International Journal of Scientific Research in Science, Engineering and Technology (www.ijsrset.com)
© 2020 IJSRSET | Volume 7 | Issue 2 | Print ISSN: 2395-1990 | Online ISSN : 2394-4099
DOI : https://doi.org/10.32628/IJSRSET207253

Implementation of Lexical Analysis on Assignment Statements in C++ Programming Language

Zaw Lin Oo[1], Mya Sandar Kyin[2]
[1]Faculty of Information Science, University of Computer Studies, Taungoo, Pegu Regional Division, Myanmar
[2]Faculty of Computer Science, University of Computer Studies, Taungoo, Pegu Regional Division, Myanmar

ABSTRACT

A compiler is a computer program that translates computer code written in one programming language (the source language) into another 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 [7]. The three main processes of compilation are lexical analysis, syntax analysis and semantic analysis. A compiler has two components, front-end and back-end. Front-end portion of a compiler has to do to main tasks lexical analysis and syntax analysis. On the lexical analysis, input source code are scanned and tokenized into various tokens [6]. In the system, front-end portion of the compiler, lexical analysis is used. There are many token elements in C++ programming language. In this system, line break token, white space tokens (space and tab) and operators (+,-,*,/,=,+= and so on) are used as token elements for the assignment statements of C++ source program. This system is taken all the assignment statements of C++ program as input. Of course, the extracted assignment statements may be literals or values assignment statement (e.g. x=3; or pi= 3.142;), variable assignment statement (e.g. x=y; or x=z;) and expression assignment statement (e.g. a=b+c; or x=y*z; or a=b*(c+d); and produced symbol table, step by step recognized table by using finite state automata and lexeme table.

Keywords : Learn C++ syntax, Compiling Technique, Compiler Design Theory, Tokenization in NLP, Lexical Analysis on C++

I. INTRODUCTION

Nowadays, software is primarily written in high level language by using an appropriate compiler. A compiler is a program that takes as input a program written in a source language such as Pascal or C++ and translates it into a functionally equivalent program in the target language (assembly or machine code). When no error is detected in the input source code, the translation can be completed. The process of translating a source code language to a target language code is called compilation process can be divided into two parts namely analysis and synthesis stages. In analysis stage, the source program break into constituent pieces (tokens) and creates intermediates representations (symbols). In synthesis stage, compiler generates the target program from the intermediate representations. It is a process translates what the programmer tells intermediate code optimizer into another equivalent code. The analysis part can be divided into lexical analysis, syntax analysis and semantic analysis. The next part is the synthesis part that can be divided into intermediate code generator, code optimizer and code generator [MMO].

II. METHODS AND MATERIAL

Compilation Process

The compiler is to translate the bit patterns (streams) that represent a program written in some computer language into a sequence of machine instructions that carry out the programmer's intend. The compilation process is done by a serial connection of three boxes which it is called the lexical box, the syntax box and the code generator. These three boxes have access to a common set of tables where long term or global information about the program may be entered. One such table is the symbol table, in which information about each variables or identifier is accumulated as shown in Fig1 [MMO].

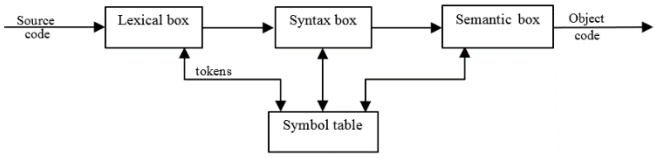

Fig.1: Step by step compiling job.

In this system, lexical phase would like to be presented in emphasis for assignment statements of the C++ source program. The input to a compiler is a bit pattern representing a string of characters. The lexical box is concerned with breaking up the string of character into the words they represent. In this phase, the input string of character is transformed into new entities. These entities are often called tokens. Each token consists of two parts, a class part and a value part. The class part denotes that the token is in one of a finite set of classes and indicated the nature of information included in the value part [1].

2.1. Lexical Analysis

Lexical analysis or scanning is the process where the stream of characters making up at the source program is read from left- to- right and grouped into tokens. Tokens are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language constants (integer, double, character, string, etc.), operators (arithmetic, relations, logical), punctuation and reserved words. Lexical analyzer take a source program as input and produces a stream of token as output. A lexeme is the actual character sequence forming a token; the token is the general class that a lexeme belongs to. Some tokens have exactly one lexeme for other, there are many lexemes (e.g. integer constant).The scanner is tasked with determining that the input stream can be divided into valid symbols in the source language, but has no smarts about which token should come where few errors can be detected at the lexical-level alone because the scanner has very localized view of the source program without any context. The scanner can report about characters that are not valid tokens (e.g. an illegal or unrecognized symbol and a few other malformed entities constant, un-terminated, comments, etc.). It does not look for or detect garbled sequences; token out of the place, undeclared identifiers, misspelled keywords, mismatches types and mismatches operators. The lexical analyzer can be a convenient place to carry out some other chores like stripping out comments and white space between tokens and perhaps even some features like macros and conditional compilation[4].For example, the assignment statement 'z=a*(x+y);' is input into the code window (text editor) and it passes through the lexical box. It is tokenized into five token elements such as '=', '*', '(', '+', and ')'. The lexical box generates and produces lexemes table. It is shown in Table1.

Table 1: Lexeme table for assignment statement

SR .No	Token Type	Lexeme
1	Identifier Token	z
2	Operator(assignment)	=
3	Identifier Token	a
4	Operator Token	*,+
5	Operator Precedence Token	'(' and ')'
6	Separator or terminator	;

2.2.1 Tokenization

A block of text corresponding to the token is known as a lexeme. A lexical analyzer processes lexemes to categorize them according to function, giving them meaning. This assignment of meaning is known as tokenization. A lexical analyzer processes lexemes to

categorize them according to function, giving them meaning. This assignment of meaning is known as tokenization. A Token can look like anything; it just needs to be a useful part of the structured text (source program syntax). An essential function of a compiler is to build the Symbol table where the identifier the program is recoded along with various attributes [3].

2.2.2 Symbol Recognition

In addition to recognizing the symbols of the language the lexical analyzer will usually perform one or two other simple tasks such as

- Deleting comments
- Inserting line numbers
- Evaluating constants

Table 2: Symbol table for assignment statement

Lexeme	Token
z	Identifier
=	Operator
a	Identifier
*	Operator
(	Operator
x	Identifier
+	Operator
Y	Identifier
)	Operator
;	Separator

Though there are arguments that the last of these is better left to the machine dependent back end of the compiler. The lexical analyzer is only concerned with recognizing language symbols in order to pass them on to the syntax analyzer. It is not concerned at all with the order in which symbols appear. It would be up to the syntax analyzer to realize that they did not form the start of any program. For the purposes of the lexical analysis, regular expressions are a convenient method of representing symbols such as identifiers and constants [2]. Let us turn to assignment statement of our example that is "z=a*(x+y);" the lexical analyzer scans the input assignment statement and generates the symbol table. This table is shown in Table 2.

2.2.3 Regular Expression and Regular Language

A regular expression is a text pattern consisting of a combination of alphanumeric characters and special characters known as meta-characters. A close relative is in fact the wildcard expression which is often used in file management. The result of a match is either successful or not, however when a match is successful not all of the pattern must match. Regular expressions are the key to powerful, flexible and efficient text processing.

Regular expressions themselves, with a general pattern notation almost like a mini programming language, can be used to describe and phrase text. With additional support provided by the particular tool being used, regular expressions can add, remove, isolate, and regularly fold, spindle, and mutilate all kinds of text data. The set of all integer constants or the set of all variable names are sets of strings. Such a set of strings is called a language. For integers, the alphabet consists of the digits 0-9 and for variable names the alphabet contains both letters and digits. Regular expressions, and algebraic notation that is compact and easy for humans to use and understand, can be used to describe sets of strings [5].

Table 3: Example of regular expression

Regular Expression	Explanation
a*	0 or more a's
a+	1 or more a's
(a/b)*	all strings of a's and b's (including)
(aa/ab/ba/bb)*	all strings of a's and b's of even length
[a-z A-Z]	shorthand for "a|b|…..|z|A|…..|Z"
[0-9]	shorthand for "0|1|2|…..|9"

Regular expressions are concise, linguistic characterization of regular languages (regular sets). Each regular expression defines a regular language a set of strings over some alphabet, such as ASCII characters; each member of this set is called a sentence, or a word we use regular expressions to define each category of tokens. For example, an

identifier specifies a set of strings that are a sequence of letters, digits, and underscores, starting with a letter. Example of regular expressions can be seen in Table 3.

III. RESULTS AND DISCUSSION

System Design for Lexical Analysis on Assignment Statement

This system intends to the learner who studies the compiler design and compiling technique. The only assignment statements of the input source code in C++ are considered and produced symbol table, step by step recognized table and lexeme table. This system cannot serve the whole lexical box functions. It is intended to easily understand and clearly seen the transition of assignment statements. This system provides the learners who learn the compiler design theory and then they may implement the whole lexical box of compilation processes.

Implementation of Lexical Analysis on Assignment Statements with C++ Programming Language is to produce symbol table, step by step recognized table (transition table) and lexeme table of identifiers and operators for the assignment statements of C++ source code. From the input source code, the assignment statements are extracted and these statements are tokenized to include each types and values of tokens for all assignment statements. And then, the symbol table is generated for each input identifier or operator tokens. Moreover, the system can recognize each input assignment statement with their states. Nowadays, the most Integrated Development Environment (IDE) such as C-Free4, C-Free5 and devc or code editor does not trim for the line spaces and white spaces (double spaces, tab and etc.). The overall system is shown in Fig. 2. In this system, it will trim that multi-lines, unnecessary spaces and tab are written by programmer. For example, z=a * (x+y); this statement is legal for C++, but not readable. It should be z=a*(x+y); that is more suitable for programmers and readable form.

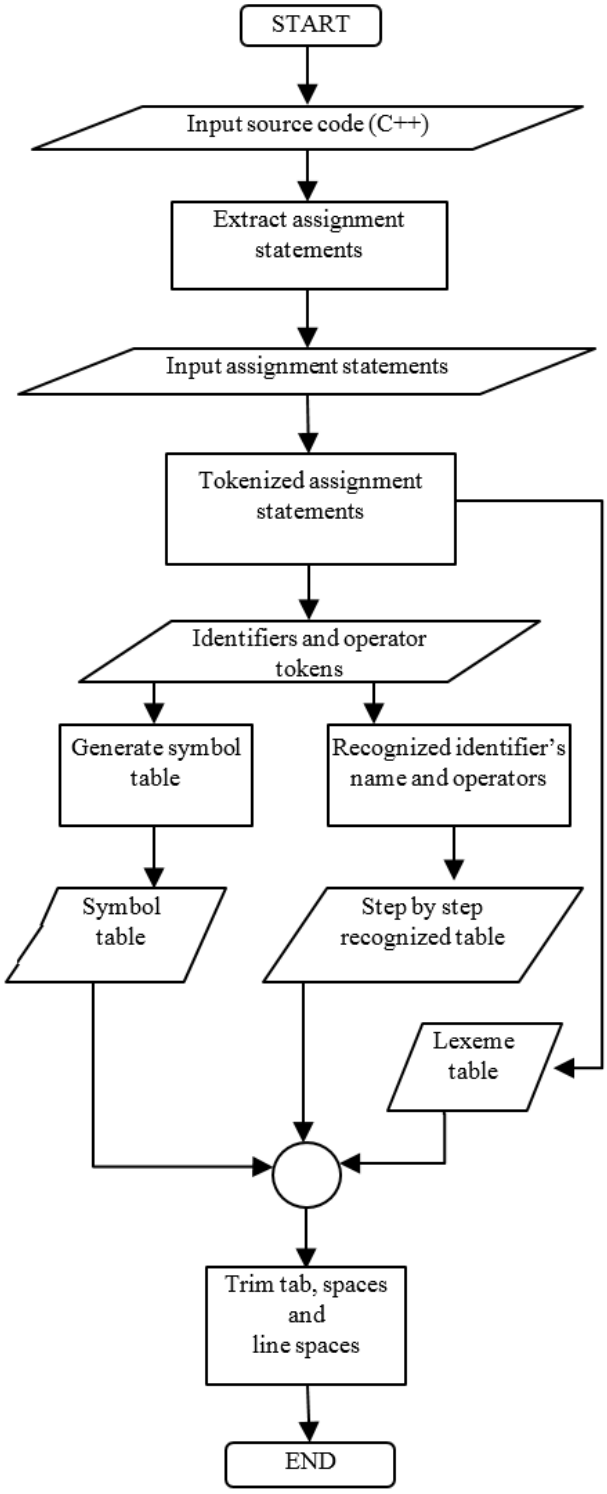

Fig. 2: System design

IV. REFERENCES

[1]. P.M. LEWISII, D.J. ROSENKRANTZ, R.E. STEARNS, "Compiler Design Theory", Third Printing, November 1978, ISBN 0-201-14455-7

[2]. ROBIN HUNTER, "The Essence of Compilers", First published In ISBN 0-13-727835-7
[3]. NWE NWE THANT, "Implementing Syntax Analyzer for Compiler Process", 2009
[4]. Maggie Johnson and Julie Zelenski, "Lexical Analysis", Printed in 2008
[5]. Myo Myint Oo, "Implementation of Lexical Analysis on Declaration Statements in C++ Programming Language", 2011
[6]. https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm
[7]. https://en.wikipedia.org/wiki/Compiler

Cite this article as :

Zaw Lin Oo, Mya Sandar Kyin, "Implementation of Lexical Analysis on Assignment Statements in C++ Programming Language", International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Online ISSN : 2394-4099, Print ISSN : 2395-1990, Volume 7 Issue 2, pp. 269-273, March-April 2020. Available at doi : https://doi.org/10.32628/IJSRSET207253
Journal URL : http://ijsrset.com/IJSRSET207253