BANGLADESH UNIVERSITY OF PROFESSIONALS

FACULTY OF SCIENCE & TECHNOLOGY

DEPT. OF COMPUTER SCIENCE & ENGINEERING (CSE)

Lab Report
NO - 02

COURSE NAME: Compiler Laboratory

COURSE CODE: CSE-3104

TITLE: Implementing a Simple Tokenizer Using Flex

SUBMITTED BY:

NAME : Israt Tabassum Kochy
ROLL : 2252421055
SECTION : A
LEVEL/TERM : 3rd

SEMESTER :1st

DATE OF SUBMISSION: November 23, 2024

Experiment No: 02

Title: Implementing a Simple Tokenizer Using Flex

Introduction: Tokenization is the process of dividing a sequence of input characters into meaningful units known as tokens. These tokens represent various elements such as keywords, identifiers, operators, literals, comments, and whitespaces. Tokenization is a fundamental step toward compilation or interpretation, as it enables code to be systematically analyzed. In our lab task, we implemented a tokenizer with specific categories of identification: keywords (e.g., int, float, if, else), operators (arithmetic, logical, relational), identifiers (variable or function names), literals (strings, numeric values, or characters), and comments (single-line or multi-line). Additionally, we integrated error-handling mechanisms to detect and report invalid tokens.

To achieve this, we used Flex (Fast Lexical Analyzer Generator) to design our tokenizer. Patterns were defined for each token type, and corresponding rules were implemented to classify the matched inputs. By combining pattern recognition with basic conditions, we developed a functional tokenizer. The tokenizer operates on file input and output, converting lines of code into categorized tokens. This foundational tool sets the stage for more complex language processing tasks, such as parsing and syntax analysis, in programming language development.

Objective:

- To learn about tokenization and its representation
- To differentiate between various character patterns based on structure
- To implement a tokenizer parallel to the actual tokenization process

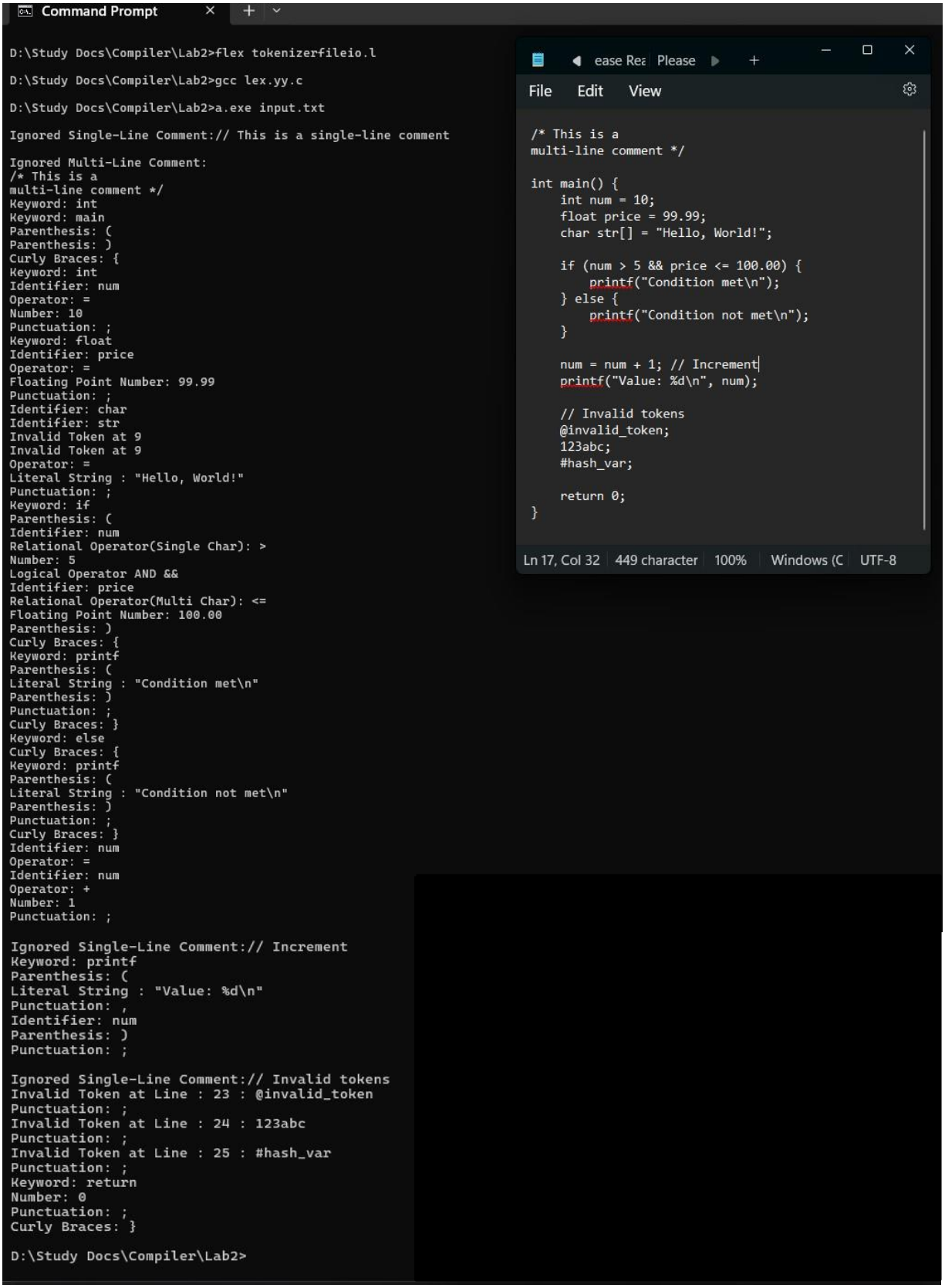
Code:

```
%{

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

%}

%%


"if"|"else" {printf("Keyword: %s\n",yytext);}
"printf"|"scanf" {printf("Keyword: %s\n",yytext);}
"int"|"main"|"return" {printf("Keyword: %s\n",yytext);}


["][^"]*["] {printf("Literal String : %s\n",yytext);}
[0-9]+[.][0-9]* {printf("Floating Point Number: %s\n",yytext);}
[0-9]+ {printf("Number: %s\n",yytext);}
[a-zA-Z_][0-9a-zA-Z_]* {printf("Identifier: %s\n",yytext);}
[<>=!][=] {printf("Relational Operator(Multi Char): %s\n",yytext);}


[&][&] {printf("Logical Operator AND %s\n",yytext);}
[|][|] {printf("Logical Operator OR %s\n",yytext);}


[<>] {printf("Relational Operator(Single Char): %s\n",yytext);}
[+\-*/=?] {printf("Operator: %s\n",yytext);}
[ \t] {} //Ignored Whitespace


[/][/][^\n]* {printf("\nIgnored Single-Line Comment:%s\n",yytext);line_cnt++;}
[/][*][a-zA-Z0-9@#$%^&*(){},;!_+-_ \t\n]*[*][/] {printf("\nIgnored Multi-Line
Comment:\n%s\n",yytext);}


[\(\)] {printf("Parenthesis: %s\n",yytext);}
[\{\}] {printf("Curly Braces: %s\n",yytext);}
[\,\;] {printf("Punctuation: %s\n",yytext);}
[\n] {line_cnt++;}

[0-9]+[a-zA-Z_]+ {printf("Invalid Token at Line : %d : %s\n", line_cnt,yytext);}
```

```
    [#@!][a-zA-Z_]+ {printf("Invalid Token at Line : %d : %s\n",
    line_cnt,yytext);} . {printf("Invalid Token at %d\n", line_cnt);}

    %%

    int yywrap(void){}
    int main(int argc, char
    **argv){
    if (argc>1)
    {
     FILE *file = fopen(argv[1], "r");
    if(!file){
         perror("fopen");
    return EXIT_FAILURE;
    }
 yyin=file;

}
 yylex();
 return 0;

}
```

Output:

For this program, we opted to use a file-based input method instead of a command-line interface (CLI). Tokenizing directly from the CLI can be inefficient and prone to errors, as manually typing a large set of diverse tokens increases the risk of missing critical elements. To address this, we implemented a file I/O approach. A sample code intended for tokenization was written into a file named input.txt. Using the argc and argv commands, we specified the file name as input, which was then processed for tokenization. Despite the simplicity of the code, the presence of numerous tokens resulted in extensive output, with each token being printed on a new line.


Command Prompt
D:\Study Docs\Compiler\Lab2>flex tokenizerfileio.l
D:\Study Docs\Compiler\Lab2>gcc lex.yy.c
D:\Study Docs\Compiler\Lab2>a.exe input.txt
Ignored Single-Line Comment:// This is a single-line comment
Ignored Multi-Line Comment:
/* This is a
multi-line comment */
Keyword: int
Keyword: main
Parenthesis: (
Parenthesis:)
Curly Braces: {
Keyword: int
Identifier: num
Operator: =
Number: 10
Punctuation: ;
Keyword: float
Identifier: price
Operator: =
Floating Point Number: 99.99
Punctuation: ;
Identifier: char
Identifier: str
Invalid Token at 9
Invalid Token at 9
Operator: =
Literal String : "Hello, World!"
Punctuation: ;
Keyword: if
Parenthesis: (
Identifier: num
Relational Operator(Single Char): >
Number: 5
Logical Operator AND &&
Identifier: price
Relational Operator(Multi Char): <=
Floating Point Number: 100.00
Parenthesis:)
Curly Braces: {
Keyword: printf
Parenthesis: (
Literal String : "Condition met\n"
Parenthesis:)
Punctuation: ;
Curly Braces: }
Keyword: else
Curly Braces: {
Keyword: printf
Parenthesis: (
Literal String : "Condition not met\n"
Parenthesis:)
Punctuation: ;
Curly Braces: }
Identifier: num
Operator: =
Identifier: num
Operator: +
Number: 1
Punctuation: ;
Ignored Single-Line Comment:// Increment
Keyword: printf
Parenthesis: (
Literal String : "Value: %d\n"
Punctuation: ,
Identifier: num
Parenthesis:)
Punctuation: ;
Ignored Single-Line Comment:// Invalid tokens
Invalid Token at Line : 23 : @invalid_token
Punctuation: ;
Invalid Token at Line : 24 : 123abc
Punctuation: ;
Invalid Token at Line : 25 : #hash_var
Punctuation: ;
Keyword: return
Number: 0
Punctuation: ;
Curly Braces: }
D:\Study Docs\Compiler\Lab2>
ease Rea Please
File Edit View
/* This is a
multi-line comment */
int main() {
int num = 10;
float price = 99.99;
char str[] = "Hello, World!";
if (num > 5 && price <= 100.00) {
printf("Condition met\n");
} else {
printf("Condition not met\n");
}
num = num + 1; // Increment
printf("Value: %d\n", num);
// Invalid tokens
@invalid_token;
123abc;
#hash_var;
return 0;
}
Ln 17, Col 32
449 character
100%
Windows (C
UTF-8

Discussion:

In this experiment, we explored the process of tokenization and the various types of tokens it involves. We implemented a basic tokenizer capable of handling keywords, identifiers, and numbers (both integers and decimals). Single and multi-line comments are ignored but logged with an appropriate message. We also included rules to identify single and multi-character operators, both logical and relational. String literals enclosed in double quotes are detected and printed unless part of a comment.

For error handling, we maintained a global line counter that increments explicitly with each newline, helping pinpoint the exact line where invalid tokens are encountered. Handling string literals and comments posed challenges due to their potentially infinite character pattern variations. String literals were terminated by double quotes ("), single-line comments by newline characters, and multi-line comments by the */ syntax. To accommodate these variations, we utilized the "negated character class" (^ operator) in our patterns. For example, [^\"] ensures all characters are treated as part of the string until a double quote is encountered, at which point the buffer stops collecting input. Similarly, [^\n] processes characters until a newline is detected.

While multi-character pattern matching can become complex, we focused on basic patterns for simplicity. This structured approach allowed us to successfully develop a functional tokenizer.