



International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

Life Cycle of Source Program - Compiler Design

Vishal Trivedi*

Gandhinagar Institute of Technology, Gandhinagar, Gujarat, India

E-mail: raja.vishaltrivedi@gmail.com

Abstract: This Research paper gives brief information on how the source program gets evaluated and from which sections source code has to pass and parse in order to generate target code or predicted output. In addition to that, this paper also explains the concept of Pre-processors, Translators, Linkers and Loaders and procedure to generate target code. This paper concentrates on Concept of Compiler and Phases of Compiler.

Keywords: Macro; Token; Lexemes; Identifier; Operators; Operands; Sentinel; Prefix; Postfix; Quadruple; Triple; Indirect triple; Subroutine

I. INTRODUCTION

Whenever we create a source code and start the process of evaluating it, computer only shows the output and errors (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. In addition to that touched topics are High level languages, Low level languages, Pre-processors, Translators, Compilers, Phases of Compiler, Cousins of Compiler, Assemblers, Interpreters, Symbol Table, Error Handling, Linkers and Loaders [1].

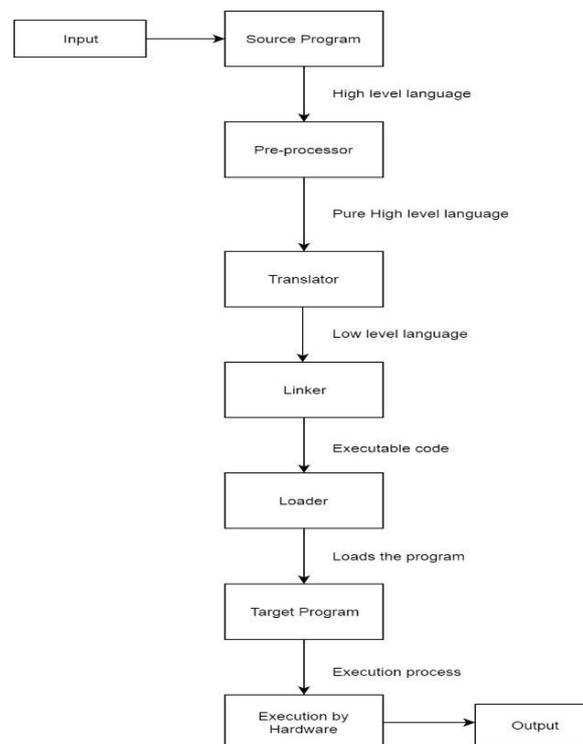


Figure 1: Life cycle of source code.



International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

II. HIGH LEVEL LANGUAGES

Source program is in the form of high level language which uses natural language elements and is easier to create program. It is programming language having very strong abstraction [1]. It makes the process of developing source code easier, simpler and more understandable. High level languages are very much closer to English language and uses English structure for program coding. Examples of high level languages are Visual Basic, PHP, Python, Delphi, FORTRAN, COBOL, C, Pascal, C++, LISP, BASIC etc.

III. LOW LEVEL LANGUAGES

Low level languages are languages which can be directly understand by machines. It is programming language having little or no abstraction. These languages are described as close to hardware. Examples of low level languages are machine languages, binary language, assembly level languages and object code etc.

IV. PRE-PROCESSORS

Pre-processor is a computer program that manipulates its input data in order to generate output which is ultimately used as input to some other program or compiler. Input of pre-processor is high level languages and output of pre-processor is pure high level languages. Pure high level language refers to the language which is having Macros in the program and File Inclusion. Macro means some set of instructions which can be used repeatedly in the program. Macro pre-processing task is done by pre-processor. Pre-processor allows user to include header files which may be required by program known as File Inclusion. Example: # defines PI 3.14 shows that whenever PI encountered in a program, it is replaced by 3.14 values (Figure 1).

V. 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 [2]. There are mainly three types of translators (Figure 2).

- 1) Compilers
- 2) Assemblers
- 3) Interpreters

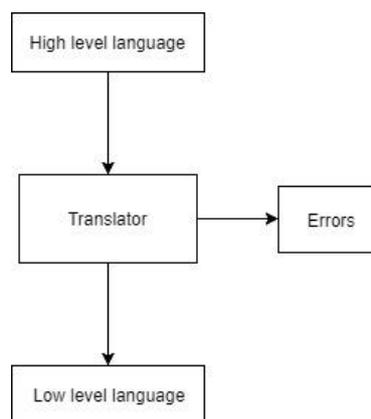


Figure 2: Translators.

VI. COMPILERS

Compiler reads whole program at a time and generate errors (if occurred). Compiler generates intermediate code in order to generate target code. Once the whole program is checked, errors are displayed. Example of compilers is Borland Compiler, Turbo C Compiler. Generated target code is easy to understand after the process of compilation [3]. The process of compilation must be done efficiently. There are mainly two parts of compilation process (Figure 3).

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

6.1 Analysis Phase

This phase of compilation process is machine independent. The main objective of analysis phase is to divide to source code into parts and rearrange these parts into meaningful structure. The meaning of source code is determined and then intermediate code is created from the source program. Analysis phase contains mainly three sub-phases named lexical analysis, syntax analysis and semantic analysis.

6.2 Synthesis Phase

This phase of compilation process is machine dependent. The intermediate code is taken and converted into an equivalent target code. Synthesis phase contains mainly three sub-phases named intermediate code, code optimization and code generation.

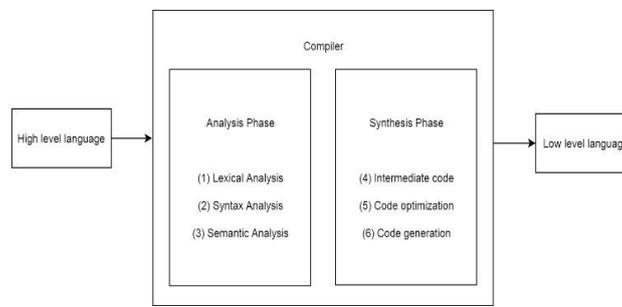


Figure 3: Compilers.

VII. PHASES OF COMPILER

As mentioned above, compiler contains lexical analysis, syntax analysis, semantic analysis, intermediate code, code optimization and code generation phases (Figure 4).

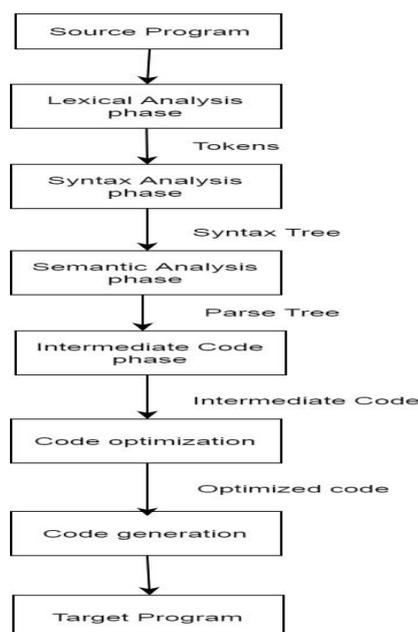


Figure 4: Phases of compiler.

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

7.1 Lexical Analysis

- Lexical Analysis is first phase of compiler.
- Lexical Analysis is also known as Linear Analysis or Scanning.
- First of all, lexical analyser scans the whole program and divides it into Token. Token refers to the string with meaning. Token describes the class or category of input string. Example: Identifiers, Keywords, Constants etc.
- Sentinel refers to the end of buffer or end of token.
- Pattern refers to set of rules that describe the token.
- Lexemes refers to the sequence of characters in source code that are matched with the pattern of tokens. Example: int, i, num etc.
- There are two pointers in lexical analysis named Lexeme pointer and Forward pointer [4].
- In order to perform token recognition, Regular Expressions are used to construct Finite automata which are separate topic itself (Table 1).
- Input is source code and output is token.
- Consider an Example:
Input: a=a + b*c*2;
Output:

=	a
+	b
*	c
	2

Table 1: Tokens or tables of tokens.

7.2 Syntax Analysis

- Syntax analysis is also known as syntactical analysis or parsing or hierarchical analysis.
- Syntax refers to the arrangement of words and phrases to create well-formed sentences in a language.
- Tokens generated by lexical analyzer are grouped together to form a hierarchical structure which is known as syntax tree which is less detailed (Figure 5).

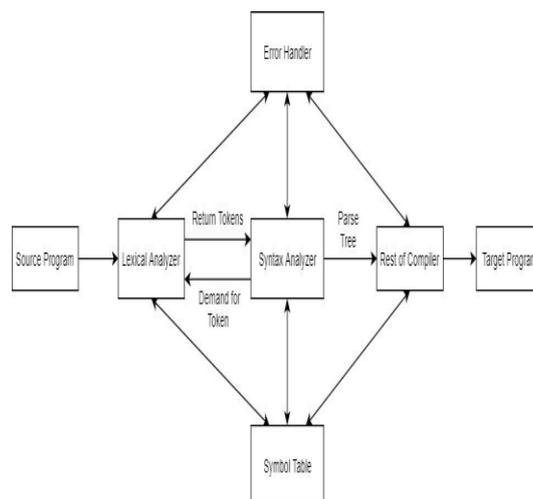


Figure 5: Lexical and syntax analyzer.

- Input is token and output is syntax tree (Figure 6).
- Grammatical errors are checked during this phase. Example: Parenthesis missing, semicolon missing, syntax errors etc.

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

- For above given example (Table 2).
Input:

=	A
+	B
*	C
	2

Table 2: Tokens or tables of tokens.

Output:

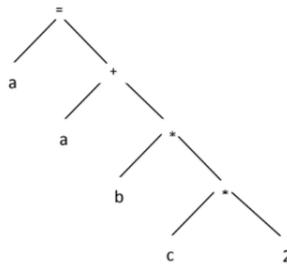


Figure 6: Syntax tree.

7.3 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) [5,6].
- Output for above given syntax tree is parse tree (Figure 7).

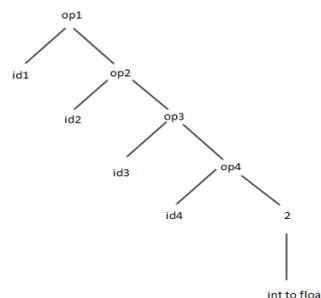


Figure 7: Parse tree.

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

7.4 Intermediate Code Input

- Intermediate code (IC) is code between high level language and low level language or we can say IC is code between source code and target code.
- Intermediate code can be easily converted to target code.
- Intermediate code acts as an effective mediator between front end and back end [7].
- Types of intermediate code are three address code, abstract syntax tree, prefix (polish), postfix (reverse polish) etc.
- Directed Acyclic Graph (DAG) is kind of abstract syntax tree which optimizes repeated expressions in syntax tree (Figure 8).

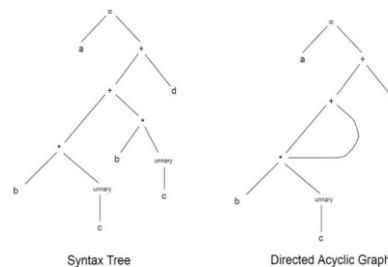


Figure 8: DAG

- Most commonly used intermediate code is three address code which is having no more than three operands.
- There are three representations used for three address code such as Quadruple, Triple and Indirect Triple.
- Input: Parse Tree

Output: Three address code

```
t1=int to float(2);
t2=id4*t1;
t3=id3*t2;
t4=id2+t3;
t4=id1;
```

7.5 Code Optimization

1. Code optimization is used to improve the intermediate code and execution speed.
2. It is necessary to have a faster executing code or less consumption of memory.
3. There are mainly two ways to optimize the code named Front-end (Analysis) and Back-end (Synthesis) [8,9].
4. In Front-end, programmer or developer can optimizes the code. In Back-end, compiler can optimizes the code.

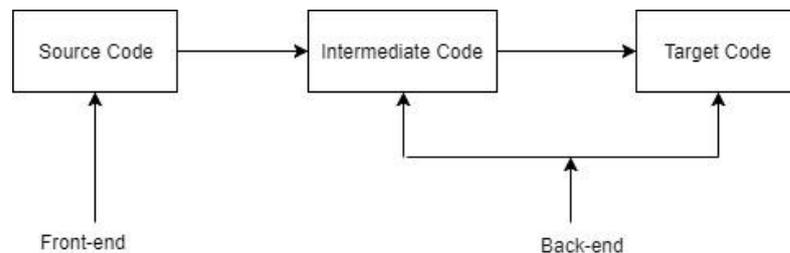


Figure 9: Code optimization ways.

5. Mentioned below are various techniques for code optimization (Figure 9).
 - Compile Time Evaluation
 - Constant Folding



International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

- Constant Propagation
- Common Sub Expression Elimination
- Variable Propagation
- Code Movement
- Loop Invariant Computation
- Strength Reduction
- Dead Code Elimination
- Code Motion
- Induction Variables and Reduction in Strength
- Loop Unrolling
- Loop Fusion etc.

Input: three address codes

Output: Optimized three address code

t1=id4*2.0;

t2=t1*id3;

id1=t2+id2;

6. Code Generation

- a) Code Generation is final phase of compiler.
- b) Intermediate code is translated into machine language which is pass and parse from above phases and lastly optimize.
- c) Properties desired by code generation phase are mentioned below.
 - Correctness
 - High Quality
 - Quick Code Generation
 - Efficient use of resources of target machine
- d) Common issues in code generator are mentioned below.
 - Memory management
 - Input of code generator
 - Target programs
 - Approaches to code generation
 - Choice of evaluation order
 - Register allocation
 - Instruction selection etc.

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

f) 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

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

VIII. COUSINS OF COMPILERS

Cousins refer to the context in which the compiler typically operates. There are mainly three cousins of compiler.

1. Pre-processors
2. Assemblers
3. Linkers and Loaders

IX. ASSEMBLERS

Assembler is a translator which takes assembly language as an input and generates machine language as an output. Output of compiler is input of assembler which is assembly language. Assembly code is mnemonic version of machine code. Binary codes for operations are replaced by names. Binary language or relocatable machine code is generated from the assembler [10]. Assembler uses two passes. Pass means one complete scan of the input program (Figure 10).

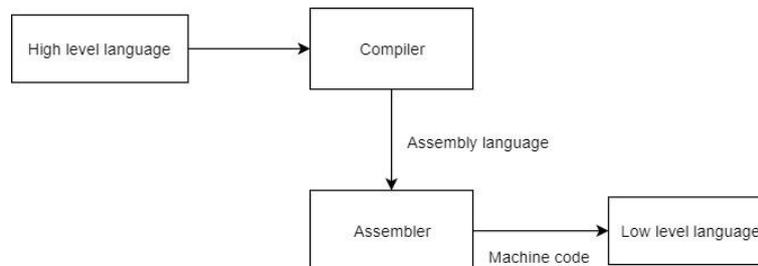


Figure 10: Assemblers.

X. INTERPRETERS

Interpreter performs the line by line execution of source code. It takes single instruction as an input, reads the statement, analyzes it and executes it. It shows errors immediately if occur [11]. Interpreter is machine independent and which does not produces object code or intermediate code as it directly generates the target code. Many languages can be implemented using both compilers and interpreters such as BASIC, Python, C#, Pascal, Java, and Lisp etc. Example of interpreter is UPS Debugger (Built in C interpreter). There are mainly two phases of Interpreter (Figure 11).

- 1) Analysis Phase: Analysis phase contains mainly three sub-phases named lexical analysis, syntax analysis and semantic analysis. This phase of compilation process is machine independent. Analysis phase of interpreter works same as analysis phase of compiler.
- 2) Synthesis Phase: This phase of compilation process is machine dependent. Synthesis phase contains sub-phase named code generation (direct execution) as it does not generate intermediate code.

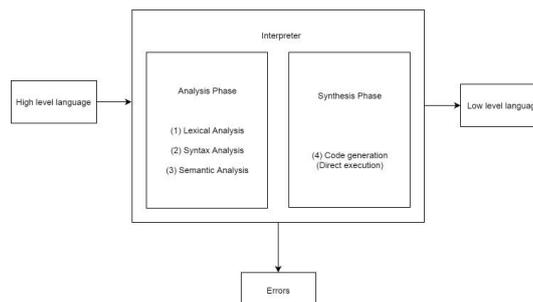


Figure 11: Interpreters.

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

XI. SYMBOL TABLE AND ERROR HANDLING

Translators such as compilers or assemblers use data structure known as Symbol table to store the information about attributes. It stores the names encountered in source program with its attributes. Symbol table is used to store the information about entities such as interfaces, objects, classes, variable names, function names, keyword, constants, subroutines, label name and identifier etc.

Each and every phase of compiler detects errors which must be reported to error handler whose task is to handle the errors so that compilation can proceed [9]. Lexical errors contain spelling errors, exceeding length of identifier or numeric constants, appearance of illegal characters etc. Syntax errors contain errors in structure, missing operators, missing parenthesis etc. Semantic errors contain incompatible types of operands, undeclared variables, not matching of actual arguments with formal arguments etc. There are various strategies to recover the errors which can be implementing by analyzers (Figure 12).

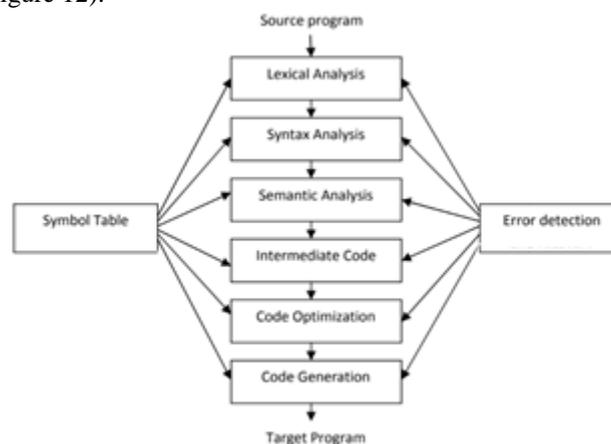


Figure 12: Symbol table.

XII. LINKERS AND LOADERS

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 [8].

Loader is utility program which takes object code as input and prepares it for execution. It also loads the object code into executable code. Loader refers to initializing of execution process. Tasks done by loaders are mentioned below.

- Allocation
- Relocation
- Link editing
- Loading

Relocation of an object means allocation of load time addresses and placement of load time addresses into memory at proper locations.

XIII. CONCLUSION

To conclude this research, source program has to pass and parse from above mentioned all sections to be converted into predicted target program. After studying this research paper, one can understand the exact procedure behind the compilation task and step by step evaluation of source code which contains pre-processors, translators, compilers, phases of compiler, cousins of compiler, assemblers, interpreters, symbol table, error handling, linkers and loaders.



ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Vol. 5, Issue 12, December 2017

XIV. ACKNOWLEDGMENT

I am using this opportunity to express my gratitude to everyone who supported me in this research. I am thankful for their aspiring guidance, invaluable constructive criticism and friendly advice during the research. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the research work.

XV. REFERENCES

1. High-level programming language and Compiler
2. <https://www.draw.io/s>
3. B Vangie, High-level language. Webopedia.
4. A Puntambekar Anuradha, Compiler Design. Technical Publication Second Revised Edition 2016.
5. K Dixita, Compiler Design. Darshan Institute of Engineering and Technology.
6. DA Hardik, System Programing. Darshan Institute of Engineering and Technology
7. Compiler Design - Symbol Table. Tutorials Point.
8. P Matt, W Christopher, Compilers. Department of Computer Science University of Wales Swansea, UK 2007.
9. P Neha, WM Niharika, et al. Introduction to Compilers. International Journal of Science and Research 2015; 4: 2399-2405.
10. A Charu, A Chetna, et al. Research Paper on Phases Of Compiler. International Journal of Innovative Research in Technology 2014; 1: 259-260.
11. VA Alfred, LS Monica, et al. Compilers: Principles, Techniques and Tools. Second Edition, Pearson, 2014.