# 

**MINI PROJECT**

**ON**

**LEXICAL ANALYZER**

**Submitted by:**

**Candidate Name: SHREYA KORI**

**Candidate ID: 104891**

# 

# 

# TABLE OF CONTENTS

|  |  |
| --- | --- |
| CONTENTS | Pg. No |
| Title Page | 1 |
| Table of Contents | 2 |
| List of Figures | 3 |
| Chapter 1: Introduction  1.1 Problem Statement  1.2 Project Description | 4 |
| Chapter 2: Requirements  2.1 Software Used  2.2 Functional Requirements  2.3 Working | 5  6 |
| Chapter 3: Test Plan  3.1 Implementation Details  3.2 Sample Specification  3.3 Actions | 7  8 |
| Chapter 4: Test cases and Expected Results | 9 |
| Chapter 5: Application and Future works | 10 |
| Chapter 6: References | 10 |

# 

# 

# LIST OF FIGURES

|  |  |
| --- | --- |
| FIGURES | Pg. No. |
| Fig. 1: Illustrative Model of Lexical Analyzer | 5 |
| Fig. 2: Data Flow Diagram for Lexical Analyzer | 6 |
| Fig. 3: Block Diagram of Lexical Analyzer | 7 |
| Fig. 4: Flowchart of Lexical Analyzer | 8 |

**CHAPTER 1:**

# INTRODUCTION:

## **1.1 Problem Statement:**

To design a lexical analyzer using C language. This lexical analyzer ignores redundant spaces, tabs and newlines.

## **1.2 Project Description:**

Lexical Analyzer is the first phase of a compiler. Lexical Analysis is the process of taking an input string of characters such as the source code of a computer program and producing a sequence of symbols called lexical tokens, which may be handled more easily by a parser. A program or function which performs lexical analysis is called a lexical analyzer, lexer, or scanner. A lexer often exists as a single function which is called by a parser or another function. Here a lexical analyzer is designed to recognize identifiers, operators, keywords as well as integers that are written as a part of the source program.

A token is a string of characters, categorized according to certain rules into different symbol types. (e.g. Identifier, Operator). Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. The process of forming tokens from an input stream of characters is called tokenization.

In this project the lexical analyzer uses C language, reads the input values and further performs the following tasks:

1. Reads the input characters from the source program.
2. Identifies the tokens according to the types defined using functions after dividing the source code.
3. Eliminates white spaces in the form of blanks, tab and newline characters.
4. Correlates error messages/invalid identifiers if any.

**CHAPTER 2:**

# REQUIREMENTS

## **2.1 Software Used:**

Code Blocks IDE version 20.03

## **2.2 Functional Requirements**:

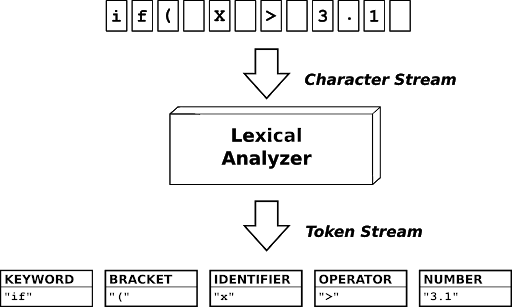
Since this project uses C language, it is designed only for the subset of C. The Lexical Analyzer is designed as a procedure that will be called every time a token is needed by the parser. The routine consists of at least two parameters that are returned to the calling routine: “Code” which is the numeric representation of the token and “String” which is the character representation of the token.

On reading the characters from the input given by the user:

* It should identify all the keywords.
* It should identify all the operators.
* It should identify all the valid Identifiers.
* It should identify the literals such as integer and real numbers (in this case).

And should print the tokens along with their symbol types.

For identification, the input tokens given by the user that are taken up the parser are compared with the entities within the defined function. On return of a Boolean value the further action takes place. Another function is written wherein the Boolean values are compared using conditional loops and finally the tokens along with their types are printed.



**Fig. 1 Illustrative Model of Lexical Analyzer**

## **2.3 Working**:

Lexical analyzer scans the entire source code of the program. It takes the input in the form of a character stream. And further produces a token stream as shown in the figure below. It identifies each token one by one.

For Example:

In this project when the user provides the following input:

**int a = 9.7 + 2**

It would result in giving the following output based on the programming done and the working explained above.

‘int’ is a keyword

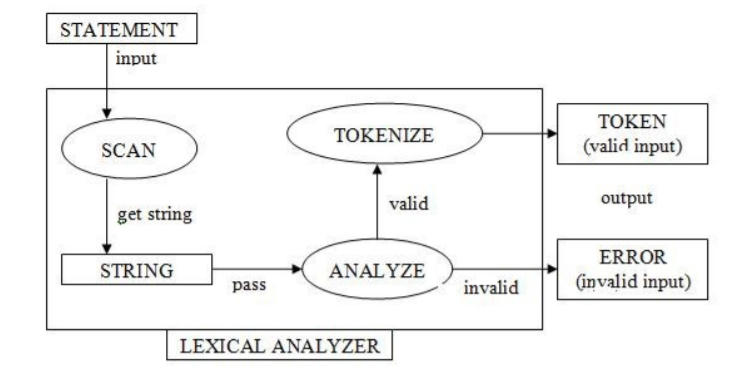
‘a’ is an identifier

‘=’ is an operator

‘9.7’ is a real number

‘+’ is an operator

‘2’ is an integer



**Fig. 2 : Data Flow Diagram for Lexical Analyzer**

**CHAPTER 3:**

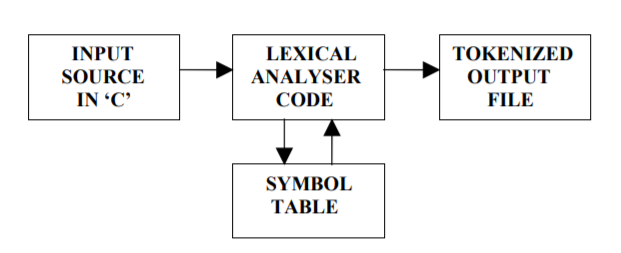
# TEST PLAN

## **3.1 Implementation details:**

This project is an implementation of Lexical Analyzer using C programming which would consist of a driver function to take input from the user in form of character stream and further provide this to a lexer.

Another function is created to perform the lexer tasks, which consists of conditional loops for comparison to generate the token types based on the symbol table.

These symbol tables are written in the program using functions of type Boolean that would return TRUE/FALSE based on the presence of the tokens in the function defined. Finally, the output is generated where the tokens along with their symbol type is printed in the output console.



**Fig. 3: Block Diagram of Lexical Analyzer**

## **3.2 Sample Specification:**

In this section a complete specification for a simple lexical analyzer is developed. This analyzer will be required to recognize and return the components of arithmetic expressions, skipping white spaces. Specifically, the components to be recognized are:

1. Valid Identifiers
2. Invalid Identifiers
3. Integer and real numbers
4. Keywords (“int”, “float”, etc.)
5. Operator symbols ("+", "-", "\*", and "/")

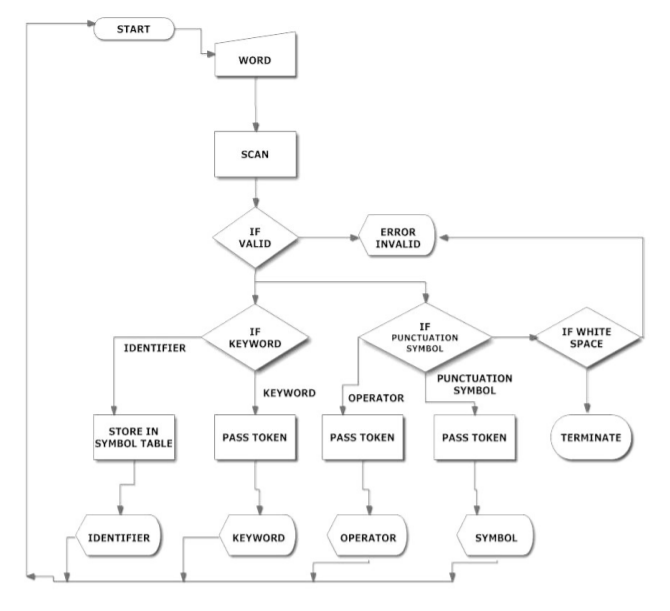
## **3.3 Actions:**

Actions must be specified for each of the patterns to be recognized. To simplify this discussion, assume there exists a function called “LEX” that creates a token value from the information collected by the lexical analyzer.

The requirements for this function will be conditional statements such as “if”, “if-else” to compare the result generated from the function to the specified boolean value. Here the character streams are first broken into simple character substrings and then are used for comparison by the created functions.

The use of this would be clear once the programming is done.

Finally, the Specification Test and Unit test should be conducted to check the behaviour of the written program against various test cases provided by the user. This would ensure the quality of the program, i.e. it would check if the code meets the requirements laid out in this document.



**Fig. 4 : Flowchart for Lexical Analyzer**

**CHAPTER 4:**

# TEST CASES

|  |  |  |
| --- | --- | --- |
| **Sl. No** | **INPUT** | **EXPECTED RESULTS** |
| 1. | int a = 9.7 + 2; | ‘int’ is a keyword  ‘a’ is a valid identifier  ‘=’ is an operator  ‘9.7’ is a real number  ‘+’ is an operator  ‘2’ is an integer  ‘;’ is a separator |
| 2. | float b = { 1c + d } ; | ‘float’ is a keyword  ‘b’ is a valid identifier  ‘=’ is an operator  ‘{’ is a separator  ‘1c’ is not a valid identifier  ‘+’ is an operator  ‘d’ is a valid identifier  ‘}’ is a separator  ‘;’ is a separator |
| 3. | sizeof (char ); | ‘sizeof’ is a keyword  ‘(‘is a separator  ‘char’ is a keyword  ‘)’ is a separator  ‘;’ is a separator |

**CHAPTER 5:**

# APPLICATIONS AND FUTURE WORKS

This lexical analyzer can be used as a stand-alone string analysis tool, which can analyze a given set of strings and check their lexical correctness.

This can also be used to analyze the string sequences delimited by white spaces in a C / C++ source code (\*.c / \*.cpp) file and output all the results in a text file, if proper functionality of file handling will be used in the source code of the lexical analyzer, this functionality will not be a part of the present project but will be available in an upgraded version, if time permits the development of it.

Furthermore, the applications of a lexical analyzer include: -

1. Text Editing

2. Text Processing

3. Pattern Matching

4. File Searching

An enhanced version of this lexical analyzer can be incorporated with a Parser having the functionality of syntax directed translation, to make a complete Compiler in the future.

**CHAPTER 6:**

# REFERENCES

1. <https://en.wikipedia.org/wiki/Lexical_analysis>
2. <https://rosettacode.org/wiki/Compiler/lexical_analyzer>
3. <https://www.geeksforgeeks.org/introduction-of-lexical-analysis/?ref=lbp>
4. <https://brainly.in/question/6387481>
5. <https://www.slideshare.net/SunitaAher1/implement-lexical-analyzer-for-clanguage>