COUSINS OF A COMPILER

G.SATYA MOHAN CHOWDARY [1] ,AKELLA S NARASIMHA RAJU[2]
[1]Assistant Professor,CSE,VSM College of Engineering,
[2]Associate Professor,Aditya PG College

Abstract*— This Research paper gives brief information on how the source program gets evaluated and how it is converted into various forms and finally into machine understandable language.We also discuss here the cousins of compiler which are Pre-processors,Assemblers,Linkers and Loaders and procedure to generate target code

Keywords— Macro; Token; Lexemes; Identifier; Operators; Operands; Intermediate code; Quadruple; Triple; Indirect triple;

I.INTRODUCTION

Whenever we write a program using any programming language and start the process of executing it, computer shows the output and errors (if occurred). We don't know what is the actual process involved in execution. 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 it also explains the types of languages,cousins of a compiler, Pre-processors, Translators, Compilers, Phases of Compiler,Interpreters, Symbol Table, Error Handling

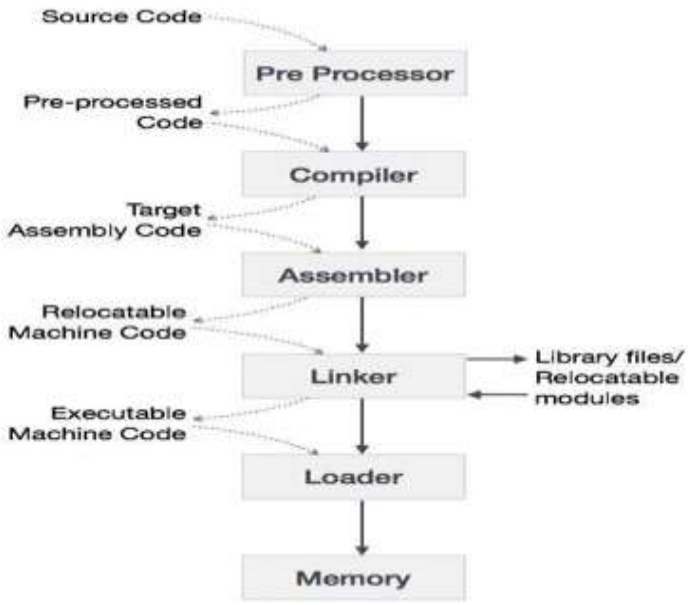

Figure 1: Life cycle of source code.

II.TYPES OF LANGUAGES

2.1 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 High level languages are very much closer to English language and uses English structure for program coding. Examples of high level languages are PHP, Python, Delphi, FORTRAN, COBOL, C, Pascal, C++, LISP, BASIC etc.

2.2 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.

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

III.PRE-PROCESSORS

Pre-processor converts the HLL (high level language) into pure high level language. It includes all the header files and also evaluates if any macro is included. It is the optional because if any language which does not support #include and macro preprocessor is not required.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).

IV.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. Assembler uses two passes. Pass means one complete scan of the input program (Figure 2).

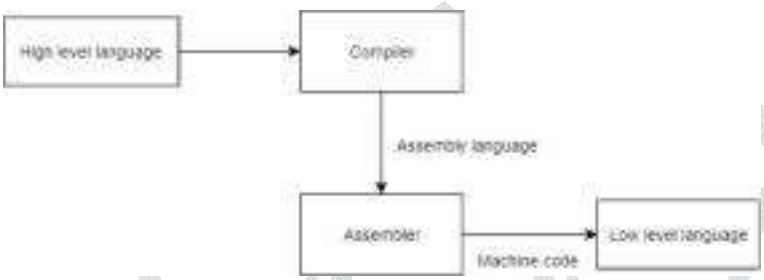

Figure 2: Assemblers.

V.LINKING AND LOADING

It has four functions

1. **Allocation:**
   It means get the memory portions from operating system and storing the object data.
2. **Relocation:**
   It maps the relative address to the physical address and relocating the object code.
3. **Linker:**
   It combines all the executable object module to pre single executable file.
4. **Loader:**
   It loads the executable file into permanent storage.

VI.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 (Figure 3).

A. Compilers
B. Assemblers
C. Interpreters

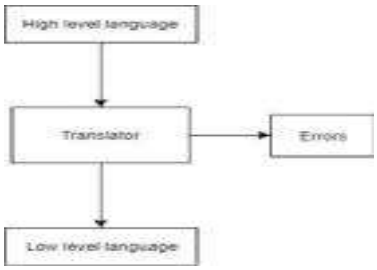

Figure 3: Translators.

VII.COMPILERS

Compiler reads whole program at a time and generate errors (if occurred). It is a translator which converts the source code from high level of language to low level language. The process of compilation must be done efficiently. There are mainly two parts of compilation process (Figure 4).

7.1 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.

7.2 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.

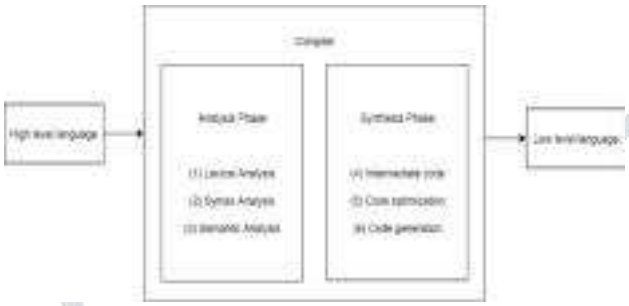

Figure 4: Compilers.

VIII.PHASES OF COMPILER

As mentioned above, compiler contains lexical analysis, syntax analysis, semantic analysis, intermediate code, code optimization and code generation phases (Figure 5).

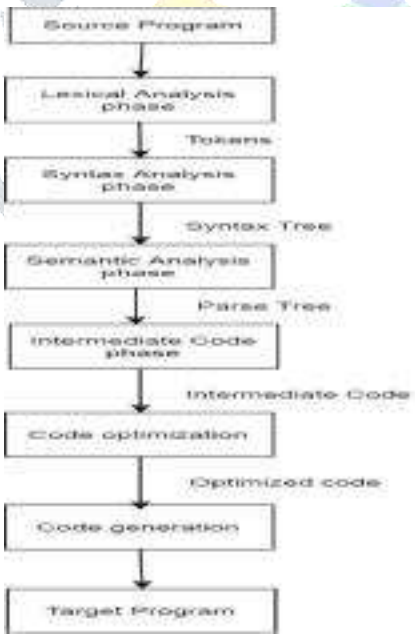

Figure 5: Phases of compiler.

8.1 Lexical Analysis

1) Lexical Analysis is the first phase of a compiler.
2) Lexical Analysis is also known as Scanner.
3) First of all, lexical analyser scans the whole program and divides it into Tokens. Token refers to the string with meaning and it describes the class or category of input string. Example: Identifiers, Keywords, Constants etc.
4) Pattern refers to set of rules that describe the token.

5) Lexemes refers to the sequence of characters in source code that are matched with the pattern of tokens. Example: int, i, num etc.
6) The scanning process is done by using Input Buffering.There are two pointers in Input buffering named Lexeme pointer and Forward pointer
7) Regular expressions are used to recognize tokens.
8) Here the input is source code and output is token.

8.2 Syntax Analysis

- Syntax analysis is also known as parser 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 tree structure which is known as syntax tree (Figure 6).

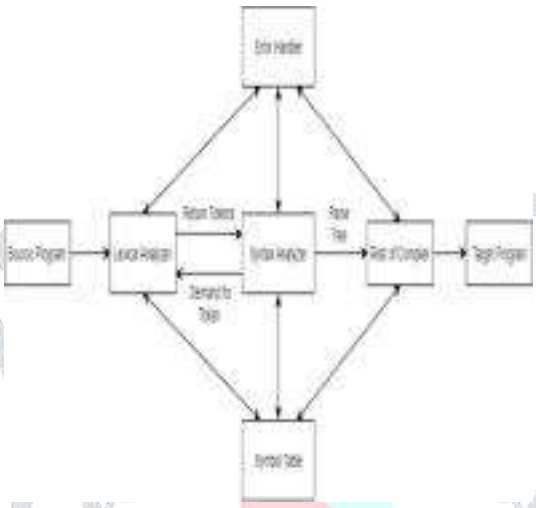

Figure 6: Role of Lexical analyzer and syntax analyzer.

- Input is token and output is syntax tree (Figure 7).
- Grammatical errors are checked during this phase. Example: Parenthesis missing, semicolon missing, syntax errors etc.

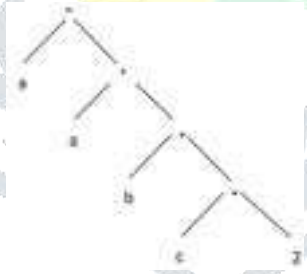

Figure 7: Syntax tree.

8.3 Semantic Analysis

- Semantic analyzer checks the meaning of source program as the word **semantic** itself represents meaning.
- 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)
- Output for above given syntax tree is parse tree (Figure 8).

.

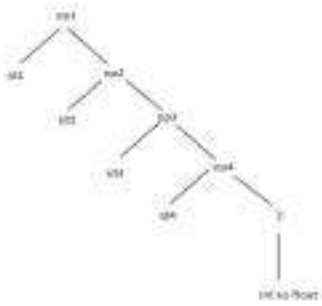

Figure 8: Parse tree.

8.4 Intermediate Code Generator

- 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
- 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 9).

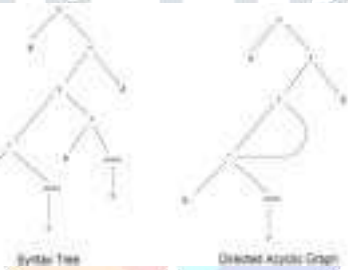

Figure 9: DAG

- Most commonly used intermediate code is three address code which is having atmost 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;
```

8.5 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)
   In Front-end, programmer or developer can optimizes the code. In Back-end, compiler can optimizes the code.

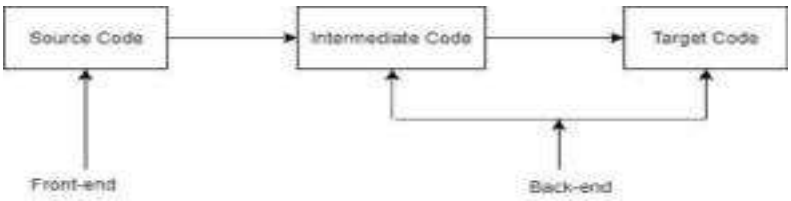

Figure 10: Code optimization ways.

4. Mentioned below are various techniques for code optimization (Figure 10).
   - Compile Time Evaluation
   - Constant Folding

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

8.6 Code Generation

a) Code Generation is the final phase of a compiler.
b) Intermediate code is translated into machine language which is pass and parse from above phases and lastly optimize.
c) Target code which is now low level language goes into linker and loader.
d) Input: Optimized three address code

IX.SYMBOL TABLE AND ERROR HANDLING

9.1 Symbol Table

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.

9.2 Error Handling

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.
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 11)

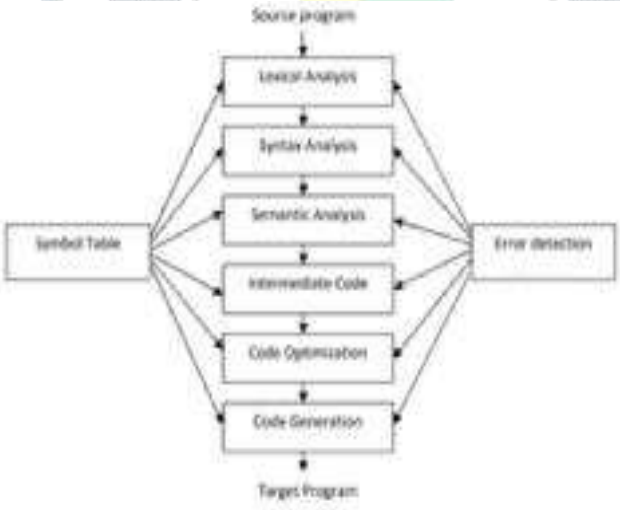

Figure 11: Symbol table.

X.EXAMPLE FOR THE PHASES OF A COMPILER

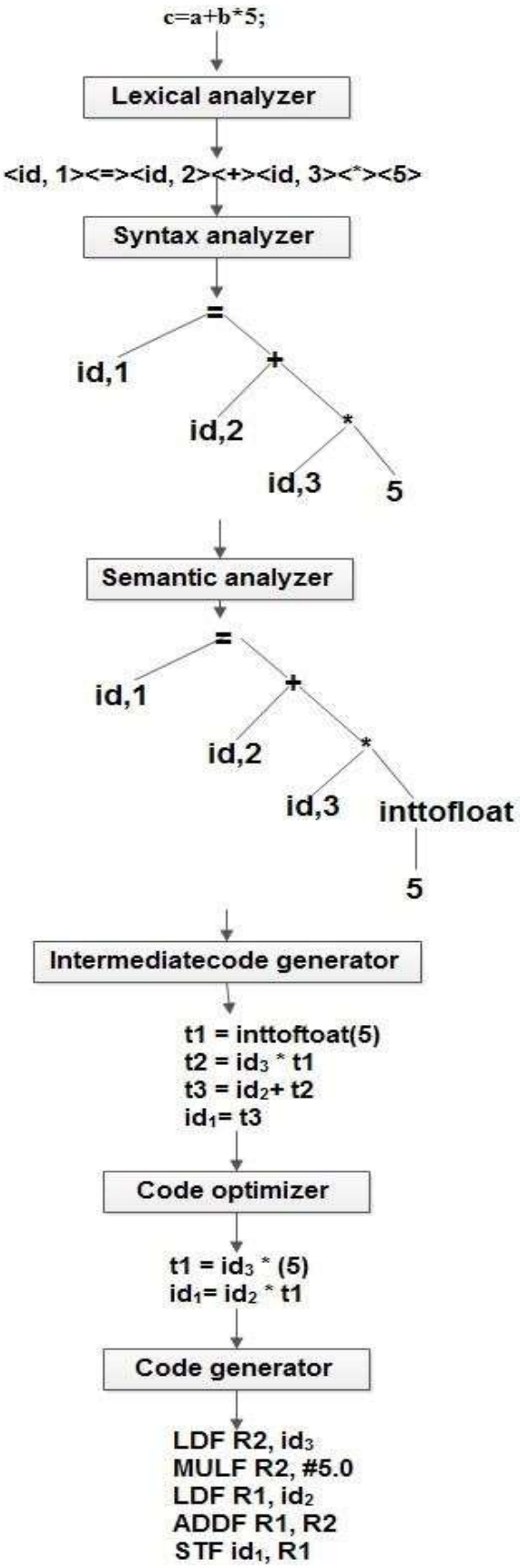

FIGURE 11: CODE TRANSLATION VIA A COMPILER

XI. CONCLUSIONS

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.

XII. REFERENCES

[1] A Puntambekar Anuradha, *Compiler Design. Technical Publication* Second Revised Edition 2016.
[2] Compiler Design - Tutorials Point. https://www.tutorialspoint.com//compiler_design/index.htm
[3] P Matt, W Christopher, Compilers. Department of Computer Science University of Wales Swansea, UK 2007.
[4] Vishal Trivedi et al. Life cycle of source program-Compiler Design,
International Journal of Innovative Research in Computer and Communication Engineering