ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 10, Issue 2, pp: (97-106), Month: April - June 2022, Available at: www.researchpublish.com

Front End to Back End- Compiler Design

[1]Mr. Rupak Kumar Gogoi, [2]Mr. Abinash Borah, [3]Ms. Chandrani Borah

[1,2,3] Jorhat Engineering College, Jorhat, India

DOI: https://doi.org/10.5281/zenodo.6735151

Published Date: 25-June-2022

Abstract: **This research article explains how source code is assessed at the Front End and Back End of the compiler and which sections source code must pass and parse in order to generate target code. Furthermore, this paper discusses the concept of Pre-processors, Translators, Linkers, and Loaders, as well as the mechanism for using them and produces the code for the target The focus of this paper is on the concept of Compiler and Compiler Phases.**

Keywords: **Macro, Token, Lexemes, Identifier, Operators, Operands, Sentinel, Prefix, Postfix, IC, IR, Binary program.**

1. INTRODUCTION

When we construct a source code and begin the evaluation process, the computer just displays the output and any faults (if any). We don't know how it works in practice. The full technique underlying the compilation work and step-by-step evaluation of source code are given in this study report. High level languages, Low level languages, Pre-processors, Translators, Compilers, Compiler Phases, Compiler Cousins, Assemblers, Interpreters, Symbol Table, Error Handling, Linkers, and Loaders are all covered.

Fig 1: Phases of Source Program to Target Program

2. HIGH LEVEL LANGUAGES

The source software is written in a high-level language that incorporates natural language components and is easier to write. It is a programming language with a high level of abstraction. It simplifies, simplifies, and clarifies the process of writing source code. High-level languages are substantially closer to English than lower-level languages, and they use the same framework for program coding. Visual Basic, PHP, Python, Delphi, FORTRAN, COBOL, C, Pascal, C++, LISP, BASIC, and other high-level languages are examples.

3. LOW LEVEL LANGUAGES

Low level languages are those that machines can understand directly. It is a programming language with minimal or no abstraction. These languages have been described as being close to hardware. Machine languages, binary languages, assembly level languages, and object code are examples of low level languages.

4. PRE-PROCESSORS

A pre-processor is a computer program that manipulates its input data to produce output that is then used as input to another program or compiler. Pre-processor input is high level languages, and pre-processor output is pure high level languages. A pure high level language is one that includes Macros and File Inclusion in the program. A macro is a set of instructions that can be used repeatedly in a program. Pre-processors handle macro pre-processing. The pre-processor allows the user to include header files that may be required by the program, which is known as File Inclusion. For example: # define PI 3.14 indicates that whenever PI is encountered in a program, it is replaced by the value 3.14.

5. TRANSLATORS

Translator is a program that takes input as a source program and converts it into another form as output. Translator takes input as a high level language and convert it into low level language. There are mainly three types of translators.

1. Compilers
2. Interpreters
3. Assemblers

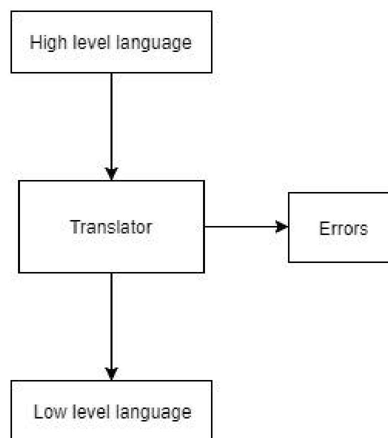


Fig 2: Translator

6. COMPILERS

The compiler reads the entire program at once and generates errors (if occurred). In order to generate target code, the compiler generates intermediate code. Errors are displayed once the entire program has been checked. Borland Compiler and Turbo C Compiler are two examples of compilers. Following the compilation process, the generated target code is simple to understand. The compilation process must be completed quickly. The compilation process is divided into two parts.

[A] Analysis Phase: The analysis phase of the compilation process is machine independent. The primary goal of the analysis phase is to separate the source code into parts and rearrange these parts into a meaningful structure. The meaning

of the source code is determined, and then intermediate code is generated from it. The analysis phase is divided into three sub-phases: lexical analysis, syntax analysis, and semantic analysis.

[B] Synthesis Phase: The machine is in charge of this stage of the compilation process. The intermediate code is translated into a target code that is equivalent. Intermediate code, code optimization, and code creation are the three major sub-phases of the synthesis phase.

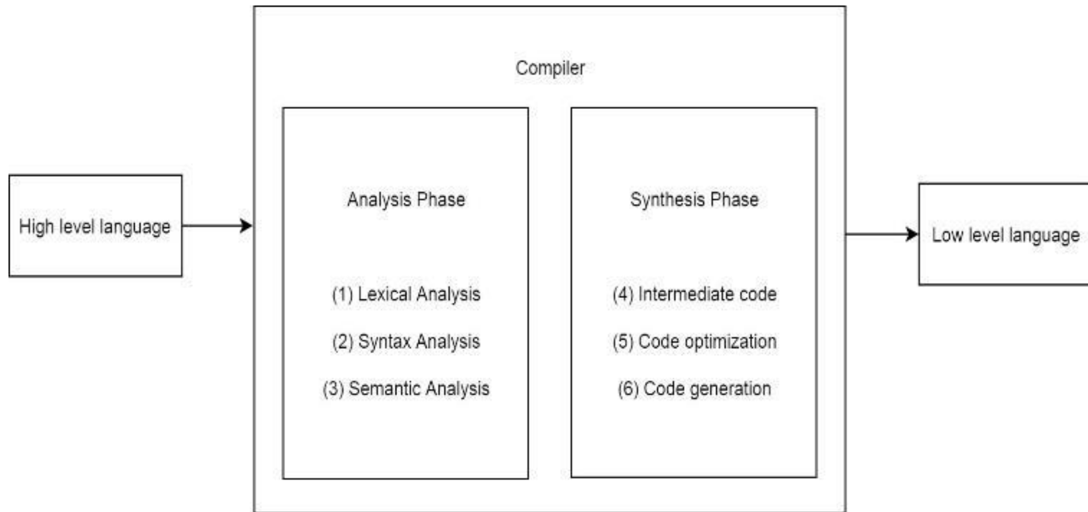


Fig 3: Compiler

7. PHASES OF COMPILER

As previously stated, lexical analysis, syntactic analysis, semantic analysis, intermediate code, code optimization, and code creation are all aspects of the compiler.

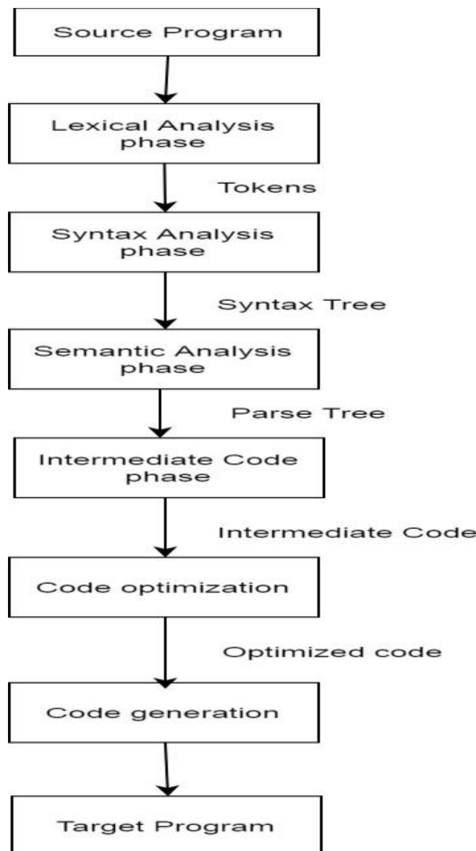


Fig 4: Phases of Compiler

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 10, Issue 2, pp: (97-106), Month: April - June 2022, Available at: www.researchpublish.com

(i) ***Lexical Analysis:*** The initial phase of the compiler is lexical analysis.

Lexical Analysis is also known as Scanning or Linear Analysis.

To begin, the lexical analyzer examines the entire program and divides it into tokens. The string with meaning is referred to as a token. The input string's class or category is described by the token. Identifiers, Keywords, Constants, and so on. Sentinel refers to the end of the buffer or token.

The token is described by a set of rules known as a pattern.

Lexemes are the sequence of characters in source code that correspond to the token pattern. For example *int, i* , *num* etc.

There are two pointers in Lexical analysis they are *Lexeme pointer* and *Forward pointer*. To recognize a token Regular expressions are used to construct Finite Automata. Input is the source code and output is the tokens.

E.g.

Input: x= x + y*z*3

Output: Tokens or table of tokens

=	x
+	y
*	z
	3

(ii) **Syntax Analysis:**

Syntax analysis, also known as syntactical analysis, parsing, or hierarchical analysis, is a type of analysis that examines the structure of a sentence. Syntax is the arranging of words and phrases in a language to produce well-formed sentences. The tokens generated by the lexical analyzer are put together to form a less detailed hierarchical structure known as the syntax tree.

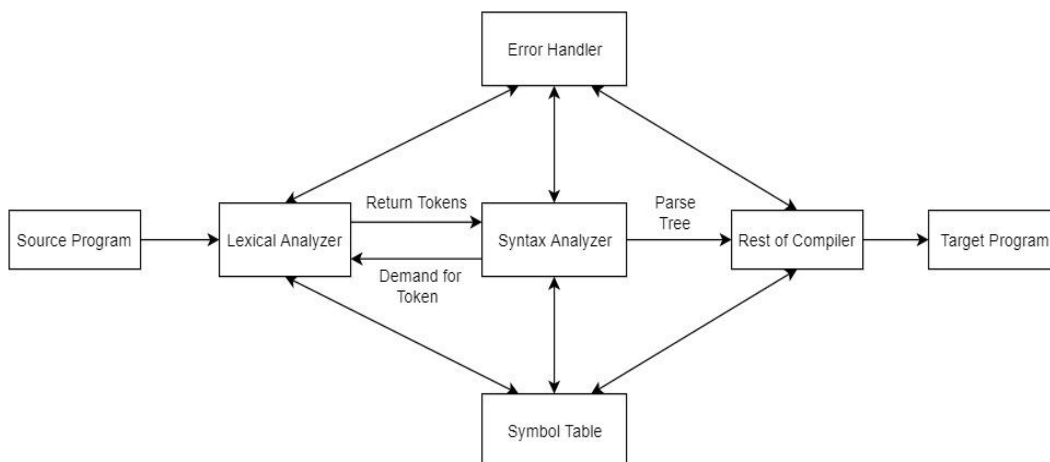


Fig 5: Lexical and Syntax Analyzer

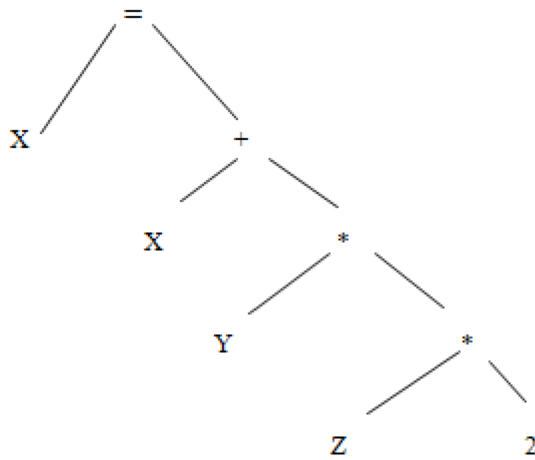
Input is *token* and output is *syntax tree*.

Grammatical errors are checked during this phase. Example: *Parenthesis missing*, *semicolon missing, syntax errors* etc.

For example: Input: *tokens*

=	X
+	Y
*	Z
	3

Output:



(iii) Semantic Analysis:

Semantic analyzer checks the meaning of source program. Logical errors are checked during this phase. Example: *divide by zero, variable undeclared* etc.

Example of logical errors

int a;

float b;

char c;

c=a+b;

Parse tree refers to the tree having meaningful data. Parse tree is more specified and more detailed.

Input is *syntax tree* and output is *parse tree* (syntax tree with meaning).

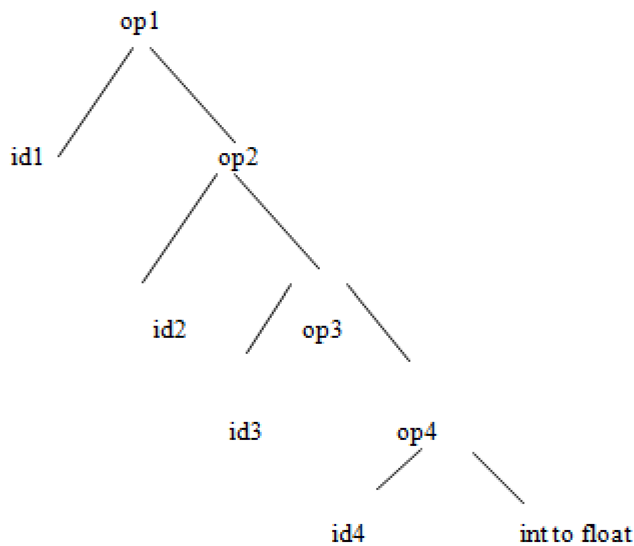


Fig 6: Parse Tree

(iv) **Intermediate Code:**

Intermediate code (IC) is code that sits between high-level and low-level languages, or code that sits between source and target code. The conversion of intermediate code to target code is simple. Intermediate code functions as a bridge between the front end and the back end. Three address codes, abstract syntax trees, prefix (polish), postfix (reverse polish), and other types of intermediate code exist.

Directed Acyclic Graph (DAG) is kind of abstract syntax tree which optimizes repeated expressions in syntax tree.

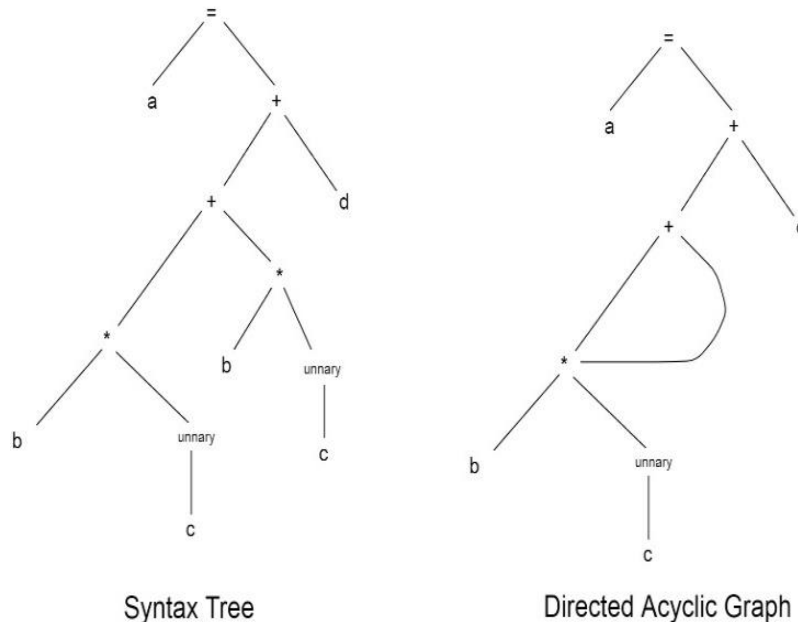


Fig 7

The three-address code, which has no more than three operands, is the most often used intermediate code.

Input: *Parse tree*

Output: Three address code

temp1=int to float(2);

*temp2=id4*t1;*

*temp3=id3*t2;*

temp4=id2+t3;

temp4=id1;

(v) **Code optimization:** To increase intermediate code and execution performance, code optimization is used. It is vital to have code that executes faster or consumes less memory. There are mainly two ways to optimize the code named Front-end (Analysis) and Back-end (Synthesis).

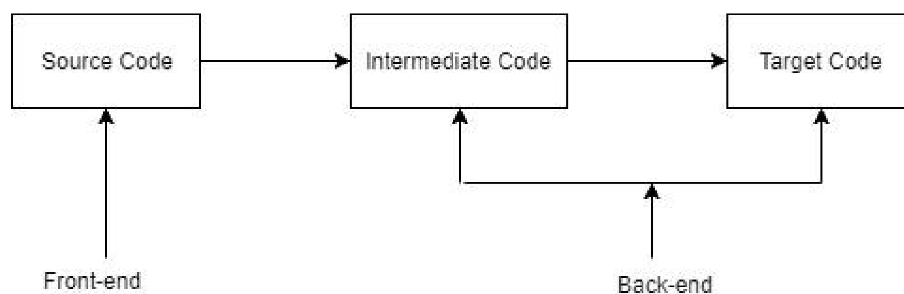


Fig 8: Code Optimization

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 10, Issue 2, pp: (97-106), Month: April - June 2022, Available at: www.researchpublish.com

A programmer or developer can optimize the code in front-end. The compiler can optimize the code on the back-end.

Various strategies for code optimization are listed below.

• Compile Time Evaluation • Constant Folding • Constant Propagation • Common Sub Expression Elimination • Variable Propagation • Code Movement • Loop Invariant Computation • Strength Reduction • Dead Code Elimination • Code Motion • Induction Variables and Strength Reduction.

Input: *Three address code*

Output: : Optimized three address code

*temp1=id4*2.0;*

*temp2=temp1*id3;*

id1=temp2+id2;

(vi) **Code Generation:** The compiler's final process is code generation. The intermediate code is translated into machine language, which is then sent through the pass and parse phases before being optimized.

The properties that are desired during the code generation process are listed below.

• Accuracy • High Quality

• Generate Quick Codes

• Effective utilisation of the target machine's resources

The following are some of the most common code generating issues:

• Memory management • Code generator input • Target programmes • Code generation approaches • Evaluation order selection • Register allocation • Instruction selection, and so on.

Target code which is now low level language goes into linker and loader.

Input: *Optimized three address code*

Output: Machine language.

MOV id4, R1

MUL #2.0, R1

MOV id3, R2

MUL R2, R1

MOV id2, R2

ADD R2, R1

MOV R1, id1

(vii) COMPILERS' COUSINS:

Cousins refers to the typical context in which the compiler operates. There are three main cousins of compiler.

[1] Assemblers [2] Pre-processors [3] Loaders and Linkers

(viii) **Assemblers.**

Assembler is a translator that takes assembly language as input and outputs machine language. The compiler's output is the assembler's input, which is assembly language. Assembly code is a mnemonic for machine code. Names replace binary codes for operations. The assembler generates binary language or relocatable machine code. Assembler employs two passes. One complete scan of the input program is considered a pass.

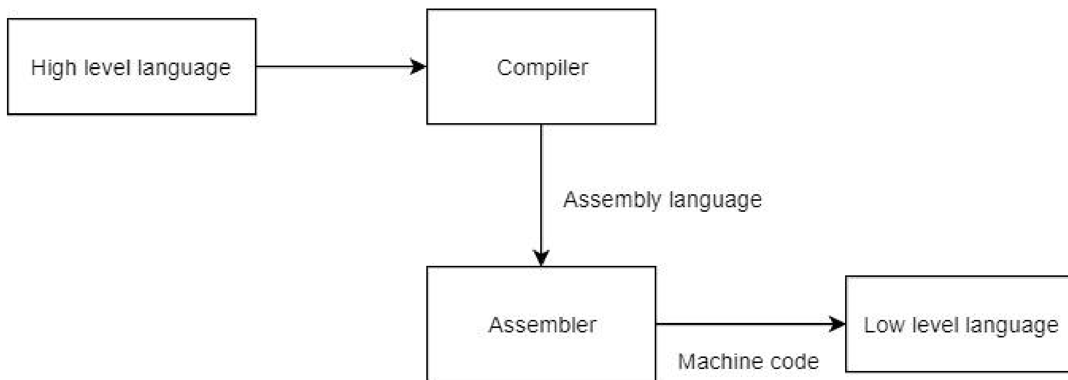


Fig 9: Assembler

(ix) **INTERPRETERS**

The interpreter executes the source code line by line. It takes a single instruction as input, reads it, analyses it, and executes it. If an error occurs, it is displayed immediately. The interpreter is machine-independent and does not generate object or intermediate code because it generates the target code directly. Many languages can be implemented using both compilers and interpreters like BASIC, Python, C#, Java etc.

There are mainly two phases of Interpreter:

1. Analysis phase: The analysis step is divided into three sections: lexical analysis, syntax analysis, and semantic analysis. This stage of the compilation process is not dependent on the computer. The interpreter's analysis phase is similar to the compiler's analysis phase.

2. Synthesis phase: Synthesis is a machine-dependent phase of the compilation process. Because it does not generate intermediate code, the synthesis phase has a sub-phase called code generation (direct execution).

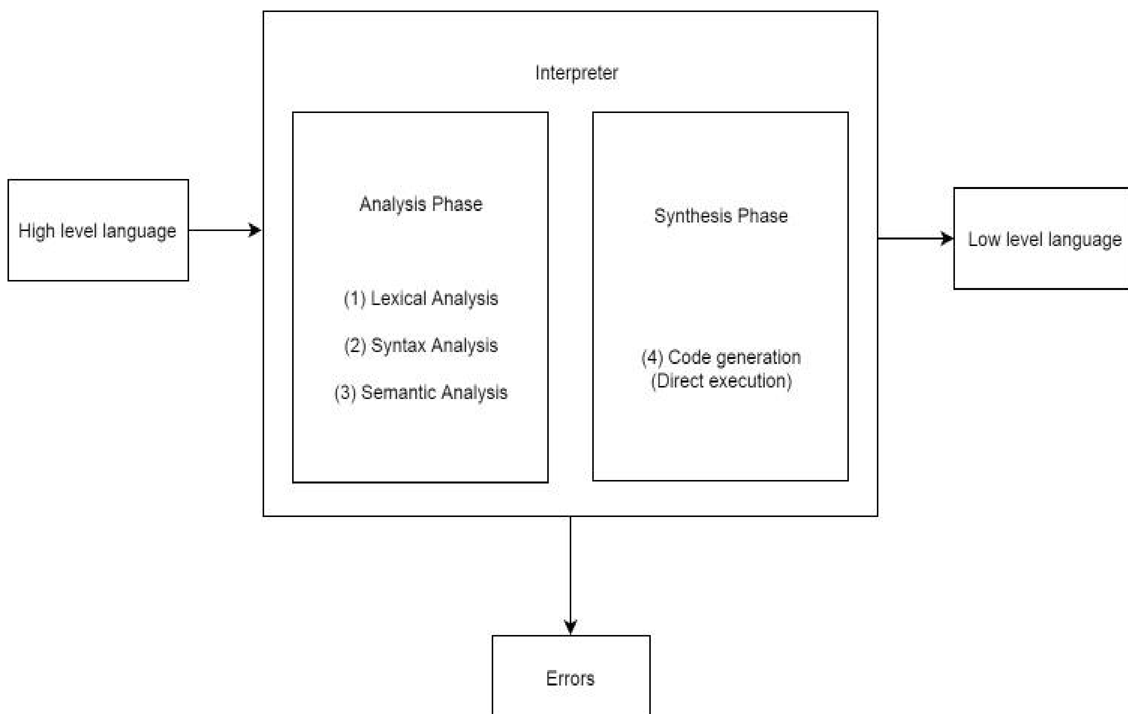


Fig 10: Interpreter

(x) SYMBOL TABLE AND ERROR HANDLING

To store information about attributes, translators such as compilers and assemblers employ a data structure known as a Symbol table. It keeps track of the names that appear in the source code, as well as their properties. The information about

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 10, Issue 2, pp: (97-106), Month: April - June 2022, Available at: www.researchpublish.com

entities such as interfaces, objects, classes, variable names, function names, keyword, constants, subroutines, label name, and identifier is stored in the symbol table.

Every phase of the compiler discovers errors, which must be sent to the error handler, whose job it is to deal with the errors so that compilation can continue. Lexical errors include misspellings, identifier or numeric constant lengths that are too long, and the occurrence of prohibited characters, among other things. Structure mistakes, missing operators, and missing parentheses are examples of syntax errors. Semantic errors include incompatible operand types, undeclared variables, and the mismatch of actual arguments with formal arguments, among other things. Analyzers can employ a variety of ways to recover errors.

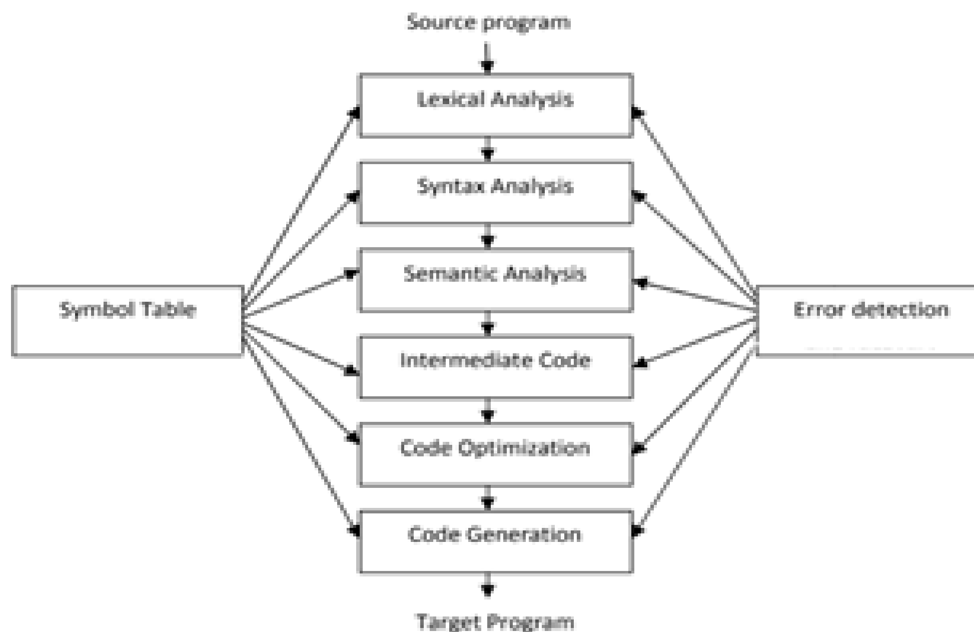


Fig 11: Symbol Table

(xi) LINKERS AND LOADERS

A linker is a program that joins two or more distinct object programs into one. It mixes the target program with functions from other libraries. The linker joins library files together and creates a single module or file. The external reference is also resolved by Linker. We can use Linker to combine multiple files into a single program.

A loader is a utility program that accepts object code and prepares it for execution. In addition, it converts object code to executable code. The initialization of the execution process is referred to as loading. The tasks performed by loaders are listed below.

1. Allocation
2. Relocation
3. Link editing
4. Loading

Relocating an object entails allocating load time addresses and placing them in memory at appropriate locations.

(xii) CONCLUSION

To conclude this study, the source program must pass and parse all of the above-mentioned sections in order to be converted into the predicted target program. After reading this research paper, one will have a better understanding of the compilation task and the step-by-step evaluation of source code, which includes pre-processors, translators, compilers, compiler phases, compiler cousins, assemblers, interpreters, symbol tables, error handling, linkers, and loaders.

ACKNOWLEDGMENT

I would like to take this opportunity to thank everyone who helped me with this research. Throughout the research, I am grateful for their aspiring guidance, invaluable constructive criticism, and friendly advice. I am grateful to them for sharing their honest and illuminating perspectives on a variety of research-related issues.

REFERENCES

[1] Aho, Lam, Sethi, and Ullman – "Compilers: Principles, Techniques and Tools" - Second Edition, Pearson, 2014

[2] Mrs. Anuradha A. Puntambekar – "Compiler Design" - Technical Publication – Second Revised Edition August 2016

[3] Diagrams and Flowcharts – Available on : https://www.draw.io/s

Wikipedia - Available on:

[4] https://en.wikipedia.org/wiki/High-level_programming_language

[5] https://en.wikipedia.org/wiki/Low-level_programming_language

[6] https://en.wikipedia.org/wiki/Compiler