Compiler Construction Detail Design

Dr. Nwanze Ashioba, Ndubuife Nonso Daniel

Abstract
A compiler is a language translator that translates a program written in high level programming language to an equivalent machine language. Compiler construction primarily comprises of some standard phases such as lexical analysis, syntax analysis, semantics analysis, intermediate code generation, code optimization and code generation. This paper analyzes the detail design of the various phases of compiler.

Keywords – Compiler, machine language, language translator

1.0 INTRODUCTION

The concept of compliers was introduced by American Computer Scientist, Grace Brewster Murray Hopper in 1952, for A-0 programming language [1]. A compiler is a language translator that translates or converts program written in high level programming language like Pascal, Java, Fortran etc., to machine code. Computer and operating systems constitute the basic interfaces between a programmer and the machine. [2]. The compiler reports to the user the present of errors in the source program and also, reads its variables from the symbol table. The structure of a compiler is illustrated in Figure 1.

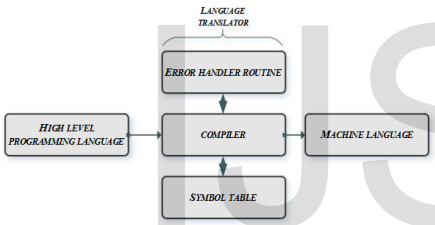

Figure 1: Structure of a Compiler

2.0 Conceptual Framework of Compilers

A compiler operates in phases and each of which transforms the source code form one representation to another thereby passing its output to the next phase. A compiler is divided into six phases, namely lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, code optimizer and code generation. The conceptual framework of a compiler is illustrated in Figure 2.

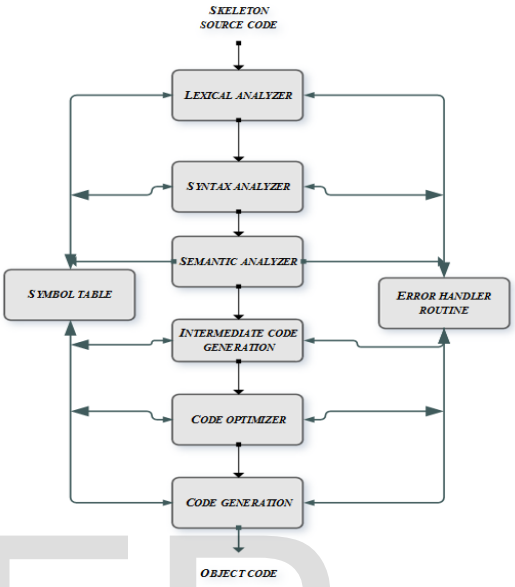

Figure 2: Conceptual framework of a compiler

3.0 Detail Design of Compilation Phases

A common division of the compilation phases is described as follows:

3.1 Lexical Analyzer

Lexical analyzer, also called token structure or scanning or tokenization, scans a sequence of characters that make up the source code and group them into a sequence of lexical token classes like identifier, keywords, operators, delimiter and separators. These words that make up the source code are called the lexemes of the programming language. A lexeme is a sequence of character string, in the program, that matches the pattern of token classes in the programming language [3]. The detail design of the lexical analyzer is shown in Figure 3. The lexical analyzer is implemented with the lexical analyzer tools like flex, lex, jflex and also with the state machine.

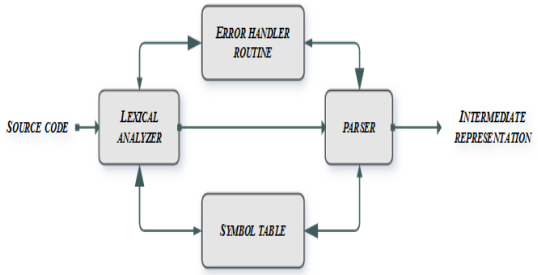

Figure 3: Detail design of the lexical analyzer

The output from the lexical analyzer is passed to the syntax analyzer for implementation. The lexical analyzer also forward the error messages to the error handler and the token are stored in the symbol table.

3.2 Syntax Analyzer

The syntax analyzer, also called parse tree, creates the syntactic structures of the source program. A parse tree is a graphical representation of the statement derivation. The parse tree uses the first components of the tokens produced by the lexical analyzer to create a tree like the intermediate representation that shows the grammatical structure of the token stream [4]. The syntax analyzer phase is shown in Figure 4.

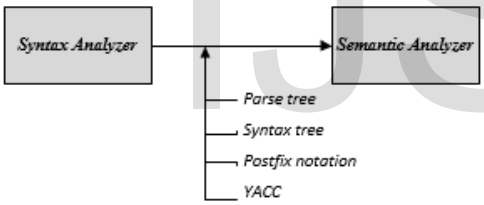

Figure 4: Detail design of syntax analyzer

Syntax analyzer is implemented using parse tree, syntax tree, grammar and YACC. The output from the syntax analyzer is passed to the semantic analyzer.

3.3 Semantic Analyzer

The semantic analyzer checks whether the input forms a sensible set of instructions in the programming language. The large part of the semantic analyzer consists of tracking variables, functions and type declarations. The output from the semantic analyzer is passed to the intermediate code generator [4]. The detail design of the semantic analyzer is shown in Figure 5.

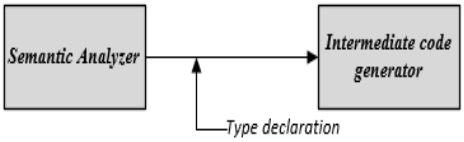

Figure 5: Detail design of the semantic analyzer

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.

3.4 Intermediate code generator

The intermediate code is also called a middle-level language code. The generator represents its instruction as a syntax tree, postfix notation and three address codes which are expressed as quadruples, triples and indirect triples. The detail structure of the intermediate code generation phase is illustrated in Figure 5.

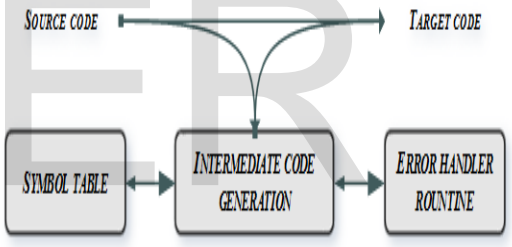

Figure 5: Intermediate code generator

3.5 Code optimization

Code optimization is the process of transferring a piece of code from the intermediate code generation phase to make it more efficient without changing its output or side effects. It attempts to improve the intermediate code, so that a faster running machine code can be produce. Code optimization can be implemented by using the following techniques: constant folding elimination, common sub-expression elimination, variable propagation elimination and dead code elimination. The structure of the code optimization phase is illustrated in Figure 6.

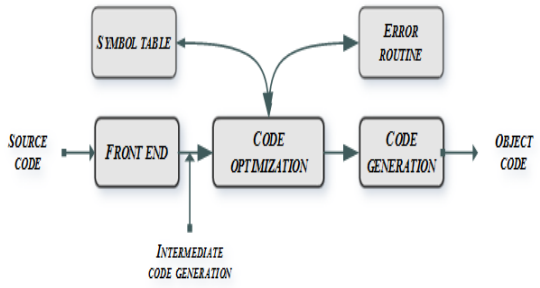

Figure 6: Detail design of the code optimization phase

3.6 **Code generation phase**

Code generation is the final phase in the compilation process. It is the process by which a compiler's code generator converts some intermediate representation of the source code to a form that can be readily executed by a machine. The detail design of the code generation phase is illustrated in Figure 7.

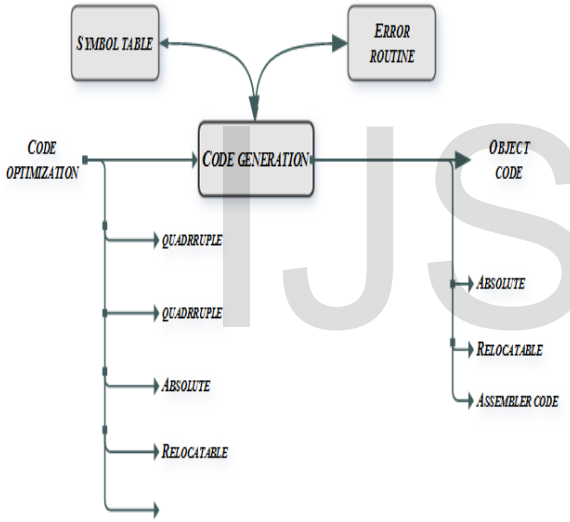

Figure 7: Detail design of the code generation phase

4.0 Compiler implementation

Summary of the implementation processes of the compiler is illustrated in Figure 8.

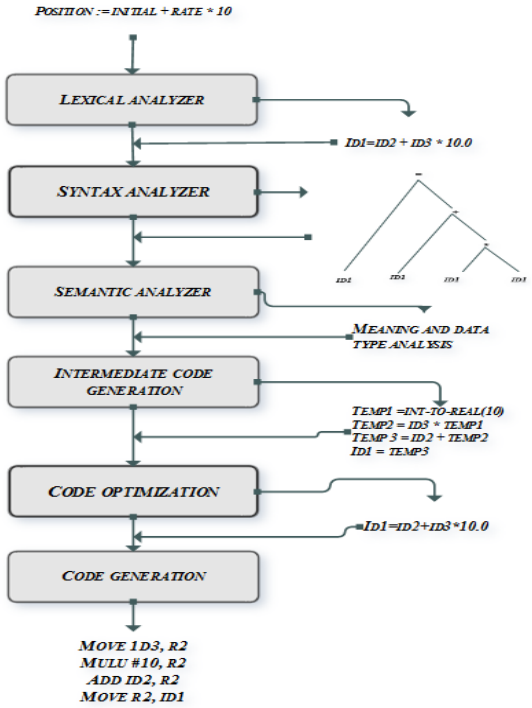

Figure 8: Compiler implementation phases

5.0 CONCLUSION

This paper has outline the basics of the compilation phases as well as detail design of the phases of a compilation processes which are used to construct a well-designed compiler.

REFERNECES

[1] P. Prajakta and M. Dawale (2019). Introduction to Compiler and its phases. *International Research Journal of Engineering and Technology (IRJET), Vol. 6, Issue 01.*

[2] Md. A. Hossain, R. Rihab, H. Islam and A. Azam (2019). A study on language processing policies in Compiler Design. *American Journal of Engineering Research (AJER), Vol. 8, Issue 12, pp. 105-114.*

[3] A. N. Jalgeri, B. B. Jagadale and R. S. Navale (2017). Study of Compiler Construction. *International Journal of Innovative Trends in Engineering (IJITE), Vol. 28, issue 46, No. 2.*

[4] M. Jain, N. Sehrawat and N. Munsi (2014). Compiler Basic Design and construction. *International Journal of Computer Science and mobile Computing (IJCSMC), Vol. 3, Issue 10, pp. 850-852.*