Overview of Compiler Design

Vikash Chauhan[1], Vineet Patwal[2], Dovkush[3]
[1,2,3] *B. tech students Dronacharya College of Engineering, Gurgaon, Haryana*

***Abstract*— Research in compiler construction has been one of the main research areas in computer science. Researchers in this domain try to understand how a computer system and computer languages associates. 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 changing the meaning of the program. This paper aims to explain what a compiler is and give an overview of the stages involved in translating computer programming languages.**

***Index Terms:* compiler, assembler, phases of a compiler, analysis, synthesis, types of a compiler.**

INTRODUCTION

Assembly or high-level languages are the languages used to write a program. However, a computer system understands neither 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. Examples of such languages are Java, C#, C 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. This object code is sometimes called machine code or target code. A compiler is a computer system software that translates source code into an intermediate code which afterwards transformed into target code without changing the meaning of the source code. The result of this transformation must be efficient and optimized in terms of time and space. The interface between a computer programmer and a computer system is the compiler and the operating system. A compiler detects errors 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. The only detected error during compilation processes is the syntax error.

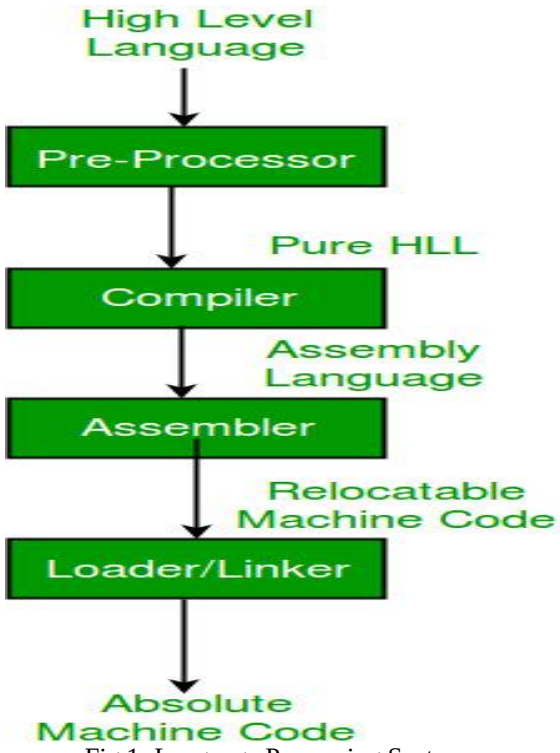

Fig 1: Language Processing Systems

- High-Level Language: - If a program contains #define or #include it is called high-level language (HLL). They are human readable but not for machines.
- Pre-Processor: - The pre-processor removes all the '#' directives by including state that is a combination of machine instructions and some other data required for the execution.
- Assembly Language – It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- Assembler – For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of the assembler is called an object file. It translates assembly language to machine code.

- Relocatable Machine Code – It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate with the program movement.
- Loader/Linker – It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message. Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

PHASES 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. 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. The phases of a compiler consist of six phases. These phases can be regrouped into two major categories –

1.1 Analysis
1.2 Synthesis

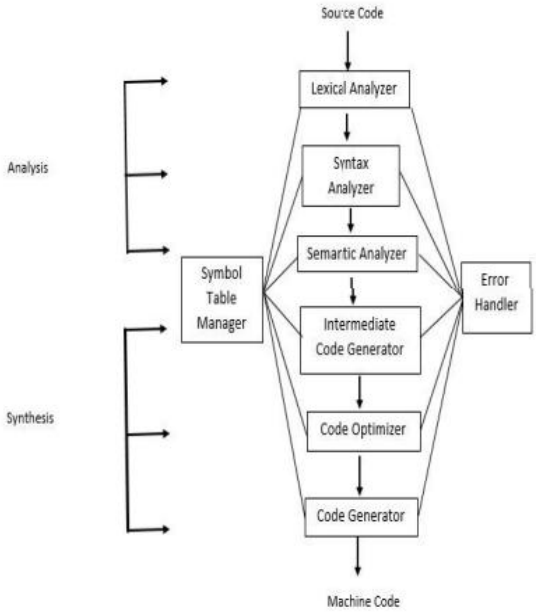

Fig 2: Block Diagram of Compiler

2.1 ANALYSIS:
Analysis is further subdivided into three subparts as follows:
1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

2.2 SYNTHESIS:
The output of the analysis part is used here to produce the machine code. This section is also divided into three subparts as follows:
1 Intermediate Code Generation
2 Code Optimization
3 Code Generation

LEXICAL ANALYSIS

Lexical analysis is the first stage of compiler design. In this stage, the source code is scanned to remove any whitespaces or comments. Then, the source code is categorised into tokens (meaningful sequences of lexical item). This stage is also called “scanning”.
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<Token-name, attribute-value>
A lexical analyser either be implement using Regular expression from automata theory and deterministic finite automata (DFA). A Regular expression is used to specify the token while deterministic finite automata are used to recognise the token.

SYNTAX ANALYSIS

Syntax analysis is the second stage of compiler construction. It is sometimes called a “parser or parsing”. It constructs the parse tree. It takes all the tokens produced in first stage one by one and uses Context-free grammar to construct the tree. A context-free grammar CFG notations are used to the syntactic specification of any program. The goal of parser is to determine the syntactical validity of a source string.

There are certain rules associated with the derivation tree.

- Any identifier is an expression
- Any number can be called an expression
- Performing any operations in the given expression will always result in an expression.

For example, the sum of two expressions is also an expression.

- The parse tree can be compressed to form a syntax tree

Syntax error can be detected at this level if the input is not in accordance with the grammar.

SEMANTIC ANALYSIS

Semantic Analysis is the third stage of compiler construction. It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.

INTERMEDIATE CODE GENERATION

This is the fourth stage of compiler design. In this phase, an intermediate machine-oriented code is generated. It represents a program for some abstract machine. The intermediate code is between a program written in human-oriented and machine-oriented.

CODE OPTIMIZER

This is the fifth stage of compiler design. 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. The optimization is achieved by removing unnecessary lines of codes.

CODE GENERATOR

This is the sixth stage of compiler design. 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 machine code. This stage depends on the machine architecture.

TYPES OF COMPILERS

1. Cross Compilers: - They produce an executable machine code for a platform but, this platform is not one on which the compiler is running.
2. Bootstrap Compilers: - These compilers are written in a programming language that they have to compile.
3. Source to source/trans compiler: - These compilers convert the source code of one programming language to the source code of another programming language.
4. Decompiler: - It is just the reverse of the complier; it converts the machine code into high level language.

FEATURES OF A COMPILER

Features of a compiler are as follows:

- Compilation speed
- Good error detection
- Speed of machine code
- Checking the code correctly Grammarly
- The correctness of machine code

REFERENCE

[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

[6] Ross, D. T. [1967]. The AED free storage package. Communications of the ACM, 10(8):481492.

[7] Rutishauser, H. [1952]. Automatische Rechenplanfertigungbei Programm-gesteuerten Niklaus Wirth This is a slightly revised version of the book published by Addison-Wesley in

1996ISBN 0-201-40353-6Zürich, November 2005.
[8] Aho, Alfred V. and Ullman, Jeffrey D. [1972]. The Theory of Parsing, Translation,
[9] Aho, Alfred V. and Ullman, Jeffrey D. [1977]. Principles of Compiler Design.Addision.