www.ijecs.in
*International Journal Of Engineering And Computer Science ISSN: 2319-7242*
*Volume 5 Issue 12 Dec. 2016, Page No. 19507-19511*

# DOID: A Lexical Analyzer for Understanding Mid-Level Compilation Processes

*Oyebode Idris*[1] *and Adedoyin Olayinka Ajayi*[2]
[1]Ekiti State University, Department of Computer Science, Ado Ekiti, Nigeria
oyebodeidris@gmail.com
[2]Ekiti State University, Department of Computer Science, Ado Ekiti, Nigeria
dedoyyin@gmail.com

**Abstract:** ***This research was undertaken to give students in the local Nigerian environment a hands-on, practical and direct knowledge of Compiler Construction and Automata Theory, with the motivation to make it the Country's first programming language. The project was carried out using the Lex/Flex Lexical Analyzer Generator and the C Language. Compiler construction and automata theory has been one of the tougher subjects for local students in the Nigerian region. The purpose of this research is to make our programmers better understand the processes in program compilation and also catch up with the interest of nonprogrammers who may become fascinated by the exquisiteness of programming when it is only at the highest level, which is more like speaking to a fellow human. Locals and students will also be more interested in the subject matter when they are to learn a Language developed in their country.***

**Keywords**: Lexical Analyzer, Compiler Construction, LexFlex, Automata Theory, Programming Language, C Language.

## 1. Introduction

The evolution of Programming Languages started from the mnemonic Assembly Languages of the early 1950s to the first major step towards High Level Languages in the latter half of the 1950s with the development of Fortran, Cobol and Lisp. The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs. These languages were so successful that they are still in use today. More on the evolution of programming languages and their classifications can be found in [1].

Lexical Analysis, also called Scanning, is the first phase of a compiler. A compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language [1]. Character streams that make up the source program are read and categorized into Lexemes. This produces tokens which are passed to the syntax analysis phase. Other stages of compiler construction includes Semantic Analysis, Intermediate Code Generation and Code Optimization. Lexical Analysis also perform other works as: removing unnecessary blanks and comments, labeling lexemes with tokens that are passed for Syntax Analysis and updating symbol tables with identifiers and numbers.

A Lexical Analyzer performs Lexical Analysis, that is, it groups input character streams into lexemes [1]. It is a very important part of a compiler as without it, all other phases are useless, because it interfaces the syntax analysis and other phases. It can be developed by using a Lexical Analyzer/Scanner Generator (e.g. Flex), or writing it by hand in an assembly language (good for efficiency) or writing by hand in a high level language. The proposed analyzer called DoId will be achieved using the Lexical Analyzer generator. All existing programming language compilers have the lexical analyzers embedded in them, but it is not visible to the outside environment except for the Tiny Language Scanner and maybe a couple others.

## 2. Related Literature

The Compiler Project on MicroJava [2], is a small compiler for a Java-like language. The Project has three levels as follows:

i. Level 1 requires you to implement a scanner and a parser for the language MicroJava.

ii. Level 2, which deals with symbol table handling and type checking.

iii. Level 3, which deals with code generation for the MicroJava Virtual Machine.

The project was implemented in Java using Sun Microsystem's Java Development Kit [3] or some other development environment. It was really very inspiring in the development of this research, not to go into detail of it. Our research makes use of the Flex compiler [4, 6].

Also, a new a approach GLAP model for design and time complexity analysis of lexical analyzer was proposed in this paper. In the model different steps of Tokenizer (generation of tokens) through lexemes, and better input system implementation have been introduced [5].

## 3. Methods

This Lexical Analyzer was implemented by the Flex compiler, which is a Lexical Analyzer Generator. The Flex specifications are placed in a ".l" extension file which includes three sections [6]. These sections are the definition section which is optional, which can contain definitions for text replacements, global C code to be used by actions, etc. it contains literal block which is C code delimited by %{ and %} and between this are variable declarations and function prototypes. The second section is where the regular expressions and the actions are placed. The actions are C code and the rules section has the form:

$r_1$ action $_1$

$r_2$ action $_2$

$r_n$ action$_n$

i. $r_1$ is a regular expression and action$_1$ is C code fragment

ii. When $r_1$ matches an input string, action$_1$ is executed

iii. action$_1$ should be in {} if more than one statement exists

and the third section is also optional consisting of C codes also, this sections are shown as:

```
definition section

%%

rules section

%%

auxiliary functions.
```

The rules section which is the main section comprises of patterns which define the set of lexemes corresponding to a token. They are defined through regular expressions which define regular languages. A regular language is a subset of all languages that can be defined by regular expressions.

The DoId Analyzer is run by first processing the Flex file( ".l" file) which generates a scanner saved as lex.yy.c. This scanner is then compiled, it consists of a function yylex() which is going to be used as it is by running it repeatedly on the input, a main function is grabbed out of the Lex library [7] using a ( -ll ) option, this grabs the default main() routine out of the library. The scanner is then run which can take any valid inputs as specified in the DoId specifications. The steps for this are shown below:

**Commands**

```
Flex DoId.l                  #creates lex.yy.c file

Gcc lex.yy.c -ll    #compiles lex.yy.c

DoId.exe <samplefile.in    #executable file, DOID compiles a sample file
```

These commands are executed on a command line.

The DoId Analyzer can be invoked by a parser via the function yylex(), which returns a token or zero. Some tokens have further information associated with them. These are the following:

1. Tokens which indicate numbers (INTEGER and FLOAT) are first saved in the yytext variable as a string, they are then converted to the actual number and saved in the yylval global variable before being returned in the variable INT_VAL and FLOAT_VAL.

2. Tokens which indicate character strings (CHAR and STRING) return the length of the string in yylval and leave the string itself accessible in the global variable stringbuf. Escape sequences are processed before the string is returned. The string can contain embedded nulls and is not null terminated.

They are returned in the variables CHAR_VAL and STRING_VAL respectively.

3. Tokens which indicate identifiers collect the identifier string, null-terminate it, and make a privately allocated copy of it. A pointer to this copy is returned in yylval. The scanner/lexical analyzer examines the symbol table to find if the identifier is a user defined type, and returns IDENTIFIER.

4. Tokens which are reserved words are returned just as reserved words e.g Din, Dout, while, if etc.

5. Comments are terminated and not compiled with the program when run.

6. Tokens that are unidentified return error messages of "unknown character"

The Scanner reads the inputs line by line and checks for errors on each of the line, if an error is encountered on a line, it calls the error function and prints an error message. The Finite State Machine for some of the tokens can be seen in Figure 1.

## 3.1. The Program Structure in DOID

Doid programs are programs which may consist of a bundle statement and other valid DoId statements only,or a bundle statement and with classes containing valid DoId statements. Either of these may include a 'Use' statement which can be used to call the DoId library for built in classes. All valid DoId programs must also terminate with an End Statement which signifies the end of the program.

**Sample DoId Programs**

A Hello World Program in DoId can be written as follows:

```
//Hello World Program
bundle Hello;
use Doid.swing.console;

public class HelloWorld {
def main( ) {
Dout("Hello World!!!");
}
}
End
```

It can also be achieved in a much easier way as:

```
bundle Hello;
Dout("Hello World!!!");
End
```

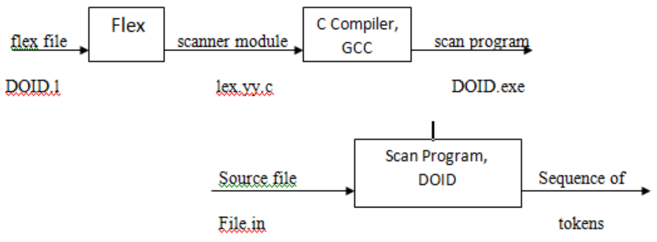

**Figure 1: Creation of DOID Lexical Analyzer.**

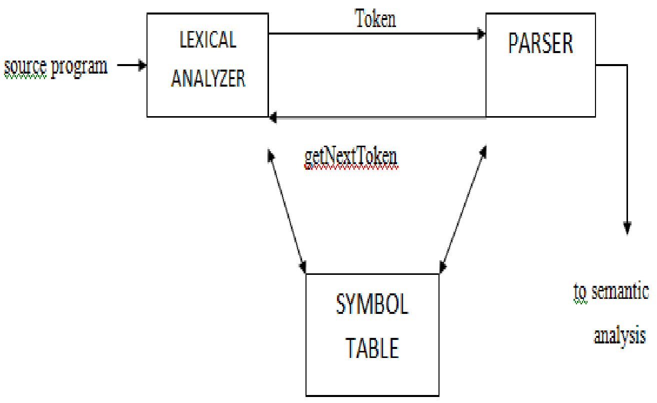

**Figure 2: Lexical Analyzer, DOID Architecture (communication with parser)**

The above Figure 2 shows how the DoId Lexical Analyzer communicates with a parser.

**Lexical Details**

Some of the tokens in DOID are

i. Integer tokens: sequence of digits, for example 1234.

ii. Float tokens: digits with decimal points or signed numbers, for example -123.45

iii. Boolean tokens: true and false.

iv. String Tokens: ASCII characters in double quotes, for example "Hi there".

v. Identifiers: a letter followed by a sequence of letters, digits and underscores.

vi. Reserved words: these are for, while, bundle, class etc.

vii. Operators: these includes arithmetic, relational etc

**Expressions**

DoId Expressions take the following forms;

a) Variables

b) Arithmetic operator applications such as +, -, /, *, % with precedence and associativity applies.

c) Relational operators applications such as not equal, <, >, <=, >=. Precedence and associativity also applies.

d) Operators 'and' and 'or' works as conjunctions and disjunctions.

**Algorithm**

The regular expressions are equal to finite automata. They are converted to an NFA by Flex, the NFA is then converted to an equivalent DFA which is simulated [1]. An algorithm for the DoId analyzer is as follows:

**Algorithm: Analyze(A)**

```
Where A = Input string
Step 1: Is A valid DOID word?
        i. if Yes, go to step 2
        ii. if No, print error, "unknown character"
Step 2: check if A is a keyword
        i. if A is keyword, return keyword
        ii. if not, goto step 3
Step 3: check if A is a special character or operator
        i. if A is special character or operator, return special character or operator respectively
        ii. if not, goto step 4
Step 4: check if A is a valid identifier
        i. if A is a valid identifier, insert identifier into symbol table and return identifier
        ii. if not, goto step 4
Step 5: check if A is a character or string value
        i. if Yes, return character or string respectively
        ii. if No, goto step 5
Step 6: check if A is integer or floating value
        i. if A is integer or floating value, convert from string to real numbers, save, then return integer or floating value respectively
        ii. if not, goto step 6
Step 7: A is a comment, discard comment.
```

The above algorithm works with tokens as its major raw material. A token recognizer is therefore used which uses transition diagrams explained in Figures 3, 4 and 5. These transition diagrams for the DoId tokens can be seen below and they are separated to avoid difficulties.

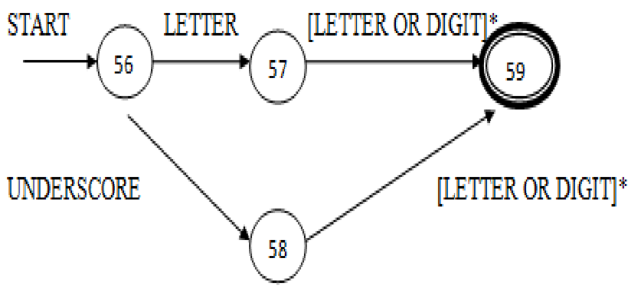

**Figure 3: Transition diagram for identifier**

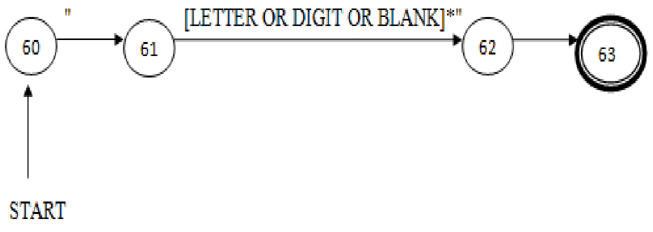

**Figure 4: Transition diagram for String**

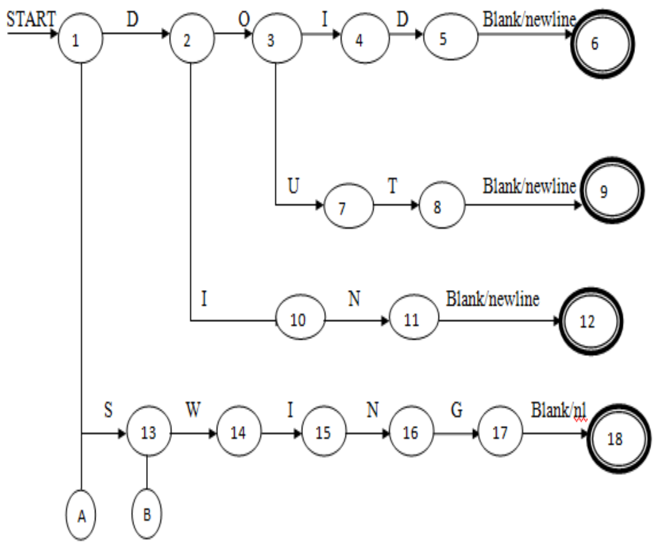

**Figure 5: Transition diagram for some DOID Keywords (a)**

DOI: 10.18535/ijecs/v5i12.43

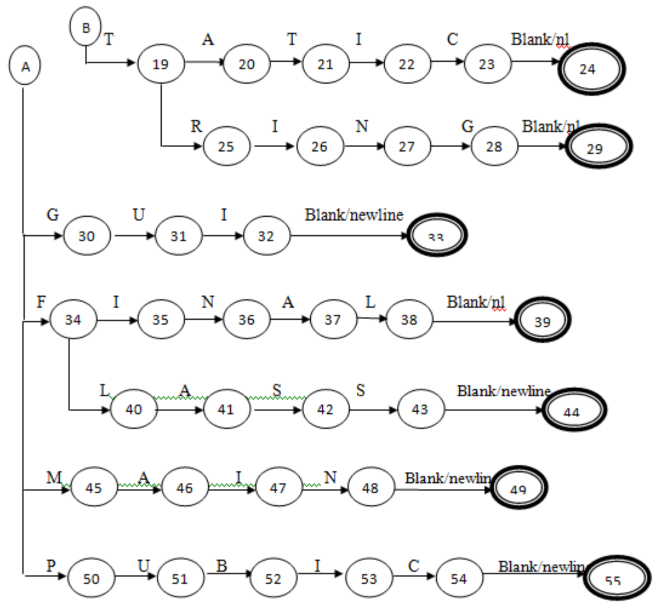

**Figure 5: Transition diagram for some DOID Keywords (b)**

Considering an example, if the action of the DoId lexical analyzer constructed from the transition diagram above. If it encounters a DOI followed by a blank, the lexical analyzer would go through states 1, 2, 3 and 4, then fail and retract the input to the identifier transition. It would then startup the second transition diagram at state 56, go through state 57 two times, go to state 58 on the blank, retract the input one position, then insert DOI into the symbol table.

## 4. CONCLUSION AND FUTURE WORK

We have briefly discussed the development of lexical analyzers using the Lexical Analyzer Generator for Languages for a proposed DOID Lexical Analyzer which offers fewer lexical specifications but still covering the necessary specifications for any standard Programming Language. We believe that having a small set of lexical specifications improves the speed of computation. So that, due to small specification set, DOID provides faster computation and is easier to learn. Our implementation of this will also enhance the way local students view the study of Programming Languages and the possibility and need to research more in the field, and the beauty of the desugaring model.

We hope that our work encourages researchers and developers to look more into the branch of developing more efficient and effective programming languages for and can be used as basic guide for future researches. We hope this will be a platform to build a local indigenous programming language.

## REFERENCES

[1] AHO, Alfred V., RAVI Sethi, LAM, Monica S. and JEFFREY D. Ullman, Compilers, Principles, Techniques and Tools., Addison-Wesley, Boston, 2006.

[2] Hanspeter Mössenböck (2014). “MicroJava”. http://microjava.com. Last accessed: 21 December, 2014

[3] Sun Microsystem's Java Development Kit JDK available via: http://java.sun.com/j2se/1.5.0/index.jsp)

[4] Aaby Anthony (2004). Compiler Construction using Flex and Bison. Available at: http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf. Last accessed: 21 January, 2015

[5] BISWAJIT Bhowmik, ABHISHEK Kumar, ABHISHEK Kumar Jha, RAJESH Kumar Agrawal. (2010) A New Approach of Complier Design in Context of Lexical Analyzer and Parser Generation for NextGen Languages. International Journal of Computer Applications (0975 – 8887)Volume 6– No.11, September 2010.

[6] MUHAMMED, Mudawwar.(n.d.) Scanning Practice: Using the Lex Scanner Generator, a TINY Language and Scanner – Compiler Design

[7] LESK, Michael E., SCHMIDT, Eric (1975). Lex – A Lexical Analyzer Generator. New Jersey: Bell Laboratories