International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 03 | Mar 2021 www.irjet.net p-ISSN: 2395-0072

An Overview of Compiler Construction

Bashir S. Abubakar[1], Abdulkadir Ahmad[2], Muktar M. Aliyu[3], Muhammad M. Ahmad[4], Hafizu U. Uba[5]

Department of Computer Science

Kano University of Science and Technology, Wudil, Kano, Nigeria

---

Abstract – *Research in compiler construction has been one of the core research areas in computing. Researchers in this domain try to understand how a computer system and computer languages associate. A compiler translates code written in human-readable form (source code) to target code (machine code) that is efficient and optimized in terms of time and space without altering the meaning of the source program. This paper aims to explain what a compiler is and give an overview of the stages involved in translating computer programming languages.*

Key Words: **- compiler, phases of a compiler, analysis, synthesis, features of a compiler**

1. INTRODUCTION

Assembly or high-level languages are the languages used to write a computer system program. However, a computer system understands none of these languages. Therefore, a compiler is needed to translate the high-level language. A high-level language is a language written in a human-readable form with an easy-to-read syntax [6]. Examples of such languages are Java, C#, Delphi, Ruby and many others. Any computer program written in a high-level language is known as source code. A compiler uses a source code as input, processes it and produces an object code without changing the meaning of the source code [6]. The object code is sometimes called machine code or target code [7].

A compiler is a computer system software that transfigures source code into an intermediate code which afterwards transformed into target code without altering the meaning of the source code [5, 3]. The result of this transformation (machine code) must be efficient and optimized in terms of time and space (memory size). The interface between a computer programmer and a computer system is the compiler and the operating system [3]. A compiler detects an error(s) in the source code during compilation processes and handle. There are three types of error in computer programming. They are syntax, runtime and logic error [7, 6]. The only detected error during compilation processes is the syntax error. The other two types of errors occur during program execution [4].

The back-end and the front-end [7] are the two parts of a compiler. The task of the back-end is to synthesis the target language. Then, the front-end analyses the source code [6]. In a perfect compiler design, the back-end will lack any knowledge of the source code, and the front-end will also lack knowledge about the target code. A compiler operates in stages. Each stage performance a specific task. These stages are a scanner, parser, semantic analysis, intermediate code, code optimization and code generator [1, 6, and 7].

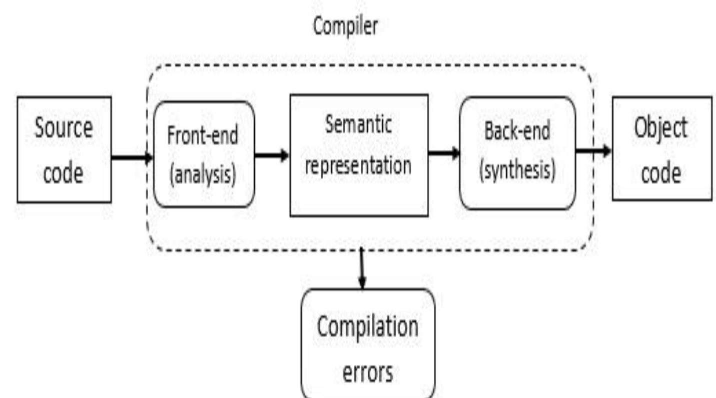


Fig 1: Abstract view of a compiler

1.1 Features of a compiler

a. Correctness
b. Speed of compilation
c. Preserve the correct meaning of the code
d. Compile-time proportion to program size
e. Good diagnostics for syntax errors
f. Good error reporting and handling
g. Work well with the debugging

1.2 Types of compiler

A compiler is divided into 3, namely:

a. Single-pass compiler
b. Two-pass compiler
c. Multi-pass compiler

2. THE COMPONENTS OF A COMPILER

Before a compiler translates source code to object code, the source code undergoes a series of steps, and these steps are called phases of a compiler [6]. Each stage performs a single and unique duty. A data structure called a symbol table is needed to store the output of each stage, and an error handler needs to be present to keep tracks of errors encounter [7].

The phases of a compiler consist of six (6) phases. These phases can be regrouped into two (2) categories as follow below [6].

2.1 Analysis:

The source code is divided into meaning characters and creates an intermediate representation. This part is further subdivided into three (3) as follows:

a. Lexical analysis
b. syntax analysis
c. Semantic analysis

2.2 Synthesis

The output of the analysis is used here to produce the desired machine-oriented code. This section is subdivided into three (3).

a. Intermediate code generation
b. code optimization
c. code generator

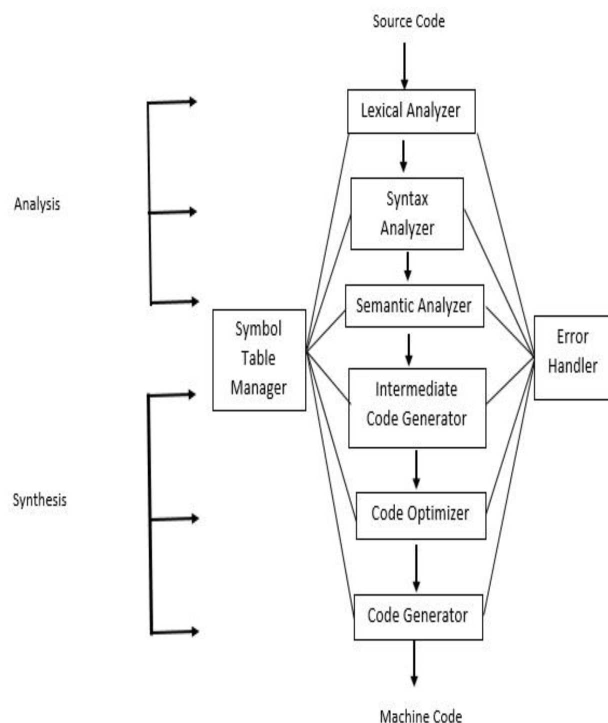


Fig 2: Block Diagram of Compiler

LEXICAL ANALYSIS

Lexical analysis is the first stage in compiler construction. This stage is also called scanning [6]. In this stage, the source code is scan to remove any whitespace or comments. Then, the source code is categories into meaningful sequences of lexical item called tokens.

A token may be composed of a single character or sequence of character. A token is classified as being either: Identifiers, Keywords Operators, Separators, Liberals, and Comments. For each lexeme the scanner produces a token as output in the form [7]: <Token- name, attribute-value>

A lexical analyser may be implement using Regular expression from automata theory and deterministic finite automata [6]. A Regular expression is used to specify the token while deterministic finite automata are used to recognise the token. Now let analyse the following:

Count = frequency + 1

Lexeme(collection of characters)	Tokens(category of lexeme)
Count	Identified (id)
=	Assignment operator
frequency	Identified
+	Addition operator
1	Integer constant

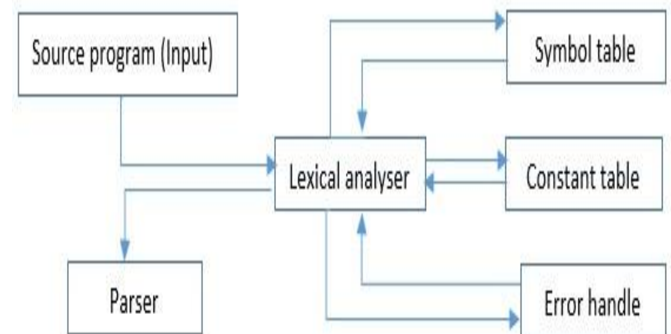
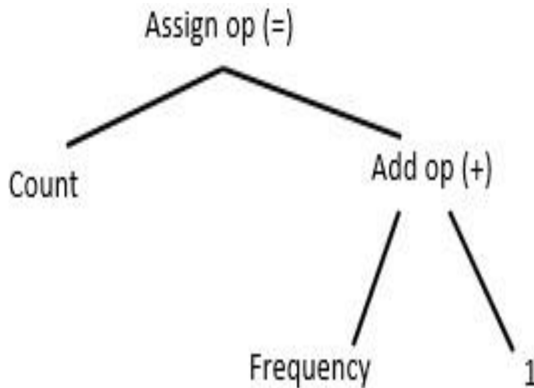


Fig 3: Lexical Analyzer Interface

SYNTAX ANALYSIS

The next stage immediately after the scanner is the syntax stage. This stage is also known as parsing [6]. The parsing stage takes a token produced in the first phase and constructs a syntax tree (parse tree). The goal of parsing is to determine the syntactical validity of a source string.

Parsing is implemented using context-free grammar (CFG) [6, 7]. A context free grammar CFG notations are used to the syntactic specification of any program.
Now let analyse the following:
Count = frequency + 1



Parse tree as an output of Parser

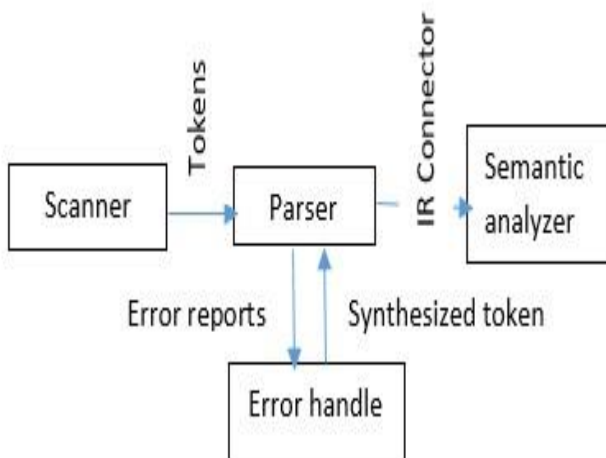


Fig 4: parse information flow

SEMANTIC ANALYSIS

This is the third stage in a compiler construction. Semantic analysis check for semantic errors in the parse tree produced by the syntax analyzer [6]. Examples of semantic errors are data compatibility (data type), undeclared variable use and many more.

INTERMEDIATE CODE GENERATOR

In this phase, an intermediate code of the machine-oriented is generated. It represents a program for some abstract machine [6]. The intermediate code is between a program written in human-oriented and machine-oriented.

CODE OPTIMIZER

The intermediate code generated in the previous stage is been optimized in this stage. The structure of the tree that is generated by the parser can be rearranged to suit the needs of the machine architecture to produce an object code that runs faster [2]. The optimization is achieved by removing unnecessary lines of codes.

CODE GENERATOR

Code generator is the last phase of a compiler construction process. The code generator uses the optimized representation of the intermediate code to generate a naïve machine code. This stage depend on the machine architecture.

3. CONCLUSIONS

This paper explains what a compiler is and gives an overview of the steps involved in translating a programing language into object code. A compiler translate source code into object without tempering with the meaning of the source code. The steps involved in translating a language are six namely; lexical, syntax, semantic, intermediate representation, code optimizer and code generator. Each of this phases perform a single task.

REFERENCES

[1]. De Oliveira Guimarães, J. (2007). *Learning compiler construction by examples. ACM SIGCSE Bulletin, 39(4), 70.* doi:10.1145/1345375.1345418

[2]. Guilan, D., Suqing, Z., Jinlan, T., & Weidu, J. (2002). *A study of compiler techniques for multiple targets in compiler infrastructures. ACM SIGPLAN Notices, 37(6), 45.* doi:10.1145/571727.571735

[3]. Jatin Chhabra, Hiteshi Chopra, Abhimanyu Vats (2014). *Research paper on Compiler Design.* International Journal of Innovative Research in Technology (IJIRT), Volume 1, Issue 5

[4]. Zelkowitz, M. V. (1975). *Third generation compiler design. Proceedings of the 1975 Annual Conference on - ACM 75.* doi:10.1145/800181.810332

[5]. Rudmik, A., & Lee, E. S. (1979). *Compiler design for efficient code generation and program optimization. Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction - SIGPLAN '79.* doi:10.1145/800229.806962

[6]. Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). *Modern Compiler Design.* doi:10.1007/978-1-4614-4699-6

[7]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, *2nd edition, Addison Wesley*, August 31, 2006, ISBN-13: 978-0321486813

[8]. Koskimies, K., Räihä, K.-J., & Sarjakoski, M. (1982). *Compiler construction using attribute grammars. Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction - SIGPLAN '82.* doi:10.1145/800230.806991

[9]. Noonan, R. E. (1986). *Compiler construction using modern tools. Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education - SIGCSE '86.* doi:10.1145/5600.5697

[10]. Demaille, A., Levillain, R., & Perrot, B. (2008). *A set of tools to teach compiler construction. Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education - ITiCSE '08.* doi:10.1145/1384271.1384291

[11]. Li, H., Hu, C., Zhang, P., & Xie, L. (2016). *Modular SDN Compiler Design with Intermediate Representation. Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16.* doi:10.1145/2934872.2959061

[12]. Chen, H., Ching, W.-M., & Hendren, L. (2017). *An ELI-to-C compiler: design, implementation, and performance. Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2017.* doi:10.1145/3091966.3091969