Bachelor of Science in Computer Science & Engineering

TokenTrove: Exploring the Lexical Landscape of C Code with Flex

by

Salman Farsi

ID: 1804102

Department of Computer Science & Engineering

Chittagong University of Engineering & Technology (CUET)

Chattogram-4349, Bangladesh.

10th August, 2023

Chittagong University of Engineering & Technology (CUET)

Department of Computer Science & Engineering

Chattogram-4349, Bangladesh.

Project Report

Compiler Design Sessional (CSE - 432)

Student Name : Salman Farsi Session : 2021-2022
ID : 1804102

Submitted To : Dr. Muhammad Moshiul Hoque
Professor
Department of Computer Science & Engineering

Ashim Dey
Assistant Professor
Department of Computer Science & Engineering

Avishek Das
Lecturer
Department of Computer Science & Engineering

Department : Computer Science & Engineering
Program : B.Sc. Engineering

Tentative Title : TokenTrove: Exploring the Lexical Landscape of C Code with Flex

Table of Contents

List of Code Snippets

1 Introduction

Lexical analysis stands as a foundational pillar in the world of compiler design. This initial stage serves to convert human-readable source code into a machine-comprehensible form, facilitating seamless processing by compilers. Operating within this realm, lexical analysis dissects C source code into constituent tokens, encompassing keywords, operators, and identifiers. The artistry of lexical analysis lies in the transformation of human expression into a coherent language understood by compilers. This process is foundational to the realm of software engineering, bridging the gap between human intent and machine execution.

The core purpose of lexical analysis is to tokenize source code, effectively segmenting it into meaningful units. These tokens encapsulate distinct elements like keywords, which hold predefined meanings in the programming language; operators, which govern actions on operands; and identifiers, signifying user-defined names for variables and functions. Each token category represents a semantic thread interwoven into the fabric of the code.

The dynamic landscape of programming and software development continually evolves, demanding efficient and accurate tools for code analysis and comprehension. In this context, the significance of lexical analysis becomes evident. With the ever-expanding realm of programming languages and the intricate structures they encompass, the need to swiftly and accurately decipher human-intended instructions into machine-understandable formats is paramount. The motivation to delve into lexical analysis is fueled by the pursuit of bridging this communication gap between human programmers and the complex world of compilers. As we endeavor to streamline the process of code transformation and empower software developers to convey their intentions seamlessly, the exploration of lexical analysis takes center stage, laying the foundation for enhanced programming experiences and robust software creation.

This project focuses on performing a lexical analysis of C source code, aiming to categorize various elements such as keywords, operators, identifiers, numbers, and strings. Lexical analysis is a fundamental step in the compilation process,

enabling the identification of different tokens that constitute the source code. The project employs the lexical analyzer tool Flex to tokenize the input source code and organize the resulting tokens into distinct categories.

2 Specific Objectives

The objectives of this project are as follows:

1. To develop a lexer using Flex to recognize various tokens in source code.
2. To categorize tokens into keywords, operators, numbers, identifiers, etc.
3. To count occurrences of each token category and display results.

3 Flex Source Code

Code 1: Flexify!

```
%{
#include<stdio.h>
#include<string.h>
#define MAX_KEYWORDS 50
#define MAX_REL_OP 50
#define MAX_ARITH_OP 50
#define MAX_ASSIGN_OP 50
#define MAX_LOGICAL_OP 50
#define MAX_NUMBERS 50
#define MAX_IDENTIFIERS 50
#define MAX_FUNCTIONS 50
#define MAX_OTHER_SYMBOLS 50
#define MAX_STRINGS 50
#define MAX_BLANK 1000

char keywords[MAX_KEYWORDS][50];
char relOperators[MAX_REL_OP][10];
char arithOperators[MAX_ARITH_OP][10];
char assignOperators[MAX_ASSIGN_OP][10];
char logicalOperators[MAX_LOGICAL_OP][10];
char numbers[MAX_NUMBERS][50];
char identifiers[MAX_IDENTIFIERS][50];
```

```
23 char functions[MAX_FUNCTIONS][50];
24 char otherSymbols[MAX_OTHER_SYMBOLS][10];
25 char strings[MAX_STRINGS][50];
26 char Blank[MAX_BLANK][50];
27
28 int numKeywords = 0;
29 int numRelOp = 0;
30 int numArithOp = 0;
31 int numAssignOp = 0;
32 int numLogicalOp = 0;
33 int numNumbers = 0;
34 int numIdentifiers = 0;
35 int numFunctions = 0;
36 int numOtherSymbols = 0;
37 int numStrings = 0;
38 int numBlank = 0;
39 %}
40
41 %%
42 "if"|"then"|"also"|"int"|"char"|"main"|"return"|"printf"|"include
      "|"stdio.h"|
43 "else"|"while"|"for"|"do"|"switch"|"case"|"default"|"break"|"
      continue"|"string.h"|
44 "short"|"long"|"float"|"double"|"void"|"signed"|"unsigned"|"
      struct"|"union"|"enum"|"typedef"|
45 "const"|"volatile"|"extern"|"static"|"register"|"auto"|"goto"|"
      sizeof"|"define"
46
47 {strcpy(keywords[numKeywords++], yytext);}
48
49 ">"|"<"|">="|"<="|"=="|"!="
       { strcpy(relOperators[numRelOp++], yytext); }
50
51 "+"|"-"|"*"|"/"|"++"|"--"
       { strcpy(arithOperators[numArithOp++], yytext); }
52
53 "="|"+="|"*="|"-="|"/="|"%="
        { strcpy(assignOperators[numAssignOp++], yytext); }
54
```

```
"&&"|"\|\|"|"!"
     { strcpy(logicalOperators[numLogicalOp++], yytext); }

[0-9]+(\.[0-9]+)?
     { strcpy(numbers[numNumbers++], yytext); }

[a-zA-Z_][a-zA-Z0-9_]*
     { strcpy(identifiers[numIdentifiers++], yytext); }

[a-zA-Z_][a-zA-Z0-9_]*"()"
     { strcpy(functions[numFunctions++], yytext); }

[{}[\]();,]
     { strcpy(otherSymbols[numOtherSymbols++], yytext); }

\"([^\\\"]|\\.)*\"
     { strcpy(strings[numStrings++], yytext); }

[\t\n\v\f\r]
    { strcpy(Blank[numBlank++], yytext); }

.   { /* ignore any other characters */ }
%%
int yywrap()
{
}
void PrintToken(const char *categoryName, char tokens[][50], int
   numTokens)
{
    printf("-----------------------------------------------\n");
    printf("%-*s\t:\toccurrence\n", 25, categoryName);
    printf("-----------------------------------------------\n");

    for (int i = 0; i < numTokens; i++) {
        int isNewToken = 1;
        for (int j = 0; j < i; j++) {
            if (strcmp(tokens[i], tokens[j]) == 0) {
                isNewToken = 0;
                break;
```

```
            }
        }

        if (isNewToken) {
            int count = 0;
            for (int j = 0; j < numTokens; j++) {
                if (strcmp(tokens[i], tokens[j]) == 0) {
                    count++;
                }
            }

            printf("%-*s\t:\t%d\n", 25, tokens[i], count);
        }
    }

    printf("\n");
}
int main()
{
    yylex();

    PrintToken("Keywords", keywords, numKeywords);
    PrintToken("Relational Operators", relOperators, numRelOp);
    PrintToken("Arithmetic Operators",arithOperators,numArithOp);
    PrintToken("Assignment Operators", assignOperators,
   numAssignOp);
    PrintToken("Logical Operators", logicalOperators,
   numLogicalOp);
    PrintToken("Valid Numbers", numbers, numNumbers);
    PrintToken("Valid Identifiers", identifiers, numIdentifiers);
    PrintToken("Functions", functions, numFunctions);
    PrintToken("Other Symbols", otherSymbols, numOtherSymbols);
    PrintToken("Strings", strings, numStrings);

    return 0;
}
```

4 Methodology

The presented methodology in this section delves into the creation of a lexical analyzer using Flex. The process is orchestrated through a sequence of regular expressions that correspond to distinct token categories as follows:

4.1 Regular Expressions for Token Identification

This subsection involves employing specific regular expressions to identify and categorize various tokens within the C source code. Each regular expression is tailored to match a specific token category and is associated with corresponding code for efficient processing.

4.1.1 Keywords

A comprehensive list of C keywords is defined using regular expressions. These expressions match exact keyword strings and store them for analysis.

```
"if"|"then"|"also"|"int"|"char"|"main"|"return"|"printf"|"include
   "|"stdio.h"|"else"|"while"|"for"|"do"|"switch"|"case"|"default
   "|"break"|"continue"|"string.h"|"short"|"long"|"float"|"double
   "|"void"|"signed"|"unsigned"|"struct"|"union"|"enum"|"typedef"
   |"const"|"volatile"|"extern"|"static"|"register"|"auto"|"goto"
   |"sizeof"|"define" {strcpy(keywords[numKeywords++], yytext);}
```

4.1.2 Relational, Arithmetic, Assignment, and Logical Operators

Separate regular expressions are employed to identify various types of operators and store them for later analysis.

```
">"|"<"|">="|"<="|"=="|"!="
{ strcpy(relOperators[numRelOp++], yytext); }
"+"|"-"|"*"|"/"|"++"|"--"
{ strcpy(arithOperators[numArithOp++], yytext); }
"="|"+="|"*="|"-="|"/="|"%="
{ strcpy(assignOperators[numAssignOp++], yytext); }
"&&"|"\|\|"|"!"
{ strcpy(logicalOperators[numLogicalOp++], yytext); }
```

4.1.3 Numbers and Identifiers

Regular expressions are employed to recognize numerical values and identifiers adhering to C's naming conventions.

```
[0-9]+(\.[0-9]+)?
{ strcpy(numbers[numNumbers++], yytext); }
```

This regular expression is used to identify numerical values in the C source code. Here's how it works:

- [0-9]+: This part of the expression matches one or more digits (0 to 9). It indicates that the token should start with at least one digit.
- `(\.[0-9]+)?`: This part is an optional group that matches a decimal point followed by one or more digits. The ? makes the entire group optional, so it will match numbers without a decimal part as well.

```
[a-zA-Z_][a-zA-Z0-9_]*
strcpy(identifiers[numIdentifiers++], yytext); }
```

This regular expression identifies valid C identifiers, which are used for variable names, function names, and more. Here's the breakdown:

- [a-zA-Z_]: This part matches a single character that is either an uppercase letter (A to Z), a lowercase letter (a to z), or an underscore (_). This ensures that the identifier starts with a letter or an underscore.
- [a-zA-Z0-9_]*: This part matches zero or more characters that are either uppercase or lowercase letters, digits (0 to 9), or underscores. This allows for the inclusion of letters, digits, and underscores in the identifier after the initial character.

4.1.4 Functions

Regular expressions are tailored to recognize C functions, identified by their parentheses following an identifier.

```
[a-zA-Z_][a-zA-Z0-9_]*"()"
{ strcpy(functions[numFunctions++], yytext); }
```

- [a-zA-Z_]: This part of the expression matches a single character that is either an uppercase letter (A to Z), a lowercase letter (a to z), or an underscore (_). This ensures that the function name starts with a letter or an underscore.
- [a-zA-Z0-9_]*: This part matches zero or more characters that are either uppercase or lowercase letters, digits (0 to 9), or underscores. This allows for the inclusion of letters, digits, and underscores in the function name after the initial character.
- "()": This part matches a literal pair of parentheses ().

In combination, this regular expression ensures that it identifies function names that start with a valid character and can include letters, digits, and underscores. It specifically looks for function names followed by an empty set of parentheses, which is a common pattern in C when calling functions without any arguments.

4.1.5 Other Symbols

Special symbols and punctuation marks are identified using dedicated regular expressions.

```
1 [{}[\](),]
2 strcpy(otherSymbols[numOtherSymbols++], yytext); }
```

- [{}]: This part matches either an opening curly brace or a closing curly brace . Curly braces are used to define code blocks and scope in C.
- `[\](),]`: This part matches any of the following characters:], (,), ;, and ,. These characters are common punctuation symbols and delimiters used in C programming for various purposes, such as indexing arrays, calling functions, terminating statements, and separating items in lists.

4.1.6 Strings

Strings within double quotes are detected using regular expressions to account for various escape characters.

```
1 \"([^\\\"]|\\.)*\"
2 { strcpy(strings[numStrings++], yytext); }
```

- \": This part matches the opening double quote " character. It indicates the start of a string literal.
- And everything inside the quotation until the closing quotation comes, take as the string.

4.1.7 Blank Spaces and Newlines

Regular expressions identify whitespace characters and newline sequences.

```
[\t\n\v\f\r]  { strcpy(Blank[numBlank++], yytext); }
```

- \t: This represents a tab character.
- \n: This represents a newline character.
- \v: This represents a vertical tab character.
- \f: This represents a form feed character.
- \r: This represents a carriage return character.

4.2 Flex Processing Flow

1. The provided source code is input to Flex, which scans the code sequentially, character by character.
2. As the source code is read, Flex identifies the regular expression patterns to categorize tokens.
3. Upon recognizing a token, Flex assigns it to its appropriate category and stores it for further analysis.
4. The analyzer then proceeds to the next token, repeating the process until the entire source code is processed.

4.3 Output Analysis and Presentation

Following the token identification phase, the program generates statistics regarding the occurrence of each token category. Utilizing C arrays, the frequency of each token type is tabulated and presented in an organized format. This aids

in visualizing the distribution of keywords, operators, identifiers, functions, and other symbols within the given source code.

4.4 Supporting Functions and Flex Utility

To facilitate the token identification process, Flex offers the yytext variable, which holds the current matched token's value. This value is copied to respective arrays to categorize and count occurrences efficiently. The flexibility of Flex allows for the creation of a comprehensive lexical analysis tool, providing developers with invaluable insights into their code's composition.

5 Results

The inputs for the implementation is taken from a input file shown in the code 2 and the generated output is also saved in another file as shown in the subsection 5.2.

5.1 Input

Code 2: Input C Code

```c
#include<stdio.h>
#include<string.h>
int main() {
    printf("%s %s %s %s %s %s %s\n", if, then, also, int, char,
   main, return);
    int x = 5, y = 10;
    if (x > y)
    {
        printf("x greater than y");
    }
    else
    {
        printf("x less than y");
    }
    int a = 5, b = 3;
    int sum = a + b;
    int product = a * b;
```

```c

    printf("%d\n", sum);
    printf("%d\n", product);

    int num = 10;
    num += 5;
    num -= 3;
    num *= 2;
    num /= 4;

    printf("%d\n", num);

    int condition1 = 1;
    int condition2 = 0;

    if (condition1 && condition2) {
        printf("Bangldesh");
    } else if (condition1 || condition2) {
        printf("CUET");
    } else {
        printf("Pahartoli");
    }

    float val_ = 1.5, val__ = 2.5;

    int my123Name = myFunc();
    printf("%d\n", my123Name);

    char text[] = "Go for Good";

    return 0;
}

int myFunc() {
    return 10 % 5;
}
```

5.2 Output

```
-------------------------------------------------------
Keywords                :       occurence
-------------------------------------------------------
include                 :       2
stdio.h                 :       1
string.h                :       1
int                     :       11
printf                  :       10
if                      :       4
then                    :       1
also                    :       1
char                    :       2
main                    :       1
return                  :       3
else                    :       3
float                   :       1

-------------------------------------------------------
Relational Operators    :       occurence
-------------------------------------------------------
<                       :       2
>                       :       3

-------------------------------------------------------
Arithmetic Operators    :       occurence
-------------------------------------------------------
+                       :       1
*                       :       1

-------------------------------------------------------
Assignment Operators    :       occurence
```

```
-----------------------------------------------------------
=                       :       13
+=                      :       1
-=                      :       1
*=                      :       1
/=                      :       1

-----------------------------------------------------------
Logical Operators       :       occurence
-----------------------------------------------------------
&&                      :       1
||                      :       1

-----------------------------------------------------------
Valid Numbers           :       occurence
-----------------------------------------------------------
5                       :       4
10                      :       3
3                       :       2
2                       :       1
4                       :       1
1                       :       1
0                       :       2
1.5                     :       1
2.5                     :       1

-----------------------------------------------------------
Valid Identifiers       :       occurence
-----------------------------------------------------------
x                       :       2
y                       :       2
a                       :       3
```

```
b                       :       3
sum                     :       2
product                 :       2
num                     :       6
condition1              :       3
condition2              :       3
val_                    :       1
val__                   :       1
my123Name               :       2
text                    :       1

-------------------------------------------------------------
Functions               :       occurence
-------------------------------------------------------------
main()                  :       1
myFunc()                :       2

-------------------------------------------------------------
Other Symbols           :       occurence
-------------------------------------------------------------
{                       :       7
(                       :       13
,                       :       14
)                       :       13
;                       :       26
}                       :       7
[                       :       1
]                       :       1

-------------------------------------------------------------
Strings                         :       occurence
-------------------------------------------------------------
```

```
"%s %s %s %s %s %s %s\n"        :       1
"x greater than y"              :       1
"x less than y"                 :       1
"%d\n"                          :       4
"Bangldesh"                     :       1
"CUET"                          :       1
"Pahartoli"                     :       1
"Go for Good"                   :       1
```

6 Required Resources

6.1 Hardware Resources

- Personal Computer

6.2 Software Resources

- OS: Windows 10
- Tools: Flex, Sublime Text
- Compiler : GNU GCC
- Programming Language: C

7 Conclusion

The implemented methodology showcases the power of lexical analysis in dissecting complex C source code. By employing regular expressions and Flex's capabilities, the program accurately identifies and categorizes tokens, enabling efficient code analysis. This approach may enable developers to gain deeper insights into the distribution and utilization of keywords, operators, identifiers, and more, thereby contributing to enhanced code understanding and software quality.