detect valid and invalid string literals and character constants.

2.8.2 Source Code

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

void check_string(const char *word)
{
    int len = strlen(word);

    if (len >= 2 && word[0] == '"' && word[len-1] == '"')
        printf("%s : Valid String Literal\n", word);
    else if (len >= 2 && word[0] == '"' && word[len-1] != '"')
        printf("%s : InValid String Literal\n", word);
    else if (len >= 1 && word[0] == '"' && word[len-1] != '"')
        printf("%s : Unterminated String Literal\n", word);
    else if (len >= 2 && word[0] == '\'' && word[len-1] == '\'')
        printf("%s : Multiple Character Constant(Invalid)\n", word);
    else if (len = 3 && word[0] == '\'' && word[len-1] == '\'')
    {
        printf("%s : valid Character Constant\n", word);
    }
    else if (len = 2 && word[0] == '\'' && word[len-1] == '\'')
    {
        printf("%s : Empty Character Constant(Invalid)
            \n", word);
    }
    else{
        printf("%s : Unterminated Character Constant(
            Invalid)\n", word);
    }


}

int main()
{
    FILE *fp = fopen("task8.txt", "r");
    if (!fp) {
        printf("file not found");
        return 1;
    }

    char ch;
    char buffer[30];
    int i;

    while ((ch = fgetc(fp)) != EOF)
    {
        if (ch == '"')
        {
            i = 0;
            buffer[i++] = ch;

            while ((ch = fgetc(fp)) != EOF && ch != '\
                n')
            {
                buffer[i++] = ch;
                if (ch == '"')
                    break;
            }

            buffer[i] = '\0';
            ///printf("@%s@\n", buffer);
            check_string(buffer);
        }
    }

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

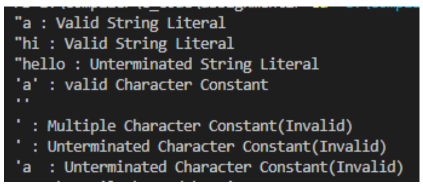
Listing 8: A C program to Identification of String Literals and Character Constants

2.8.3 Input & Output

Input:

```
"a"
"hi"
"hello
'a'
''
'ab'
'a
```

Output:



```
"a : Valid String Literal
"hi : Valid String Literal
"hello : Unterminated String Literal
'a' : valid Character Constant
''
' : Multiple Character Constant(Invalid)
' : Unterminated Character Constant(Invalid)
'a  : Unterminated Character Constant(Invalid)
```

2.8.4 Explanation of Code

Word are seperated then first and last character to detect valid and invalid string literals and character constants

2.9 Task 9: Full Lexical Token Classification and Token Stream Generation

2.9.1 Task Description

A lexical analyzer that scans a file and classifies each lexeme

2.9.2 Source Code

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

char *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"
};
int keyword_count = 34;

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

int is_identifier(const char *word) {
    if (!isalpha(word[0]) && word[0] != '_')
        return 0;

    for (int i = 1; word[i] != '\0'; i++)
        if (!isalnum(word[i]) && word[i] != '_')
            return 0;

    return !is_keyword(word);
}

int is_number(const char *word) {
    int dots = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        if (word[i] == '.') dots++;
        else if (!isdigit(word[i])) return 0;
    }
    return dots <= 1;
}

int main() {
    FILE *fp = fopen("task9.txt", "r");
    if (!fp) {
        printf("File␣not␣found!\n");
        return 1;
    }

    char ch;

    while ((ch = fgetc(fp)) != EOF) {

        /* -----------------------------------------
           IDENTIFIERS & KEYWORDS
        ------------------------------------------ */
        if (isalpha(ch) || ch == '_') {
            char word[100];
            int i = 0;
            word[i++] = ch;

            while ((ch = fgetc(fp)) != EOF && (isalnum
               (ch) || ch == '_'))
                word[i++] = ch;

            word[i] = '\0';
            ungetc(ch, fp);

            if (is_keyword(word))
                printf("%s␣:␣Keyword\n", word);
            else
                printf("%s␣:␣Identifier\n", word);

            continue;
        }

        /* ----------------------------------------
           NUMBERS (INTEGER / FLOAT)
        ----------------------------------------- */
        if (isdigit(ch)) {
            char num[100];
            int i = 0;
            num[i++] = ch;

            while ((ch = fgetc(fp)) != EOF && (isdigit
               (ch) || ch == '.'))
                num[i++] = ch;

            num[i] = '\0';
            ungetc(ch, fp);

            if (is_number(num)) {
                if (strchr(num, '.'))
                    printf("%s␣:␣Float␣Constant\n",
                       num);
                else
                    printf("%s␣:␣Integer␣Constant\n",
                       num);
            } else {
                printf("%s␣:␣Invalid␣Number\n", num);
            }
            continue;
        }

        /* ----------------------------------------
           STRING LITERALS
        ----------------------------------------- */
        if (ch == '"') {
            char buffer[200];
            int i = 0;
            buffer[i++] = ch;

            while ((ch = fgetc(fp)) != EOF && ch != '"') {
                if (ch == '\n') break;
                buffer[i++] = ch;
            }

            buffer[i++] = '"';
            buffer[i] = '\0';

            if (ch == '"')
                printf("%s␣:␣String␣Literal\n", buffer);
            else
                printf("%s␣:␣Unterminated␣String␣Literal\n", buffer);

            continue;
        }

        /* -----------------------------------------
           CHARACTER CONSTANTS
        ----------------------------------------- */
        if (ch == '\'') {
            char buf[10];
            int i = 0;
            buf[i++] = ch;
            buf[i++] = fgetc(fp);
            buf[i++] = fgetc(fp);
            buf[i] = '\0';

            if (buf[2] == '\'')
                printf("%s␣:␣Character␣Constant\n", buf);
            else
                printf("%s␣:␣Invalid␣Character␣Constant\n", buf);

            continue;
        }

        /* ------------------------------------------
           COMMENTS
        ------------------------------------------ */
        if (ch == '/') {
            char next = fgetc(fp);

            if (next == '/') {
                printf("//␣:␣Single-line␣Comment\n");
                while ((ch = fgetc(fp)) != EOF && ch
                   != '\n');
                continue;
            }

            else if (next == '*') {
                printf("/*␣*/␣:␣Multi-line␣Comment\n")
                   ;
                char prev = 0;
                while ((ch = fgetc(fp)) != EOF) {
                    if (prev == '*' && ch == '/')
                       break;
                    prev = ch;
                }
                continue;
            }

            ungetc(next, fp);
        }

        /* ------------------------------------------
           OPERATORS
        ------------------------------------------ */
        char next = fgetc(fp);
        if (next != EOF) {
            char op[3] = {ch, next, '\0'};

            if (!strcmp(op, "==") || !strcmp(op, "!=")
                ||
                !strcmp(op, ">=") || !strcmp(op, "<=")
                   )
            { printf("%s␣:␣Relational␣Operator\n", op)
               ; continue; }
```

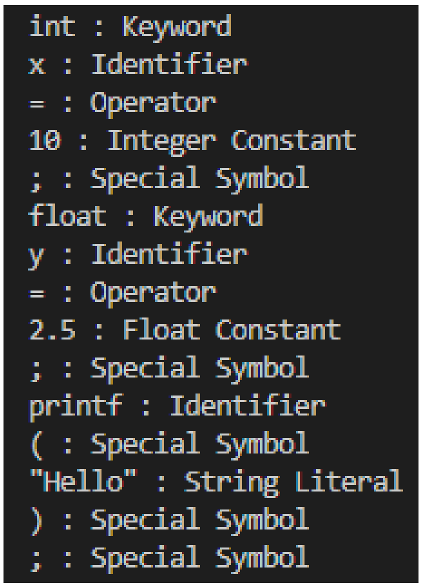
```c

            if (!strcmp(op, "++") || !strcmp(op, "--"))
            { printf("%s␣:␣Arithmetic␣Operator\n", op); continue; }

            if (!strcmp(op, "&&") || !strcmp(op, "||"))
            { printf("%s␣:␣Logical␣Operator\n", op); continue; }

            if (!strcmp(op, "+=") || !strcmp(op, "-=") ||
                !strcmp(op, "*=") || !strcmp(op, "/="))
            { printf("%s␣:␣Assignment␣Operator\n", op); continue; }

            ungetc(next, fp);
        }

        /* Single-character operators */
        if (strchr("=+-*/%><!&|", ch)) {
            printf("%c␣:␣Operator\n", ch);
            continue;
        }

        /* -----------------------------------------
           SPECIAL SYMBOLS & PUNCTUATIONS
        ------------------------------------------ */
        if (ispunct(ch)) {
            printf("%c␣:␣Special␣Symbol\n", ch);
        }
    }

    fclose(fp);
    return 0;
}
```
Listing 9: A C program for Full Lexical Token Classification and Token Stream Generation

2.9.3 Input & Output

Input:

```
int x = 10 ;
float y = 2.5 ;
printf("Hello") ;
```

Output:



```
int : Keyword
x : Identifier
= : Operator
10 : Integer Constant
; : Special Symbol
float : Keyword
y : Identifier
= : Operator
2.5 : Float Constant
; : Special Symbol
printf : Identifier
( : Special Symbol
"Hello" : String Literal
) : Special Symbol
; : Special Symbol
```

2.9.4 Explanation of Code

Combind previous problem code methods.

2.10 Task 10: Detection of Lexical Errors

2.10.1 Task Description

A program that detects and reports common lexical errors: • Invalid identifiers: 2sum, a-bc, @total • Unterminated strings: ”hello • Unclosed comments: / comment • Invalid characters: @, (in identifiers)*

2.10.2 Source Code

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

int is_keyword(const char *word) {
    char *keywords[] = {
        "int","float","char","double","long","short","
           void","return",
        "if","else","while","for","break","continue","
           struct","union"
    };
    int n = sizeof(keywords) / sizeof(keywords[0]);

    for (int i = 0; i < n; i++) {
        if (strcmp(word, keywords[i]) == 0)
            return 1;
    }
    return 0;
}

int is_valid_identifier(const char *word) {
    if (!isalpha(word[0]) && word[0] != '_')
        return 0;

    for (int i = 1; word[i] != '\0'; i++)
        if (!isalnum(word[i]) && word[i] != '_')
            return 0;

    return 1;
}

int main() {
    FILE *fp = fopen("task10.txt", "r");
    if (!fp) {
        printf("Cannot␣open␣file.\n");
        return 1;
    }

    char ch;

    while ((ch = fgetc(fp)) != EOF) {

        /* =========================================
           INVALID IDENTIFIERS (start with digit, contain symbols)
        ========================================= */
        if (isalnum(ch) || ch == '_' || ch == '@' || ch == '$' || ch == '#') {

            char word[100];
            int i = 0;
            word[i++] = ch;

            while ((ch = fgetc(fp)) != EOF &&
                   (isalnum(ch) || ch == '_' || ch == '@' || ch == '$' || ch == '#'))
            {
                word[i++] = ch;
            }

            word[i] = '\0';
            ungetc(ch, fp);

            int invalid_char = 0;
            for (int j = 0; word[j]; j++) {
                if (word[j] == '@' || word[j] == '$' || word[j] == '#')
                    invalid_char = 1;
            }

            if (isdigit(word[0]) || invalid_char || !is_valid_identifier(word)) {
                printf("Error:␣Invalid␣identifier␣'%s
                   '\n", word);
            }

            continue;
        }

        /* =========================================
           UNTERMINATED STRING
        ========================================== */
        if (ch == '"') {
            char buffer[200];
            int i = 0;
            buffer[i++] = ch;
            int closed = 0;

            while ((ch = fgetc(fp)) != EOF) {
                if (ch == '"') {
                    closed = 1;
                    buffer[i++] = ch;
                    break;
                }
                if (ch == '\n')
                    break;

                buffer[i++] = ch;
            }

            buffer[i] = '\0';

            if (!closed)
                printf("Error:␣Unterminated␣string␣
                   literal␣%s\n", buffer);

            continue;
        }

        /* =========================================
           UNCLOSED COMMENTS  /*  comment ...
        ========================================== */
        if (ch == '/') {
```

```c
            char next = fgetc(fp);

            if (next == '*') {
                int closed = 0;
                char prev = 0;

                while ((ch = fgetc(fp)) != EOF) {
                    if (prev == '*' && ch == '/') {
                        closed = 1;
                        break;
                    }
                    prev = ch;
                }

                if (!closed)
                    printf("Error:␣Unclosed␣comment␣
                       '/*'\n");

                continue;
            }

            ungetc(next, fp);
        }

        /* =========================================
           INVALID STANDALONE CHARACTERS
        ========================================== */
        if (ch == '@' || ch == '$' || ch == '#') {
            printf("Error:␣Invalid␣character␣'%c'\n",
               ch);
        }
    }

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

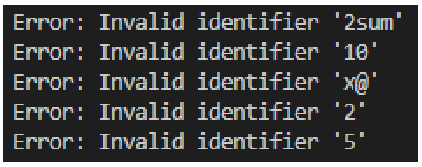
Listing 10: A C program to Detection of Lexical Errors

2.10.3 Input & Output

Input:

```
int 2sum = 10 ;
char *s = "Hello" ;
float x@ = 2.5;
```

Output:



```
Error: Invalid identifier '2sum'
Error: Invalid identifier '10'
Error: Invalid identifier 'x@'
Error: Invalid identifier '2'
Error: Invalid identifier '5'
```

2.10.4 Explanation of Code

Using previous code methods.

3 Conclusion

Lexical analysis is an essential phase of the compiler where the source code is scanned and broken into meaningful units called tokens. It acts as a bridge between raw program text and the syntax analyzer, ensuring that the input program is structured, valid, and easy to process in later stages. In this lab, I implemented basic lexical analysis tasks such as identifying keywords, operators, special symbols, and comments. I also detected single-line and multi-line comments and handled error cases like unterminated comment blocks. Through these tasks, I gained practical understanding of how a lexical analyzer processes characters, builds tokens, and performs error detection before syntax analysis begins.