www.ijecs.in
International Journal Of Engineering And Computer Science
Volume 9 Issue 10 October 2020, Page No. 25246-25251
ISSN: 2319-7242 DOI: 10.18535/ijecs/v9i10.4497

DOID: A Syntax Analyzer for Mid-Level Compilation Processes

[1]Ajayi Adedoyin Olayinka[*], [2]Adetolaju Olu Sunday, [3]Oyebode Idris

[1]Ekiti State University, Department of Computer Science, Ado Ekiti, Nigeria
[2] Ekiti State University, Department of Computer Science, Ado Ekiti, Nigeria
[3] Maynooth University, Department of Computer Science, Co Kildare, Ireland

Abstract: The importance of syntax analysis in the development of a compiler and Programming Languages cannot be overemphasized. Without it, we cannot even program correctly or properly. This research was performed in order to practicalize the theoretical knowledge of Compiler Construction and Automata Theory, and also with the motivation to make Nigeria's first Programming Language. The research discusses the development of Syntax Analyzers using Parser generators and also the general program structure in the DOID Language whose Lexical Analyzer was previously discussed. The project was carried out using the de-sugaring process, Yacc/Bison Parser Generator and the C Language. The results show there is faster computation at a trade off with smaller specification set, fewer syntaxes and easy declaration of variables when compared to some other derived programming languages.

Keywords: Yacc/Bison, Compiler, Automata, de-sugaring, Syntax Analyzer, Lexical Analyzer, Programming Languages, DOID.

Introduction

The evolution of Programming Languages has been discussed in [1] [2]. By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. In object oriented languages such as C++ and Java, the key concept is class which is a user-defined type; this class contains methods and constructors which in turn contains declarations and expressions and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation [3] but grammars are preferred because they offer more significant benefits such as giving a precise, yet easy-to-understand, syntactic specification of a programming language, allowing a language to be evolved or developed iteratively, by adding new constructs to perform new tasks and so on.

Related Literature

Since the 20th century, many programming languages have been developed and their syntax has improved over time. For example, the syntax in wording for uncaught exceptions was changed in between C++14 and C++17 [4], within the same update, static assert rules also changed as proposed by Brown in [5]. Changes between compiler and programming language updates not only involves addition of new features, removal/deprecation of features also occur. For example, C++20 deprecate the use of comma operator in subscripting expressions as they are thought to be confusing and not very useful [6].

This is not exclusive to C++ as it affects different kinds of programming Languages. Another common example is the difference between python 2 and python 3‟s print() and input() functions, the iterable objects instead of lists, next() and .next() functions, among others [7]. Java, one of the most popular programming languages in the world has also had steady updates over the years. The latest most common version java 11, has new string features such as .isBlank(), .lines() and many more added to it [8]. However, there have been newer versions after this and java 15 is set to September 2020 [9].

Prof. Dr. Hanspeter Mössenböck of the University of Linz's Compiler Project on MicroJava, which is a small compiler for a Java-like language [10]. The Project has three levels as follows:

- Level 1 requires you to implement a scanner and a parser for the language MicroJava.
- Level 2, which deals with symbol table handling and type checking.
- 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.

Also, Syntax Analysis chapter of Alfred V. Aho's Dragon Book on compilers was really of great help.

Methodology

The DOID Parser consists of Syntax Analysis (Parsing) and part of the Semantic Analysis of a Compiler, hence the term, "mid-level compilation processes". The parser is generated using the Yacc compiler. The "DOID.y" specification file is supplied to the Bison compiler which transforms the file into a C program called "y.tab.c", this file is then compiled with the C compiler, GCC. This generates the DOID parser (an executable file, DOID.exe) which can then be given input and produces output with respect to the grammar specified in the DOID specification file. This process is shown in the figure below:

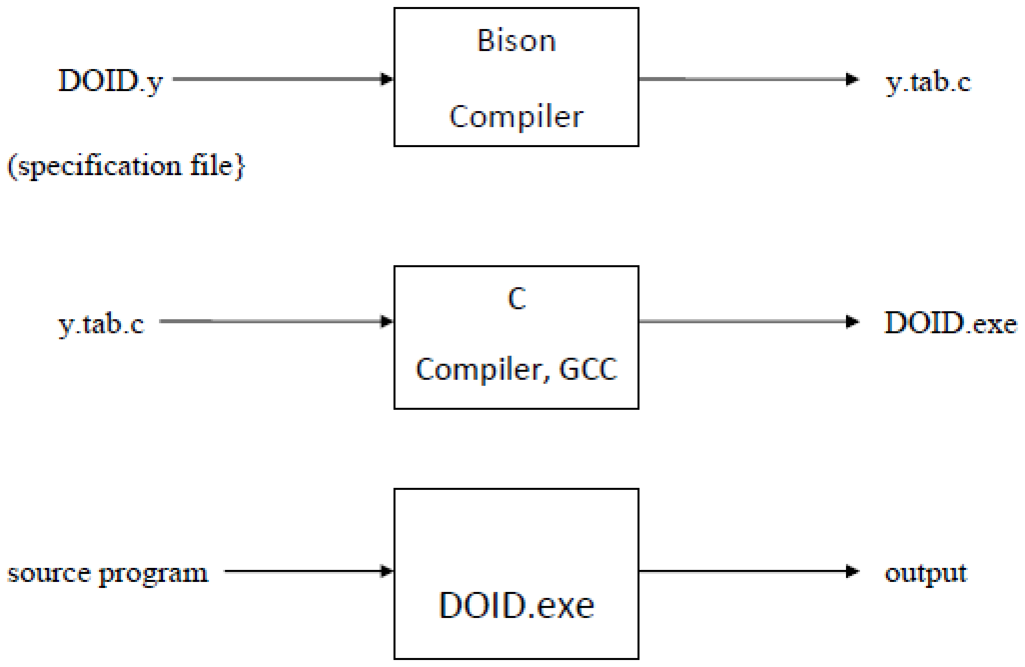

Figure 1: Creation of DOID Parser

Syntax Analysis/Parsing

Parsing according to [10] is the act of turning an input character stream into a more structured, internal representation. A common internal representation is as a tree, which programs can recursively process. The roles of a typical parser mentioned in [11] are:

a. Identify the language constructs from a given input. A parser outputs and represents valid input in the form of a parse tree [12].

b. For grammatically incorrect input string, the parser declares the detection of syntax error. No parse tree [13] in this case is generated.

The de-sugaring process discussed by Shriram in [5] [14] is the major idea behind the DOID parser, that is if we have fewer specifications as in [2] coupled with fewer syntaxes such as allowing the usage of just two looping structures amongst others makes the language simpler, run faster and aids easy teaching and understanding. This includes not neglecting the need for a programming language to be "complete". Although, there are several classifications of programming langauges such as the Object Oriented Languages which includes Languages as C++ and Java, also other languages as scripting languages, declarative, imperative and others. DOID employs the Object Oriented form and it is also a declarative language. The Syntax Analyzer was implemented by a Bison/Yacc (Yet Another Compiler Compiler)

compiler, which is a Syntax Analyzer Generator. The Yacc specifications are placed in a ".y" file which has three sections as follows;

```
Definition section
%%
Transition rules section
%%
Subroutines section
```

The variables returned from the Scanner are declared and used here with context free grammars to specify the syntax of the language. The definition section contains a part where C declarations are inserted, delimited by %{ and %}. For example an *include* call as

#include <stdlib.h>

This causes the preprocessor to include the <stdlib.h> header file. The Definition section also contain yacc grammar definitions as

%token DOID SWING GUI CLASS IDENTIFIER VOID

%token PRIVATE PROTECTED STATIC

%nonassoc STRING_VAL INT_VAL FLOAT_VAL RARR CHAR_VAL

%left ME LE EQ NE MORE LESS

%left MULT DIV

The first line is a declaration of DOID, SWING, GUI, CLASS, IDENTIFIER and VOID as valid DOID tokens. These tokens can subsequently be used in the second and third parts of the specification file.

The Translation Rules section is contained between the two %%. That is, after the first and before the second. The rules consist of grammar productions and associated semantic actions. The productions represent the context free grammar of our language. An example from the DOID language is below:

```
components: packname headfile class AT_END {printf("no errors found, compiling..."); }
|packname class AT_END {printf("no errors found, compiling..."); }
```

The non-terminals are "components", "packname", "headfile" and class as they are further defined in the program.

The subroutine section contains valid C code that supports the language processing, here, a symbol table implementation is often found which is used to to keep track of the identifiers encountered in the source code. Also, functions that might be called by actions associated with the productions in the translation rules part. These functions have also been defined in the first part (definition section).

Error recovery in the grammar is done by a few strategically placed „„error"" productions. These are invoked when the input to the parser cannot be matched by any syntactic construct known to it. A syntax error is reported in these cases and parsing continues with incorrect parse tree segments suitably disposed of.

The DoId Algorithm Showing the Lexical Analyzer [2] and Parser Interaction is shown in Figure 2.

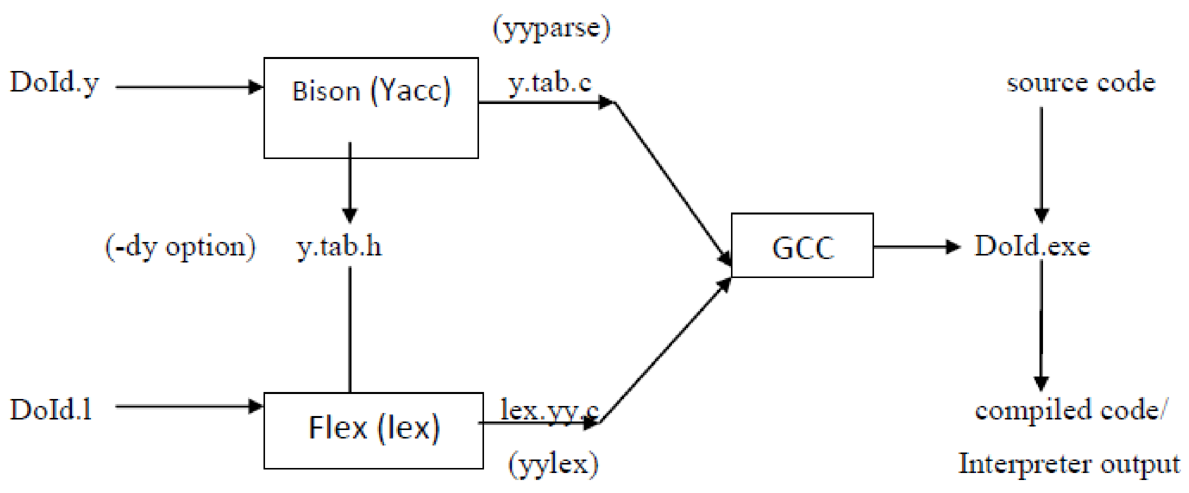

Figure 2: DoId Algorithm Showing the Lexical Analyzer and Parser Interaction

Error Reporting

Error reporting in DOID is done by calling the function yyerror(char *s). Errors such as syntax errors are specified in the Syntax analyzer while lexical errors such as open string and unknown characters are specified in the Scanner.

The Language, DOID

DOID is a simple imperative language which can occur in two forms;

1. a bundle statement, at least a class and an end statement and

2. a bundle statement, some valid DOID statements and an end statement.

The class can also contain valid DOID statements and also a call to the library as the header file. The program starts execution from the top to the bottom, i.e Top-Bottom execution. It is a mini language which will still be extended and also codes will be generated later. Also, It does not involve Garbage Collection.

Some small DOID programs

A couple of small programs in DOID includes the popular Hello World program;

```
//Hello World Program
bundle Hello;
use Doid.swing.console;
public class HelloWorld {
def main( ) {
Dout("Hello World!!!");
}
} End
```

This can also be simply achieved with the following program;

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

We can also have a program that takes in two numbers and adds them;

```
//Program that adds two no.s
```

```
Bundle addition;
Dout("Enter two no.s for addition");
Din(a,b);
C=a+b;
Dout("The sum is ",c);
End
```

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. A main method is not really necessary for a program to run, but if put in the program does not make a difference. A method and class names can be either private, protected or public and they are assumed public if not defined. Inheritance and encapsulation which are features of Object Oriented Programming are also possible in DoId. IF, WHILE and FOR statements are available conditional statements and also RETURN statements are used to return values.

Types

As in Python, a valid variable name does not need a data type specifier such as the int, float, char and string data types written before a variable to be declared. The variable is just declared and when a value is given to it, it automatically switches to the data type of that value being supplied. Boolean values True and False also follows the same ruling and also method definitions where a return value is needed doesn"t need specification, the compiler automatically knows the type of value being returned and just return it.

Statements

The statements in DoId are of different forms we use the BNF (Backaus Naur Form) forms used in the Parser to show them. They include;

- Expression ";"
- Empty Statement: ";"
- Variable declaration: Identifier ";"
- Assignment: Identifier "=" expression ";"
- Conditional: "if" "(" Expression ")" Statement "else" Statement. It can also be without the 'else' part.
- While loops: "while" "(" Expression ")" Statement
- For loops: "for" "(" Expression ")" Statement
- Input Statements: "Din" "(" input ")" ";"
- Output Statements: "Dout" "(" output ")" ";"
- Return Statements: "Return" Statement ";"
- Blocks: "{" Statements "}"

Conclusion

We have discussed the development of Syntax Analyzers using Parser generators as Bison and also the general program structure in the DOID Language. As a result of the de-sugaring process of Shriram in [14], we agree with the propositions in [2] and [14] that programs will truly run faster when the process is applied, that is faster computation at a trade off with smaller specification set. Also, the syntaxes are fewer and declaration of variables is easy as in the Python language which does not involve adding the variable type before.

References

[1.] Aho, Alfred V. Ravi Sethi and Jeffrey D. Ullman (2007). Compilers, Principles, Techniques and Tools. Addison-Wesley, Boston.

[2.] Oyebode, Idris and Ajayi, Adedoyin O. (2016) “DOID: A Lexical Analyzer for Understanding Mid-Level Compilation Processes” Journal of Engineering and Computer Science (IJECS),

https://www.ijecs.in, Volume 5 Issue 12, Dec 2016, 19507-19511, available at:]http://www.ijecs.in/index.php/ijecs/article/view/3442/3200. Last accessed: March 10, 2020.
[3.] McCracken Daniel and Reilly Edwin (2003). Backus-Naur form (BNF), Encyclopedia of Computer Science. 129-131.
[4.] Sutter Herb (2014), Wording for std::uncaught_exceptions, N4259 proposed wording for N4152. Available at: http://isocpp.org/files/papers/n4259.pdf. Accessed 30 May 2020.
[5.] Brown, Walter. Extending Static_assert, V2. 2014.
[6.] Jabot, Corentin. “P1161R2: Deprecate Uses of the Comma Operator in Subscripting Expressions.” Www.Open-Std.Org, 21 Jan. 2019, www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1161r2.html. Accessed 30 May 2020.
[7.] Raschka, Sebastian. “The Key Differences between Python 2.7.x and Python 3.x with Examples.” Dr. Sebastian Raschka, 1 June 2014, sebastianraschka.com/Articles/2014_python_2_3_key_diff.html. Accessed 30 May 2020.
[8.] “Java API Reference.” Docs.Oracle.Com, 2018, docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html. Accessed 30 May 2020.
[9.] “JDK 15.” Openjdk.Java.Net, 2020, openjdk.java.net/projects/jdk/15/. Accessed 30 May 2020.
[10.] Shriram, Krishnamurthi (2012). Programming Languages: Application and Interpretation Second Edition. Addison-Wesley, Boston.
[11.] 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*
[12.] David Galles (2005). Modern Compiler Design, Addison- Wesley.
[13.] Davide Pozza, Riccardo Sisto, Luca Durante, Adriano Valenzano (2006). Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software, IEEE Xplore, 0-7803-9575-1.
[14.] Shriram, Krishnamurthi (2012). Programming Languages - Lecture 1. September 5th. Available at: http://www.youtube.com/watch?v=PQPXkATdc04 Accessed: February 2020.