IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 1, Ver. VI (Feb. 2014), PP 11-18
www.iosrjournals.org

LEXIMET: A Lexical Analyzer Generator including McCabe's Metrics.

Rana khudier Abbass Ahmed
Al-Rafidain University College, Computer techniques engineering department

Abstract: *Due to the complexity of designing a lexical analyzer for programming languages, this paper presents, LEXIMET, a lexical analyzer generator. This generator is designed for any programming language and involves a new feature of using McCabe's cyclomatic complexity metrics to measure the complexity of a program during the scanning operation to maintain the time and effort.*

I. Introduction

This section includes an introduction to the compiler architecture and McCabe's metrics due to its relation to the aim of the research.

1.1 Compiler Structure

A compiler is system software that converts a high-level programming language program into an equivalent low-level (machine) language program [1]. Figure (1) shows the structure of the compiler.

A typical decomposition of a compiler into phases is shown in Figure (1). In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler [2].

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the end can produce a better target program than it would have otherwise pro-duced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Figure (1) may be missing [2].

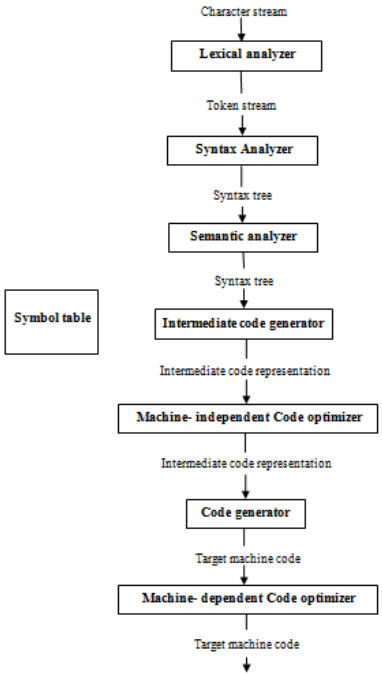


Figure (1): Phases of a compiler [2].

1. **Lexical analysis**: The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form **(token-name, attribute-value)** that it passes on to the subsequent phase, syntax analysis [2].
2. **Syntax analysis**: The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation [2].
3. ***Semantic analysis:*** The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation [2].
4. ***Intermediate Code Generation:*** In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis [2].
5. ***Code Optimization:*** The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power [2].
6. ***Code Generation:*** The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers Or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task [2].
7. ***Symbol-Table Management :*** An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned [2].

   The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly [2].

1. 2 McCabe's Cyclomatic Complexity Metrics

Cyclomatic complexity is probably the most widely used complexity metric in software engineering. Defined by Thomas McCabe in 1976 and based on the control flow structure of a program. It is easy to understand, easy to calculate and it gives useful results. It's a measure of the structural complexity of a procedure [3].

McCabe's complexity measures are based on control flow representation of the program. A program graph is used to depict control flow. In graph, nodes represent processing tasks and edges represents control flow between nodes [4].

The McCabe metric is [5]:

M = E – N + X

where M is the McCabe Cyclomatic Complexity (MCC) metric, E is the number of edges in the graph of the program, N is the number of nodes or decision points in the graph of the program and X is the number of exits from the program.

In programming terms, edges are the code executed as a result of a decision, they are the decision points. Exits are the explicit return statements in a program. Normally, there is one explicit return for functions and no explicit return for subroutines.

A simpler method of computing the MCC is demonstrated in the equation below. If D is the number of decision points in the program, then

M = D + 1

Here, decision points can be conditional statements. Each decision point normally has two possible paths [5].

The nodes of the graph correspond to the code lines of the software, and a directed edge connects two nodes if the second node might be executed immediately after the first one. If the conditional evaluation expression is composite, the expression should be broken down. For example, the expression "if(c1 && c2){}" should be treated as "if(c1){if(c2){}}". The control flow graph of a module has one and only one entry node and exit

node. If one control flow graph has e edges and n nodes, the Cyclomatic Complexity value of the corresponding module is **M = E-N+2** [6].

As an example in figure (2), is a control flow graph of a simple program, the program begins executing at the entry node, then enters a loop (group of three nodes immediately below the entry node). On exiting the loop, there is a conditional statement (group below the loop), and finally the program exits at the exit node. For this graph, E = 9, N = 8 and X = 1, so the cyclomatic complexity of the program is 9 - 8 + (2*1) = 3 [7].

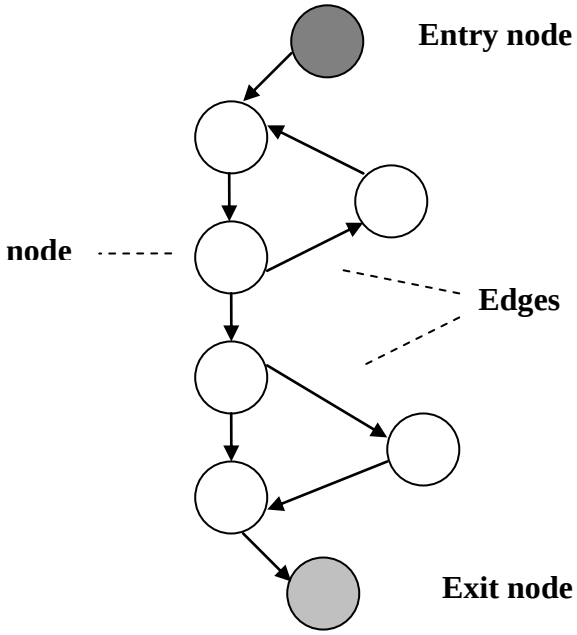


Figure (2): Control flow graph of a simple program [7].

Cyclomatic complexity metrics were always correlated with quality of the software such as reusability, maintainability, security and predicting software faults and it is a good guide for developing test cases. This metric calculates the complexity of a module based on a flow graph which makes it applicable in various representation of design
such as flow chart or Program Design Language PDL [3].

2. Related work

Hussein [8] in 2000, designed a lexical analyzer generator using Halstead's metrics. The process of generating the lexical analyzer automatically is not a new one. But such an operation that can involve any programming language and includes McCabe's metrics is a new idea, and that's what the research suggests.

3. The Aim of This Research

The aim of this research is to design a lexical analyzer generator, which we called LEXIMET, for any programming language to reduce the effort required to construct a compiler. The generated lexical analyzer has the ability to measure the complexity of the programs written in the specified programming language. The generated lexical analyzer depends on McCabe's metrics to measure the complexity.

The input of LEXIMET is the specification of a programming language and the output is a C++ program which represents a lexical analyzer for the programming language. LEXIMET was used to generate lexical analyzers for C++ and Visual Basic languages which scanned programs written in these languages successfully.

4. The LEXIMET Design

We designed a system called LEXIMET to receive the specification of a programming language as producing a lexical analyzer for the programming language. The specification includes keywords, reserved character, relational operator, numeric formats, comment formats, regular expression of the identifier, etc. The output of the LEXIMET is a lexical analyzer system, which performs the ordinary scanning process, but with a new property that is measuring the program complexity depending on McCabe's metrics. The generated lexical analyzer, is to be included in the compiler being built. Figure (3) depicts the process of generating a lexical analyzer for a specific programming language including McCabe's metrics.

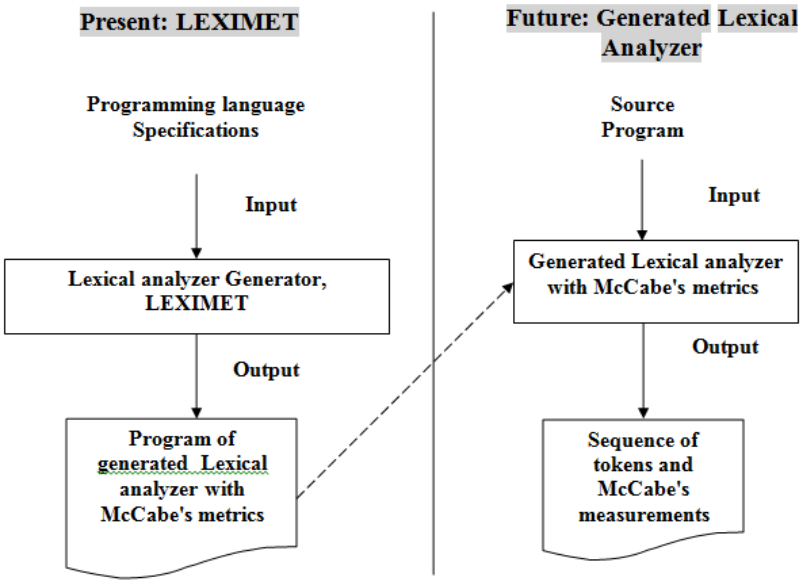
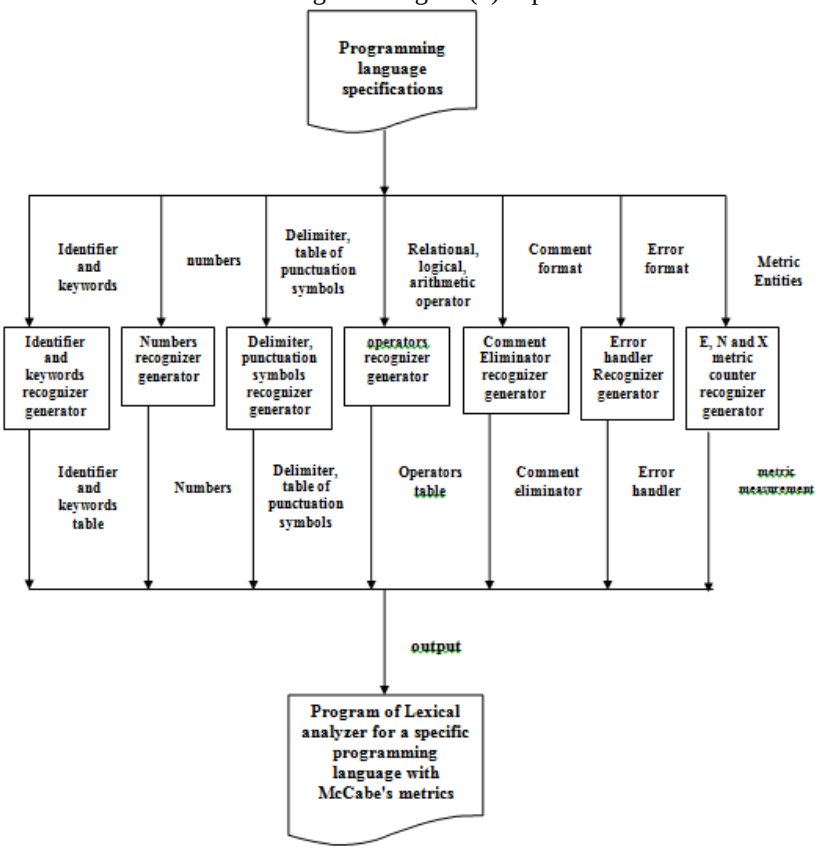


Figure (3): The role of the LEXIMET

The Architecture of the LEXIMET

Our LEXIMET system consists of many recognizers, such as identifier and reserved word recognizer, the numbers recognizer, the delimiter and punctuation symbols recognizer, the operators recognizer, the comment eliminator recognizer and the error handler recognizer. Figure (4) depicts the architecture of the LEXIMET.



Therefore, LEXIMET consists of many programming parts called, recognizer generators, each of these generators recognizes a specific part of the programming language specification and with a new feature which measures the metrics of any programming language by the E, N, and X counter recognizer to produce a specialized recognizer to be added to the generated lexical analyzer. These generators construct the recognizer as functions. Some of the recognizer generators are:

1. **Identifier and keyword recognizer generator**: This part generates a code of a routine which will recognize the identifiers and reserved words of a language. This generated code will be included in the generated lexical analyzer. For example, the identifiers for LEXIMET are described by using a transition diagram as shown in figure (5). For example, the C++ language must begin with a letter or underscore followed by letter, digit or underscore.

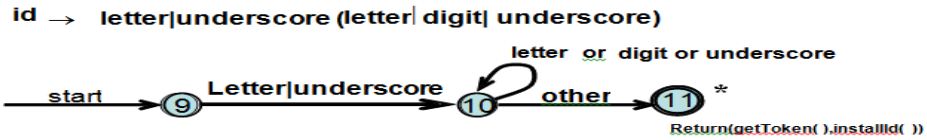


Figure (5): The transition diagram to recognize identifiers of C++ language [2].

Indeed, the transition diagram is given to the generator as a transition table as follows:

#start	Letter	Underscore	digit	Other
State 9	10	10	-	-
State 10	10	10	10	11

Figure (6), describes a part of the code generated from the LEXIMET as a lexical analyzer program for C++ identifier according to figure (4).

```
  .
  .
  .

 if (isdigit(ch)) {
        num = 0;
        do {
          ch = read_ch();
        } while ( ch != EOF && isdigit(ch));
        put_back(ch);
        return number;
      }
      if (isalpha(ch)) {
        Entry *entry;
        id_len = 0;
        do {
          if (id_len < MAX_ID) {
            id[id_len] = (char)ch;
            id_len++;
          }
          ch = read_ch();
        } while ( ch != EOF && isalnum(ch));
        id[id_len] = '\0';
        put_back(ch);
        entry = find_htab(keywords, id);
        return entry ? (Symbol)get_htab_data(entry) : ident;
      }

      . . .
```

Figure (6): Part of the code generated from the LEXIMET as a lexical analyzer program for an example, a C++ program , that describes C++ identifiers.

2. **Numbers format recognizer generator:** This part generates code to recognize numbers by the generated lexical analyzer. The number formats also described by transition diagram.

3. **Delimiter and punctuation symbols recognizer generator:** This module is responsible for generating the code which recognizes reserved punctuations, delimiters and symbols.
4. **Operator recognizer generator:** This part generates the code of generated analyzer to recognize the relational, arithmetic and logical operations. It receives the description of the operations of a language. This table consists of two entries: the operation entry and its token name. The compiler builder determines the token names. Figure (7), shows the transition diagram to recognize the relational, arithmetic and logical operations.

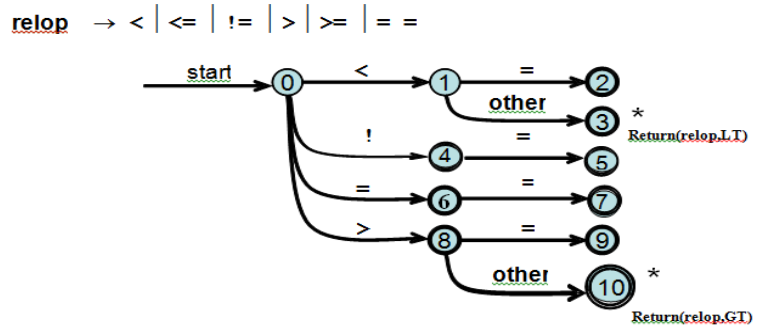


Figure (7): The transition diagram to identify tokens (relational, arithmetic and logical operations) for C++ language.

Then the transition diagram is given to the generator as a transition table as follows:

#start	<	<=	!=	>	>=	= =	other
State 0	1	-	-	8	-	-	-
State 1	-	-	-	-	-	-	3
State 8	-	-	-	-	-	-	10

Figure (8) describes a part of the code generated from the LEXIMET as a lexical analyzer program for C++ to identify relational operations.

```
switch (ch) {   # this is "switch part of the algorithm
    case EOF: return eof;

      ch = read_ch();
      return (ch == '=') ? becomes : nul;
    case '<':
      ch = read_ch();
      if (ch == '>') return neq;
      if (ch == '=') return leq;
      put_back(ch);
      return lss;
    case '>':
      ch = read_ch();
      if (ch == '=') return geq;
      put_back(ch);
      return gtr;
    default:

      error("getsym: invalid character '%c'", ch);
      return nul;
  } # case

   ...
```

Figure (8), a part of the code generated from the LEXIMET as a lexical analyzer program for an example, a C++ program , that describes how to identify relational operations

5. **Comment eliminator recognizer generator**: The comments are eliminated or removed, during lexical analysis. So the designer of the compiler must describe the format of the comment in the language that a compiler is required to by determining delimiters of the comment.
6. **Error handler recognizer generator:** This part receives the error messages and error codes to be included in the generated lexical analyzer. It is possible that none of the regular expressions denoting the tokens matches any prefix of the input. In that case, an error has occurred, and the LEXIMET must transfer control to generated module which is responsible for error handling.
7. **E, N and** X **metric counter recognizer generator:** In this part , the cyclomatic complexity is found by counting the number of linearly independent paths through the source code. This part is added for all generated lexical analyzers and it is not variant according to the specification of the language.

II. Results

Our LEXIMET system can be applied for any programming language. For an example, we examined the LEXIMET system on a visual basic program, as in figure (9).

```
Private sub command1-click( )
Dim x as integer
X=5
Text1.text="hello"
Label1.caption=x
End sub
```

Figure (9): an example of a visual basic program.

The LEXIMET system describes the token's from the chosen program and then transforms it to a c++ source code including McCabe's cyclomatic complexity metrics as shown in table (1).

Table (1): Token's described in the LEXIMET from the chosen visual basic program.

Tokens	**Sample Values**	**Informal Description**
Private	Private	Keyword private
Sub	Sub	Keyword sub
underline	_	
Click	Click	Keyword click
lparen	**(**	
rparen	**)**	
Dim	Dim	Keyword dim
As	As	Keyword as
Text	Text	Keyword text
Eq	=	
string	"hello"	
Id	X, integer, text1, text1, label1	letter followed by letters, digits, and under-scores
Caption	Caption	Keyword caption
Int	5	Integer constants
Eof		End of file

McCabe's cyclomatic complexity metrics=1

III. Discussion

The process of generating the lexical analyzers automatically is not a new one. But such an operation that includes McCabe's Metrics is a new idea, and that's what the research suggests. Although the existing of a huge number of compilers, but no one of them applies or sets up the McCabe's metrics of software engineering. We used McCabe's metrics because It can be computed early in the life cycle ,it can be used as a quality metric, and give relatively complexity of various designs. Also it is easy to apply, It can be used as an ease of maintenance metric, measures the minimum effort and best areas of concentration for testing and finally it guides the testing process by limiting the program logic during development. From the results of examining the LEXIMET on a visual basic program, we found that the value of the McCabe's cyclomatic complexity refers to a simple module without much risk according to the threshold based on categories established by the Software Engineering Institute.

IV. Future Work

1. develop the LEXIMET for parser generator and use another software engineering metric for test measurement.
2. use another software metrics with the LEXIMET to compute the product (program) quality.

References

[1]. Biswajit R Bhowmik, "A New Approach of Complier Design in Context of LexicalAnalyzer and Parser Generation for NextGen Languages" International Journal of Computer Applications (0975 –8887) Volume 6 – No.11, September 2010.
[2]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ulman, "Compilers: Principles, Techniques, and Tools" 2nd Edition, Prentice Hall, 2007.
[3]. Ayman Madi, Oussama K. Zein and Seifedine Kadry, "On the Improvement of Cyclomatic Complexity Metric" ,International Journal of Software Engineering and Its Applications, Vol. 7, No. 2, March, 2013.
[4]. N Gayatri, S Nickolas, A.V.Reddy, "Performance Analysis and Enhancement of Software Quality Metrics using Decision Tree based Feature Extraction, "International Journal of Recent Trends in Engineering, Vol 2, No. 4, November 2009.
[5]. Reg. Charney, "Programming Tools: Code Complexity Metrics", Linux Journal, 2005.
[6]. heng Yu, Shijie Zhou , "A Survey on Metric of Software Complexity", yusheng123@gmail.co, m, sjzhou@uestc.edu.cn, 2008.
[7]. Thomas J.McCabe, Sr., "Cyclomatic Complexity", www.projectcodemeter.com/cost_estimation/help/GL_cyclomatic.htm, 1976.
[8]. Hussein G. Mancy, "GENELEX: A Generator of Lexical Analyzers including Halstead's Metrics", journal of Al-Rafidain University College for sciences, vol 5, 2000.