

**Scanner for C- language**

**Course Project - 1 (July 2020 - Dec 2020)**

**CS305 Compiler Design Lab**

**National Institute of Technology, Karnataka**



**SUBMITTED TO:**

Prof. P. Santhi Thilagam

CSE Dept, NITK  
  
**TEAM**

Kesana Jahnavi (181CO127)

Shreeya Sanjay Sand (181CO150)

Shumbul Arifa (181CO152)

Keerti Chaudhary (181CO226)

**OVERVIEW**

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program. Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

INDEX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. No** | **CONTENT** | | | **PAGE NUMBER** |
| 1 | Overview | | | 1 |
| 2 | Introduction | | | 3 |
|  |  |  |  |  |
|  |  |  |
|  |  |  |  |  |
|  |  |  |
|  |  | | |  |
|  |  | | |  |
| 7 | Conclusion | | | 8 |
| 8 | References | | | 8 |

Abstract

**FEATURES**

The project objective is to construct a compiler that studies the C programming language. It will have the following features:

* The compiler is going to support the following cases:
* Keywords : eg: int, char, float
* Identifiers : eg: maximum, avg
* Constants : eg. 1, 2, 20
* Operators: eg: +, -, \*
* Strings: eg: “nitk”, “mehnaz”, “red”
* Special symbols: eg: [],\*, ()
* Support int and char data types and also short, long, signed, unsigned subtypes.
* Detection of arrays with specified datatype (eg: int arr[10])
* Detection of looping constructs such as while, nested while.
* Detection conditional statements such as if-else and nested if-else.
* Identification of user-defined functions with one argument with return types int, char, void.
* Hashing techniques used to maintain symbol and constant tables.
* Support for single-line as well as multiline comments and return appropriate error messages.
* Appropriate error messages for comments and strings that don't end until the end of the file.

**RESULTS**

* Details of the identified tokens for the source program taken as input.
* Errors in the source program along with appropriate error messages
* Symbol table will be designed using hashing organization techniques.

**TOOLS USED**

* Flex

Introduction

The Lexical Analyzer is the first phase of the Analysis (front end) stage of a compiler. In layman’s terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character and identifies ‘Lexemes’ and categorizes them into ‘Tokens’. These ‘tokens’ are represented as a symbol table and is given as input to the Parser (second phase of the front end of a compiler).

Definitions

**Tokens**

Tokens are essentially just a group of characters which have some meaning or relation when put together. The Lexical Analyzer detects these tokens with the help of ‘Regular Expressions’. While writing the Lexical Analyzer, we have to specify rules for each Token type using Regular Expression. These rules are used to check whether a certain group of

characters fall under a given token category or not. An example, in this case, would be an ‘Identifier’ token. We specify the rules for an identifier as follows: Any string of characters, that start with an \_ or an alphabet, followed by any number of \_’s, alphabets or numbers. The regular expression for Identifiers is {S}({S}|{D})\* where S is [a-zA-z] and D is [0-9].

**Lexemes**

Lexemes are instances of Tokens. An example would be ‘long int’, which is a Lexeme

of ‘Keyword’ Token.

**Symbol Table**

A symbol table is generated in the Lexical Analyzer stage, which is basically a table with the columns ‘Symbol’, ‘Type’ and ‘Token ID’. The symbol is the Lexeme itself; the ‘Type’ is the token category and the ‘Token ID’ is a unique ID given to a token, which is used in the parser stage. There are no duplicate entries in a symbol table. Each symbol is recorded only once, even if there are multiple instances. A Lexical Analyzer is internally implemented based on the concept of FSM’s (Finite State Machines). A DFA (Deterministic Finite State Automata) is internally built for each Token based on the Regular Expression provided. This is used to identify Lexemes and categorize them into Tokens.

**Flex Script**

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

*Definition section*

*%%*

*Rules section*

*%%*

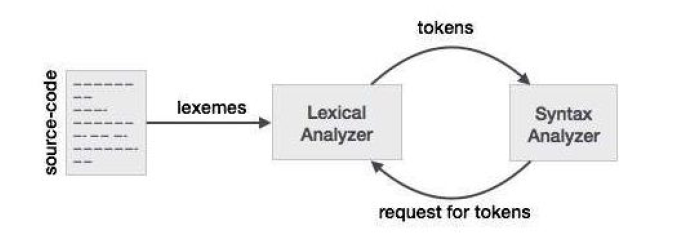
*C code section*

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

**C Program**

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned into account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input. The script also has an option to take standard input instead of taking input from a file.

DFA



Code

Explanation

* Regular expression for identifiers: The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.

[a-z|A-Z]([a-z|A-Z]|[0-9])\*

* Multiline comments: This has been supported by checking the occurence of ‘/\*’ and ‘\*/’ in the code. The statements between them has been excluded.
* Errors for unmatched and nested comments have also been displayed.
* Error Handling for Incomplete String: Open and close quote missing, both kind of errors have been handled in the rules written in the script.
* Error Handling for Nested Comments: This use-case has been handled by checking for occurrence of multiple successive ‘/\*’ or ‘\*/’ in the C code, and by omitting the text in between them.
* At the end of the token recognition, the lexer prints a list of all the tokens present in the program. As and when successive tokens are encountered, their respective values are stored in the symbol table structure and then later displayed.

Test Cases

* Without errors

TEST CASE 1: output for testcase containing conditional statement

TEST CASE 2: output for testcase containing while statement

TEST CASE 3: output for testcase containing nested conditional statement

TEST CASE 4: output for testcase containing function and print statement

TEST CASE 5: output for testcase containing while statement and print statement function

* With errors

TEST CASE 6: output displays error in incomplete string

TEST CASE 7: output displays error in preprocessor instructions

TEST CASE 8: output displays error in floating point number

TEST CASE 9: output displays the error in incomplete string, parentheses

TEST CASE 10: output displays the error in identifying the type of token

CONCLUSION

This

Future Work

The scanner that was created in this project helps in breaking source program into tokens defined by the C programming language.

In the next phase, parser will be designed which will call upon the Flex program to give it tokens and the lexical analyzer will return to the parser the integer value associated with the tokens as and when required by the parser. Together with the symbol table, the parser will prepare a syntax tree with the help of a grammar that we provide it with. The parser can then logically group the tokens to form meaningful statements and can detect C programming constructs such as arrays, loops, and functions. The parser will also help us identify errors that could not be detected in the lexical analysis phase such as unbalanced parentheses, unterminated statements, missing operators, two operators in a row, etc.

REFERENCES

* Compilers Principles, Techniques and Tool by Alfred V.Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
* <http://dinosaur.compilertools.net/lex/index.html>
* <http://www.csd.uwo.ca/~moreno/CS447/Lectures/Lexical.html/node11.html>
* <https://www.geeksforgeeks.org/cc-tokens/>
* <http://www.isi.edu/~pedro/Teaching/CSCI565-Spring11/Practice/SDT-Sample.pdf>
* StackOverflow for regex