American Journal of Engineering Research (AJER) 2019
American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN : 2320-0936
Volume-8, Issue-12, pp-105-114
www.ajer.org
Research Paper Open Access

# A Study on Language Processing Policies in Compiler Design

Md. Alomgir Hossain[1], Rihab Rahman[2], Md. Hasibul Islam[3], Mahabub Azam[4]
[1]*Senior Lecturer, Department of Computer Science and Engineering*
[2]*Student, Department of Computer Science and Engineering*
[3]*Student, Department of Computer Science and Engineering*
[4]*Student, Department of Computer Science and Engineering*
*IUBAT–International University of Business Agriculture and Technology*
*4, Embankment Drive Road, Sector-10, Uttara Model Town, Dhaka-1230*

**ABSTRACT:** *A compiler translates the source language code into a target language code. This translation takes put through a number of stages. So, we can define the compiler as collection of many stages or passes, where every pass performs a single task. This Research paper is about brief information of compiler on how the source program gets evaluated and translated, from which sections source code has to pass and parse in order to generate target code as output. Besides, this paper will also explain the concept of pre-processors, translators, linkers and loaders and procedure to generate target code.*

***KEYWORDS:*** *compiler, lexical, parsing, syntax, token, sentinels.*

Date of Submission: 29-11-2019 Date of Acceptance: 16-12-2019

## I. INTRODUCTION

Whenever we create a source code and start the process of evaluating it, computer only shows the output and errors only if occurred. We don't know the actual process behind it. In this research paper, the exact procedure behind the compilation task and step by step evaluation of source code are explained.[2] Compiler is a program which converts high level programming language into low level programming language or source code into machine code. Compilers and operating systems constitute the basic interfaces between a programmer and the machine. Compiler is a program which converts high level programming language into low level programming language or source code into machine code [3]. This paper also explains the concept of pre-processors, translators, linkers and loaders and procedure to generate target code [2].

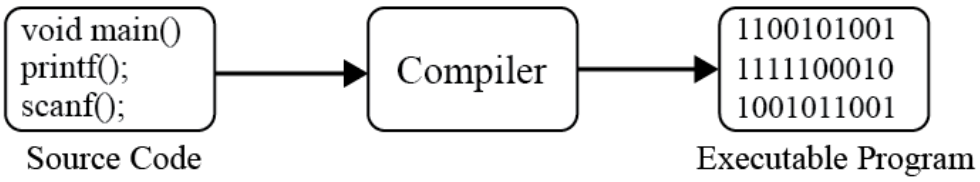



**Figure – 1: Position and task of compiler**

Here, we will discuss sequentially how this paper is organized. This paper covers the literature review in section II, Research Methodology in section III, Language Processing System in section IV, Pre-Processors in section V, Compilers in section VI, Assembler in section VII, Linker in section VIII, Loader in section IX, Algorithm of Compiler in section X, Phases of A Compiler in section XI and last all about Conclusion in section XII and References are in section XIII.

## II. LITERATURE REVIEW

Computer programs are formulated in a programming language and specify classes of computing processes. Computers, however, interpret sequences of particular instructions, but not program texts. Therefore, the program text must be translated into a suitable instruction sequence before it can be processed by a computer. This translation can be automated, which implies that it can be formulated as a program itself. The translation program is called a compiler, and the text to be translated is called source code [9]. A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language through a number of stages. Starting with recognition of token through target code generation provide a basis for communication interface between a user and a processor in significant amount of time [3]. The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. We shall be concerned with the engineering of compilers covering organization, algorithms, data structures and user interfaces. Programming languages are tools used to construct formal descriptions of finite computation. Each computation consists of operations that transform a given initial state into some final state [9]. A compiler compiles a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed. One of advantage of using a high-level level language can be the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines [6].

## III. RESEARCH METHODOLOGY

In this paper we will discuss about the compiler, its language processing system, assembler, linker, loader and differences between assembler & compiler and linker & loader. We will also discuss about different phases of compilers along with their code compilation process in detail. Therefore, this research will follow qualitative research method. Here we will collect our data by doing background study. Then we will analyze our data to sort out our required information then we will discuss each of them with proper illustration. Then we will reveal the steps of language being processed from high level to machine level. Finally, we will draw our conclusion about the efficiency of the compiler to improve for better compilation power.

## IV. LANGUAGE PROCESSING SYSTEM

In language processing system the source program is first preprocessed preprocessors. The modified source program is processed by compiler to form target assembly code which is then translated by assembler to generate relocatable object codes that are processed by linker and loader to generate target program. We write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System [10].

It can be described for C compiler as:

- User writes a program in C language Source Code high - level language.
- The C compiler, compiles the program and translates it to assembly program low - level language.
- An assembler then translates the assembly program into machine code object.
- A linker tool is used to link all the parts of the program together for execution executable machine code.
- A loader loads all of them into memory and then the program is executed.

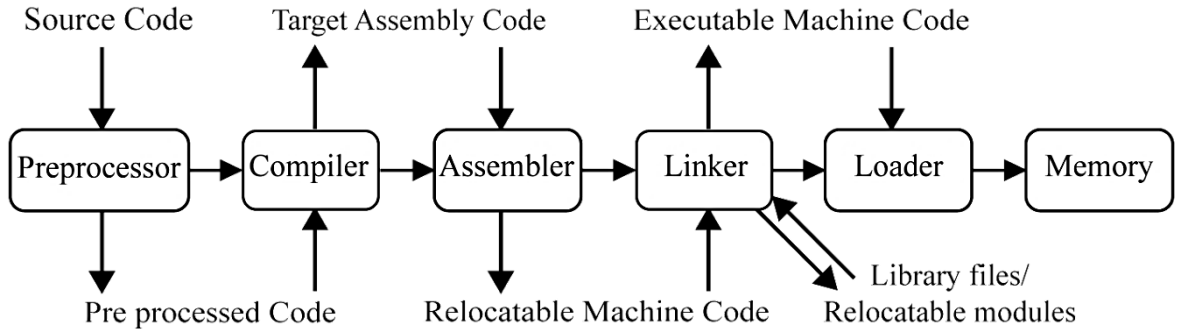



**Figure – 2: A language processing system**

## V. PRE-PROCESSORS

A computer program called Pre-Processor that manipulates its input data for generating output which is also used as input in many other compilers or programs. Output of the pre-processor is perfect high-level language from normal high-level language. Perfect high-level language indicates that has file inclusion and macros in the program. Macros refers some specific set of instructions that can be used one after another in a same program. Pre-processor do this macro pre-processing work or task. We can contemplate this one as a part of compiler, which can provide compilers input. It works with augmentation, file inclusion, language extension and most importantly macro-processing [8]. Function of preprocessor: Make sure all files are loaded into the compiler. By this we mean to load all headers, referenced assemblies or source files into one virtual file.

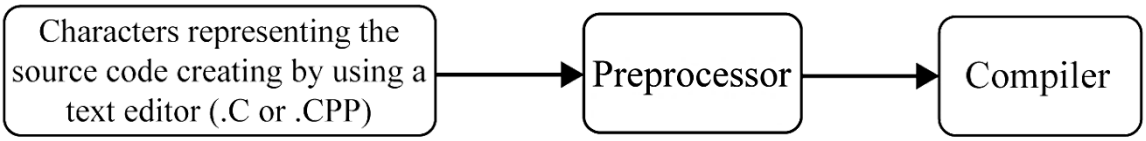



**Figure – 3: Activities of pre-processor**

## VI. COMPILERS

Compiler is a sort of translator which can be used to translate higher level language to lower level language. Compiler generate error at first by processing whole program. Compiler generates intermediate code in order to get target code. Errors are displayed after checking the full program. Compiler shows the errors after finishing the compilation, if there are any errors in source code. Compiler can successfully compile the source code only after removing all the errors before starting compilation [8]. Turbo C, Borland, Code Blocks are the example of Compiler.

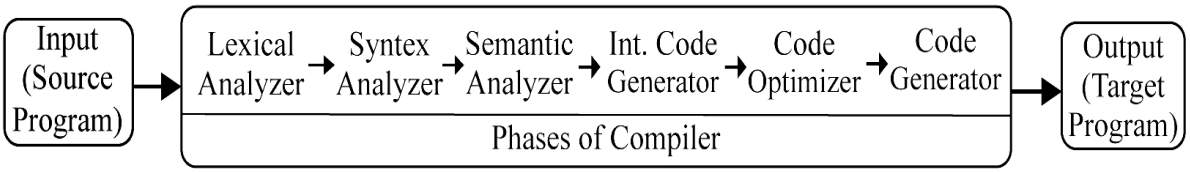



**Figure – 4: Activities of compiler**

## VII. ASSEMBLER

An assembler is also a sort of translator which can converts program that written in assembly language into machine code. A software developers and application developers can access, manage and operate a computer's hardware components and architecture by an assembler. We can say that an assembler is also a type of compiler in assembly language.

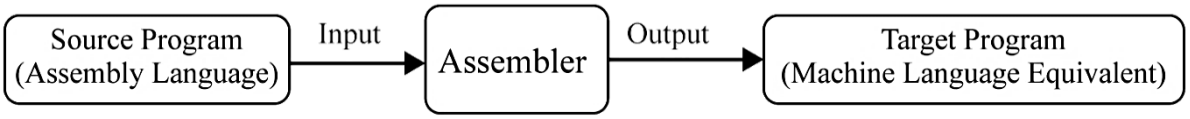



**Figure – 5: Activities of assembler**

## VIII. LINKER

Linker combines two or more separate object programs. It combines target program with other library routines. Linker links the library files and prepares single module or file. Linker also solves the external reference. Linker allows us to create single program from several files [2]. It links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by this program called Linker. Process for producing a file:

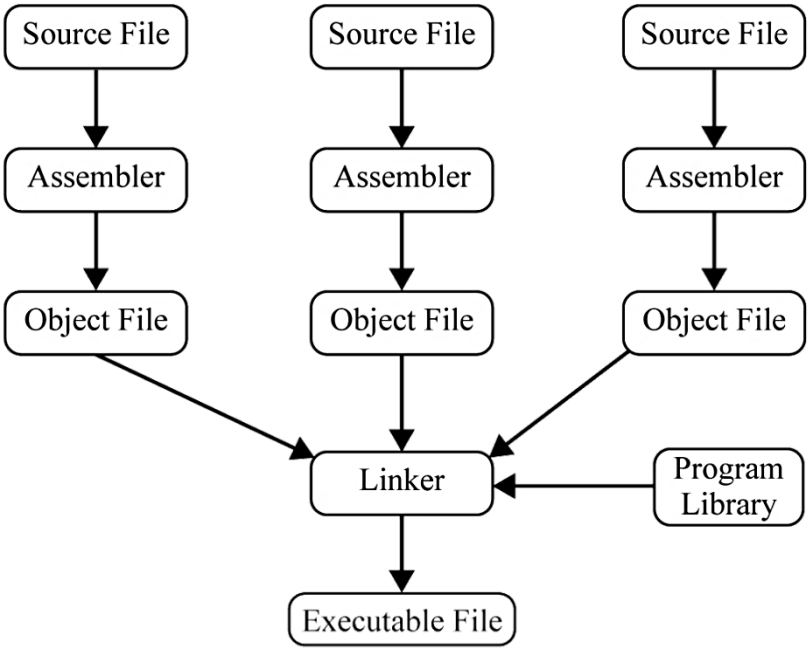



**Figure – 6: Executable file producing process**

## IX. LOADER

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program instructions and data and creates memory space for it. A loader is the part of an operating system that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution [11].

## X. ALGORITHM OF COMPILER

Algorithm of compiler denotes the activity of compiler to translate the high-level language to machine executable language. There is not any single phase for translating the high-level language. It passes through several phases where the language is processed as per their phase wise rules. By combining all of them we can propose an algorithm for compiler to work on. The algorithm can be like:

| Name of Phase | Steps of Phase |
|---|---|
| Lexical Analyzer | Step - 1: Taking input as Character Stream.<br>Step - 2: Reading the character until the next token.<br>Step - 3: Produce output as token. |
| Syntax Analyzer | Step - 4: Taking input as token.<br>Step - 5: Pass it to the Syntax analyzer, and syntax analyzer will create a syntax tree for the token.<br>Step – 6: Produce a syntax tree as output. |
| Semantic Analyzer | Step - 7: Taking input the syntax tree of a token.<br>Step - 8: Check whether the tree is semantically correct or not.<br>Step - 9: Produce a semantically correct syntax tree by type checking, level checking, flow control checking. |
| Intermediate Code Generation | Step - 10: Take input from syntax tree.<br>Step - 11: Produce intermediate code step by step. |
| Code Optimization | Step - 12: Take input from intermediate code.<br>Step - 13: Minimize the long code to short.<br>Step - 14: The temporary location also reduced here. |
| Code Generation | Step - 15: Take input from code optimizer as optimized code.<br>Step - 16: Process the task by some specialized instructions.<br>Step - 17: Get the targeted machine code. |

## XI. PHASES OF A COMPILER

***Lexical Analysis***

The traditional meaning of the word "lexical" is "pertaining to words". In the sense of programming languages, words are objects like variable names, numbers, keywords etc. Lexical analysis makes life easier for the subsequent syntax analysis phase. This is usually named tokens. It is usually not terribly difficult to write a lexer by hand, we first read past initial white-space, then you, in sequence, test to see if the next token is a keyword, a number, a variable or whatnot. a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by lexer generators, which transform human-readable specifications of tokens and white-space into efficient programs.

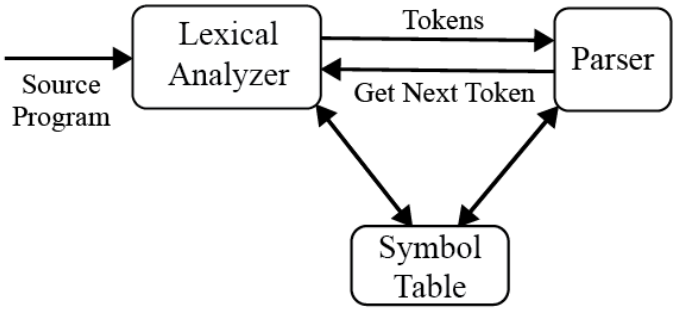
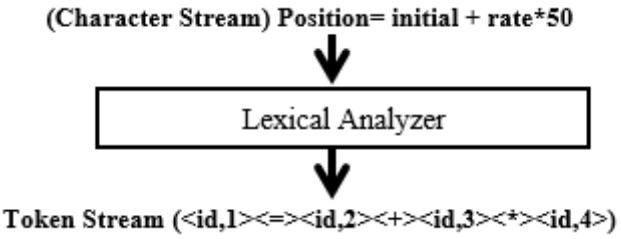



**Figure – 7: Parser**

The LA is the first phase of a compiler. Its main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

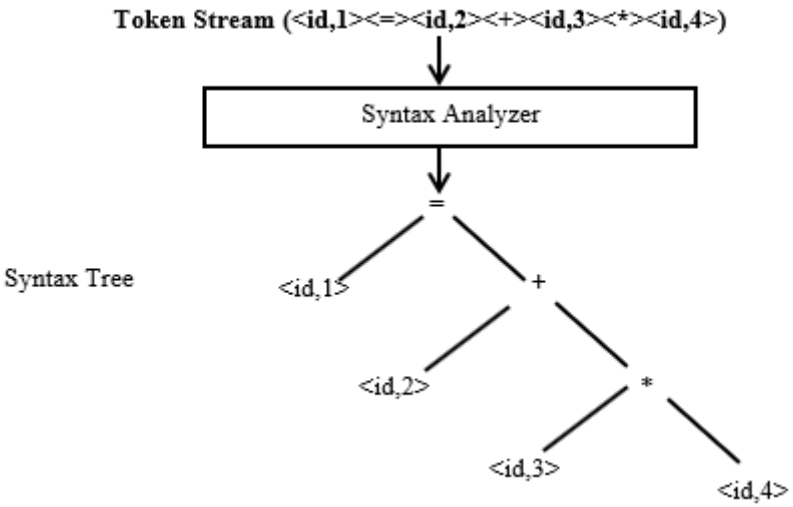
Upon receiving a 'get next token' command form the parser; the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

Steps:
Step 1: Taking input as Character Stream
Step 2: Reading the character until the next token
Step 3: Produce output as token




***Syntax Analyzer***

The Another Name of Syntax Analysis is Parsing. Where lexical analysis splits the input into tokens, then syntax analysis recombines these tokens. It is called the syntax tree of the text. This is a tree structure where the leaves are the token found by the lexical analysis. and if the leaves are read from left to right, the sequence is the same as in the input text. But in syntax tree, it is important that how the interior nodes of tree are labelled. The parser should report any syntax errors in an intelligible fashion.
Steps:
Step 1: Taking input as token
Step 2: Pass it to the Syntax analyzer, and syntax analyzer will create a syntax tree for the token.
Step 3: Produce a syntax tree as output




The two types of parsers employed are:
1. Top down parser: which build parse trees from top(root) to bottom(leaves)
2. Bottom up parser: which build parse trees from leaves and work up the root

***Top-Down Parsing***

Parser is a program that performs syntax analysis. It takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look ahead and an approach called top-down parsing without backtracking. Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down [11].

***Bottom-up Parsing***

It checks to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Generally, Build the parse tree from leaves to root. Bottom up parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the rightmost derivations of w in reverse [8].

***Semantic Analysis***

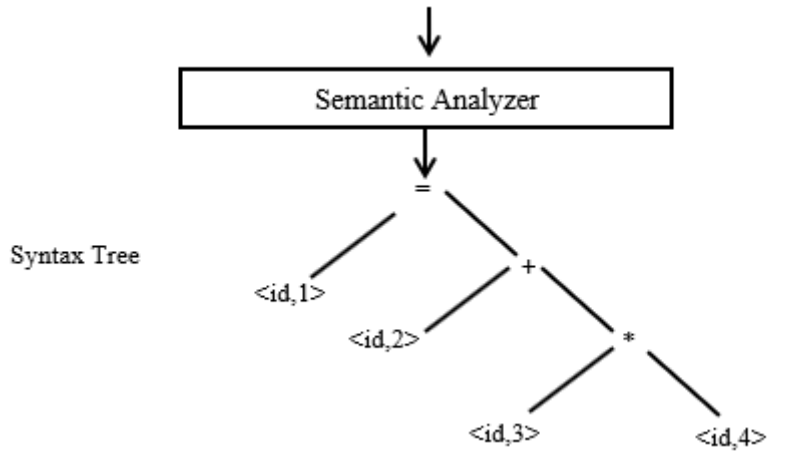
Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

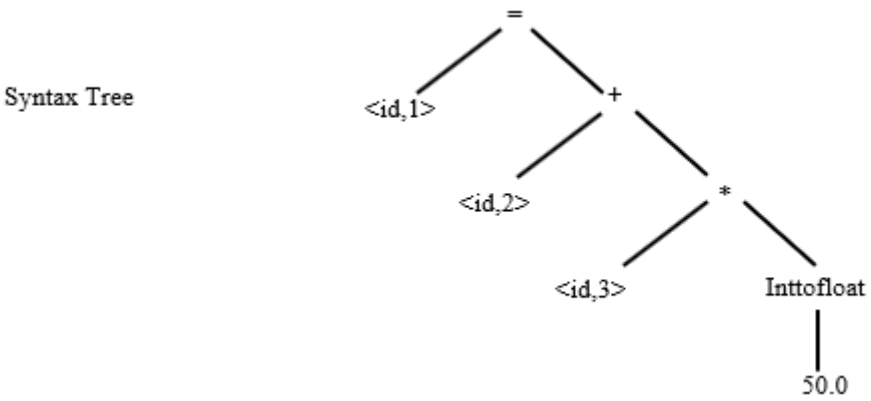
Steps:

Step 1: Taking input the syntax tree of a token

Step 2: Check whether the tree is semantically correct or not

Step 3: Produce a semantically correct syntax tree by type checking, level checking, Flow control checking






***Semantic Analysis Typically Involves***

- Type checking – Data types are used in a manner that is consistent with their definition
- Label Checking – Labels references in a program must exist.
- Flow control checks – control structures must be used in their proper fashion

Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known. In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking. The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax. As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism [11]. As representation formalism this lecture illustrates what are called Syntax Directed Translations

***Semantic Analysis in Top Down Parsing***

Imagine we're parsing:

S ® id := E
E ® T E'
E' ® + T E'
E' ® ε
T ® F T'
T' ® * F T'
T' ® ε
F ® id
F ® const
F ® ( E )

We insert the actions

S ® id {*pushid*} := {*pushassn*} E {*buildassn*}
E ® T E'
E' ® + {*pushop*} T {*buildexpr*} E'
E' ® ε
T ® F T'
T' ® * {*pushop*} F {*buildterm*} T'
T' ® ε
F ® id {*pushid*}
F ® const {*pushconst*}
F ® ( E ) {*pushfactor*}

***Intermediate Code Generation***

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code [10].

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code [10].

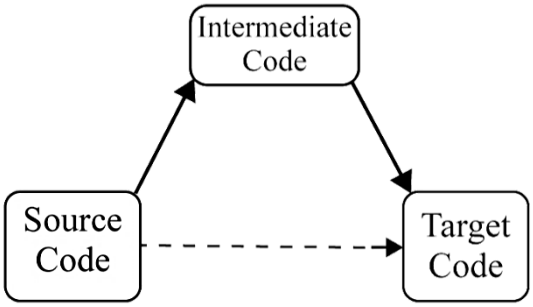
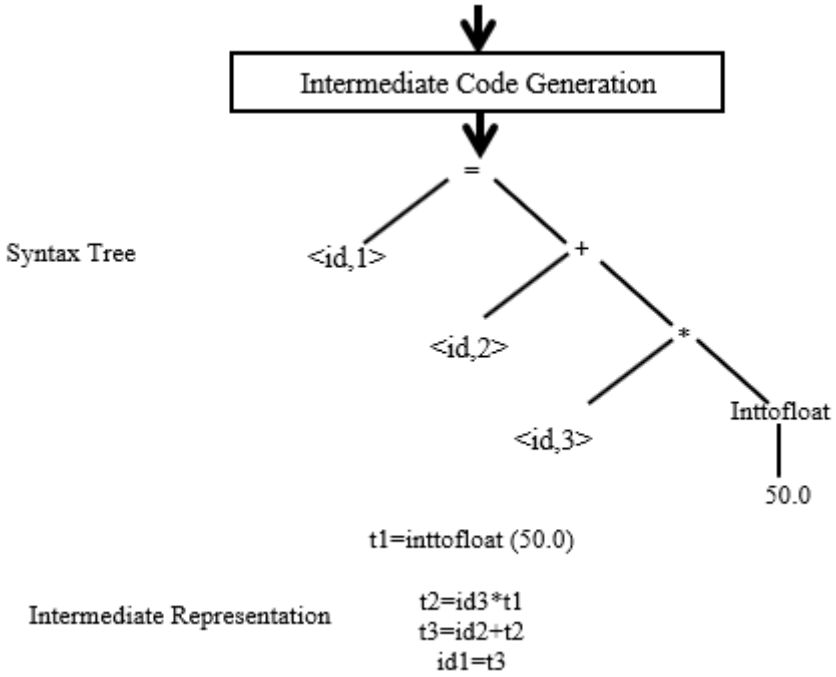



**Figure – 8: Intermediate Code Generation**

Steps:

1. Take input from syntax tree
2. Produce intermediate code step by step




***Code Optimization***

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources like - CPU, memory etc. so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Goals of code optimization: It remove redundant code without changing the meaning of program. Objective:

1. Reduce execution speed
2. Reduce code size

Achieved through code transformation while preserving semantics.

A very hard problem + non-undecidable, i.e., an optimal program cannot be found in most general case.

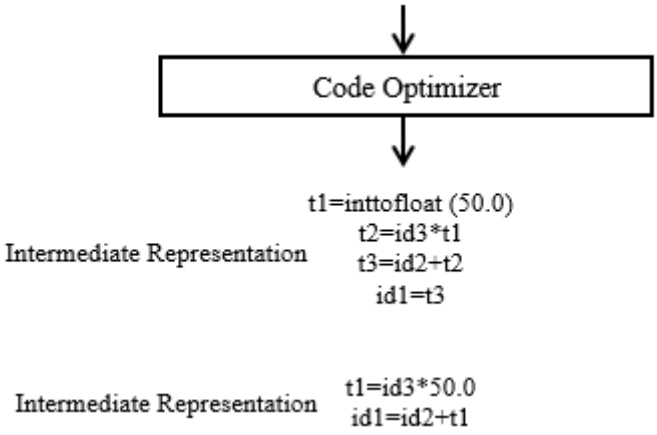
Types of Code Optimization –The optimization process can be broadly classified into two types

1. Machine Independent Optimization – This code optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

2. Machine Dependent Optimization – Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy [12].

Steps:

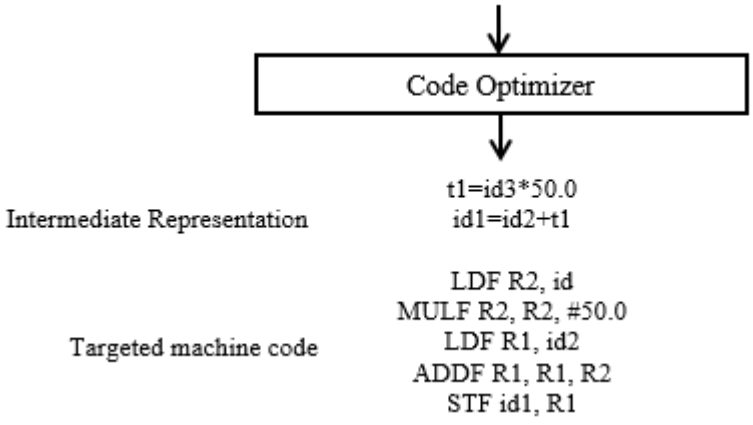
1.Take input from intermediate code
2. Minimize the long code to short
3. The temporary location also reduced here




***Code Generation***

The final phase of compilation is code generation. Optimization process is applied on the code through post code generation. The compiler generates the code, which is an object code of the lower-level programming language, such as assembly language. The higher-level source code language is transformed into lower-level language and thus come up with lower-level object code. Generally, the input of the code generation consists of parse tree or abstract parse tree. Then it is converted into a linear sequence of instructions [11].

Steps: 1. Take input from code optimizer as optimized code.
2. Process the task by some specialized instructions.
3. Get the targeted machine code.



## XII. CONCLUSION

Compiler is a translator that translate high level source code to target machine code for machine execution. Throughout this whole translation is passes through many phases but the meaning of the code never changes during the time. It basically works as language processing system. Throughout each of the phases it translates sequentially that is shown in the paper. It is illustrated as algorithm here the whole translation process. It combines the assemblers along with this for translation the high-level language though mid-level language to obtain machine language form to execute the instruction. It has two parts called as front end and back end. Front end of compiler includes lexical analyzer, syntax analyzer, and semantic analyzer. On the other hand, back end of compiler includes code generation and optimization. Here we have presented the compiler activates step by step and in future we will work on to enhance the efficiency and effectiveness of compiler in language processing system.

## XIII. REFERENCES


[1]. Neha Pathapati, Niharika W. M., Lakshmishree .C, "Introduction to Compilers", *International Journal of Science and Research (IJSR)*, https://www.Usr.neVarchive/v4i4/SUB153522.pdf, Volume 4 Issue 4, April 2015, 2399 - 2405
[2]. Trivedi, Vishal. Life Cycle of Source Program - Compiler Design. Vol. 5, IJIRCCE, 2017.
[3]. Chhabra, Jatin, et al. Research Paper on Compiler Design. Vol. 5, IJIRT, 2014.
[4]. *Loader Definition | Difference Between Linker and Loader ...* www.sitedownrightnow.com/search/loader-definition.
[5]. "Differences between Interpreters and Compilers." *Differences between Interpreters and Compilers - Wayne__Lu - CSDN*
[6]. Mogensen, Torben AE. *Basics Of Compiler Design*. 2007.
[7]. Engineering College Shri Vishnu, *"Compiler Design Lecture Notes"*
[8]. "Parsing | Set 2 (Bottom Up or Shift Reduce Parsers)." *Geeks for Geeks*, 6 Sept. 2018,
[9]. Singh, Aastha, et al. "Compiler Construction." *International Journal of Scientific and Research Publications*, vol. 3, no. 4, Apr. 2013.
[10]. tutorialspoint.com. "Compiler Design - Intermediate Code Generation." *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm.
[11]. *Compiler Design Tutorials Point" Keyword Found Websites ...* www.keyword-suggest-tool.com/search/compiler design tutorials point
[12]. "Compiler Design | Code Optimization." *GeeksforGeeks*, 4 June 2018, www.geeksforgeeks.org/compiler-design-code-optimization/.


Md. Alomgir Hossain. "A Study on Language Processing Policies in Compiler Design." American Journal of Engineering Research (AJER), vol. 8, no. 12, 2019, pp 105-114