*International Journal For Technological Research In Engineering*
*Volume 8, Issue 3, November-2020* *ISSN (Online): 2347 - 4718*

# ION - C Compiler

[1]Avi Gupta, [2]Anshul Hudda, [3]Pawan Kumar Rai, [4]Shubham Jadon, [5]Mrs. Charul Dewan
[1,2,3,4] Students (B.Tech 7th SEM), [5]Professor
Department of Information Technology
Dr. Akhilesh Das Gupta Institute of Technology & Management, New Delhi, India

***Abstract-* *A compiler is a software or program that converts high level language or source code to low level language or target code through a number of stages where each stage performs its part and makes a code or data structure along with code to be fed to the next stage. The stages of a compiler are usually divided into two phases: Analysis phase and Synthesis phase. The Analysis phase takes the source code and produces an intermediate representation. It consists of Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and Intermediate Code Generator. Each plays a different role like Lexical analyzer divides the program into "tokens", Syntax analyzer recognizes "sentences" in the program using syntax of language and Semantic analyzer checks static semantics of each construct and Intermediate Code Generator generates "abstract" code. And the synthesis phase takes intermediate code from the Analysis phase and generates equivalent target programs. It consists of Code Optimizer and Code Generator where Code Optimizer optimizes the abstract code, and the final Code Generator translates abstract intermediate code into specific machine instructions. The main aim of this paper is to go through all these stages and try to review the underlying principles of these stages and to understand the importance of these stages in building modern compilers as we have come a long way in technological advancements but we still use very old principles to develop a compiler. The final result of this paper is to provide a general knowledge about compiler designing and understanding the development of all the stages used in compiler designing.***

## I. INTRODUCTION

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical or electrical device which is controlled by the help of software. Hardware can only understand data in the binary form or electronic charge, so to instruct it we need to write code in binary form which is very difficult and even impossible for a human programmer to do. Which is why we have compilers, instead of forcing users to write code in binary we make another program that converts code written by the programmer to some other low level language or machine code that the machine can understand and perform the desired task? This translation from source code to machine code requires considerable effort and follows complex rules. The first implemented compiler was developed by Grace Hopper, who also coined the term "compiler" referring to her A-O system which was not even a complete compiler.

It functioned as a loader and linker. Like any other software the compiler also needs to be written in some language, there are benefits from implementing a compiler in a high-level language. Also, a compiler can be self-hosted means it is written in the programming language it compiles. This problem of developing a compiler in the same language it is compiling is also called Bootstrapping problem, i.e. the first such compiler for a language must be either handwritten machine code or compiled by a compiler written in another language, or compiled by running the compiler in an interpreter. Also, different compilers traverse the source code different times, to be able to complete its translation to the machine code. The compiler that reads the source code only one time is called a Single Pass Compiler while the compiler that reads source code multiple times is called Multiple Pass Compiler.

## II. STRUCTURE OF COMPILER

A compiler can be divided broadly into two phases based on their functioning: Analysis Phase and Synthesis Phase.

**Analysis Phase:** It is also known as the front-end of the compiler. The analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase of the compiler reads the input program (of high language code) divides it into core parts and then checks for lexemes, grammar and syntax errors in the code. This phase produces an intermediate representation of the high level language code and symbol table, which is then fed to the Synthesis phase as input. It consists of Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and Intermediate Code Generator.

**Synthesis Phase:** This phase produces the target program or representation or code with the help of intermediate source code representation and symbol table. It constitutes a code optimizer and code generator. It is also called the back-end of compilers.

Generally, when designing a compiler all the stages of a compiler like Lexical analysis, etc. are grouped into these two phases for the sake of code reusability and simplicity. It is just a way of dividing the large code base of the compiler. So, the compiler is just a series of stages. All the stages have their own representation of the input code or high level code, every stage gets input code, processes it and passes to the subsequent stage. So, now let's understand the stages of a compiler.

**Stages of compiler are:**

**Lexical Analysis:** It is the foremost phase of the compilation. In this the source code is scanned as a stream of characters and converted to meaningful lexemes which are represented as tokens (<token-name, attribute-value). LEX is a popular Lexical Analyzer used with the YACC parser.

**Syntax Analysis:** It is also called parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). Here, token arrangements are compared with the source code grammar, i.e. the parser verifies that the expression made by the tokens is syntactically correct. YACC is one of the most popular parsers used in compiler design.

**Semantic Analysis:** Semantic analysis checks whether the parse tree constructed follows the principles of language. For example, assignment of values should be between the compatible data types, and adding string to an integer. Also, it keeps track of identifiers, their types and expressions like checking if identifiers are declared before use or not etc. The semantic analyzer gives an annotated syntax tree as an output.

**Intermediate Code Generation:** Following semantic analysis the compiler produces an intermediate code of the input code for the target machine. It acts as a program for some abstract machine. It is intermediate among the high-level language and the machine language. This intermediate code must be produced in such a way that it makes it easier to be translated into the target machine code.

**Code Optimization:** The following phase optimizes the intermediate code. Optimization can be presumed as something that deletes unnecessary code lines, and rearranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

**Code Generation:**The code generator takes the optimized representation of the intermediate code and portrays it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) relocatable machine code. Sequence of instructions of machine code performs the task same as the intermediate code would do.

All the stages above can access a common data structure called Symbol Table. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

## III. ERROR HANDLING

Another important aspect of compiler design is Error handling. Error can arise because of many causes like design error in program, logic error in program, incorrect data error, etc. The task of the error handler of the compiler is to catch or detect the errors in the source code being compiled and report them to the user. The compiler can only point out errors related to syntax of language, Inconsistent data type of variables, and other things. It cannot detect errors related to runtime. [2] Like memory running out of space or an infinite loop. But with modification in the technology nowadays compilers are getting smarter and smarter and can be even made to point out these errors to some extent too but not completely. Generally following three types of error are encountered during compilation:

**Lexical Error**: It occurs during the Lexical analysis stage of compilation. Lexical error is a series of characters that does not equals the pattern of any token. Lexical phase errors are found during the execution of the code.

It can be a spelling error, exceeding length of identifier or numeric constants, appearance of illegal characters, to remove the character that should be present, to replace a character with an incorrect character, transposition of two characters.

**Syntax Error:** It happens during the Syntax analysis stage of compilation. When an invalid calculation enters into a calculator then a syntax error can also occur. This can be caused by entering several decimal points in one number or by opening brackets without closing them. It can also occur because of Error in structure, Missing operators, Unbalanced Parenthesis, etc.

**Semantic Error:** It happens during the Semantic analysis stage of compilation. Scope and declaration error comes under it. The semantic error can happen when using the wrong variable or using the wrong operator or doing operation in the wrong order. Some common semantic errors are undeclared variables, not matching of actual agreement and It occurs during the Semantic analysis stage of compilation. Scope and declaration error comes under it. The semantic error can happen when using the wrong variable or using the wrong operator or doing operation in the wrong order. Most common semantic errors are using undeclared variables, not matching of actual agreement and wrong operand with formal argument.

Apart from these some compilers are made such that they only give error when the whole execution of program is broken else if there is some error that does not break execution of program then compiler ignores it and places it under warning and notifies user about it.

## IV. COMPILER REQUIREMENTS

Compiler Requirements are the requirements that a compiler needs to satisfy to be considered as a compiler. There are many factors that are taken into consideration to decide whether a compiler is fit for the job of compilation or not. Some most common factors are as follows:

**Correctness:** It is the most important requirement. A buggy compiler is useless. There is no formal way to prove that a compiler is buggy. So, we use exhaustive testing to find bugs. Apart from the correct output of the program it is also tested that the compiler is able to identify errors in the program as desired. Stress on correctness means that we carefully define the semantics of the source language. The target language semantics is produced by the GNU assembler on the lab machines together with the actual machine semantics.

**Efficiency**: Efficiency of the generated code and also efficiency of the compiler itself are important considerations. As if a compiler takes too much time to compile code then it is also useless. Practically the main aim of software is to finish their tasks in the smallest of time so that the CPU can be free for other tasks. That's why efficiency is at the heart of compiler design.

**Interoperability**: Programs don't run in isolation, but are linked with library code before they're executed, or are going to be called as a library from other code. This puts additional responsibility on the compiler, which must respect certain interface specifications.

**Usability**: It also plays a very important role in compiler design as a compiler interacts with the programmer primarily when there are errors in the program. As such, it should give helpful error messages. Compilers should also be directed to generate debug information with executable code in order to help users debug runtime errors in their program.

## V. CONCLUSION

The designing of a compiler is a complex task but it can be achieved if one starts by developing individual stages that are defined to make compiler design easier. The huge use of high level languages tells the story of success of compilers. But with the advancements in technology like multiprocessor CPU, there is a need of improving efficiency of compilers even more so that they can also leverage the power of multiprocessing and multithreading also compiler optimization must bridge this widening gap. Compiler fundamentals are well understood now, but deciding where to use what optimization has become very difficult over the past few decades.

Even though the compiler field has changed the environment of computing, major compilation problems remain, whilst new challenges (such as multi-core programming) have emerged. The unresolved compiler challenges (such as the method to increase the abstraction level of parallel programming, develop robust and secure software, and verify the software stack) are of great practical importance and rank among the foremost intellectually challenging problems in computing today.

To solve problems like these, the compiler field must develop the technologies that enable more of the development in this field compared to the past 50 years. Computer science instructors must attract a number of the good students to the compiler field by showing them its deep intellectual foundations, highlighting the broad applicability of compiler technology to several areas of computing and giving them a chance to create some change in the world. Some challenges in this field (like lack of powerful and flexible compiler infrastructures) can be solved only through effort of complete community. Funding organizations and industry must be made aware of the significance and complexity of the challenges so that they can willingly invest long-term financial and human resources toward finding solutions.

## REFERENCES

[1]. Mary Hall, David Padua, Keshav Pingali. Compiler Research: The Next 50 Years. Communications of the ACM,Vol. 52 No. 2 (February 2009)

[2]. Aastha Singh, Sonam Sinha , Archana Priyadarshi. Compiler Construction. International Journal of Scientific and Research Publications, Volume 3, Issue 4, (April 2013)

[3]. Ch. Raju , Thirupathi Marupaka , Arvind Tudigani. Analysis of Parsing Techniques & Survey on Compiler Applications. International Journal of Computer Science and Mobile Computing Vol.2 Issue. 10, October- 2013