IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

State Level Conference – AITCON 2K24

Adarsh Institute of Technology & Research Centre, Vita, Maharashtra

Vol. 11, Special Issue 1, March 2024

Case Study on Working of Compiler: Using Lexical Analyzer, Syntax Analyzer, Semantic Analyzer

Mr. Pravin S. Landage[1], Mr. Mahesh N. Landage[2], Mr.Ganesh P. Buchde[3], Mr. Harsh D. Waghmode [4], Mr. Hrushikesh R. Lawate[5]

I/C HOD, Department of Computer Technology, Shivaji polytechnic Atpadi, India [1]

Lecturer, Shivaji Polytechnic Department of Computer Technology, Atpadi, Maharashtra, India[2-3]

Students, Shivaji Polytechnic Department of Computer Technology, Atpadi, Maharashtra, India[4-5]

Abstract: Compiler infrastructures that support experimental research are crucial to the advancement of high- performance computing. Compiler technology must be implemented and evaluated in the context of a complete compiler, but developing such an infrastructure requires a huge investment in time and resources. in this research document i studied about what actually compiler do, and how languages are interpreted the human readable language into the machine readable language using the compiler and *Lexical Analyzer*, actually the compilers are the interpreters or translators to translate the human language or *high-level language* into machine language for generating output for users in this research document we are knowing about how does computer analyzing our language and generating output for us.

Keywords: interpreter, high-level language, low level language Lexical Analyzer

I. INTRODUCTION

The **Compiler** is software that converts a program written in a high-level language (Source Language) to a low- level language (Object/Target/Machine Language /0, 1's). A translator or language processor is a program that translates an input program written in a programming language into an equivalent program in another language. The compiler is a type of translator, which takes a program written in a high-level programming language as input and translates it into an equivalent program in low- level languages such as machine language or *assembly language*. Compiler is an intelligent program as compare to an assembler. Compiler verifies all types of limits, ranges, errors, etc. Compiler program takes more time to run and it occupies huge amount of memory space. The speed of compiler is slower than other system software. It takes time because it enters through the program and then does translation of the full program. When compiler runs on same machine and produces machine code for the same machine on which it is running. Then it is called as self-compiler or resident compiler. Compiler may run on one machine and produces the machine codes for other computer then in that case it is called as cross compiler.

II. TYPES OF LANGUAGES

a. Low Level Language
b. Middle Level Language
c. High Level Language

a. Low Level Language:
Low level language is also known as Machine Language. Machine language is also known as Machine code. Binary language is an example of *low level language*. The binary language contains only two symbols 1 and 0. All the instructions of binary language are written in the form of binary numbers 1's and 0's. A computer can directly understand the binary language. *Low Level language* is as the First generation language.

b. Middle Level Language
Middle level language is also known as *Assembly language* or Symbolic language. *Assembly language* is an example of Middle level language. In *Assembly language*, the instructions are created using symbols such as letters, digits and special characters. In *assembly language*, we use predefined words called mnemonics. A program written is an *assembly language* using mnemonics called *assembly language* program or symbolic program. The process of translating an *assembly language* program into its equivalent machine language program with the use of an assembler. Assembler is used to translate middle level language to *low level language*.

c. High Level Language

High level can be easily understood by the users. It is very similar to the human language and has a set of grammar rules that are used to make instructions more easily. Every *high level language* has a set of predefined words known as keywords and a set of rules known as syntax. *High level language* is a programming language. Languages like COBOL, BASIC, FORTRAN, C, C++, JAVA etc. All these programming languages are to write program instructions. These instructions are converted to *low level language* by the complier or *interpreter*.

III. COMPILERS/INTERPRETERS/ASSEMBLER

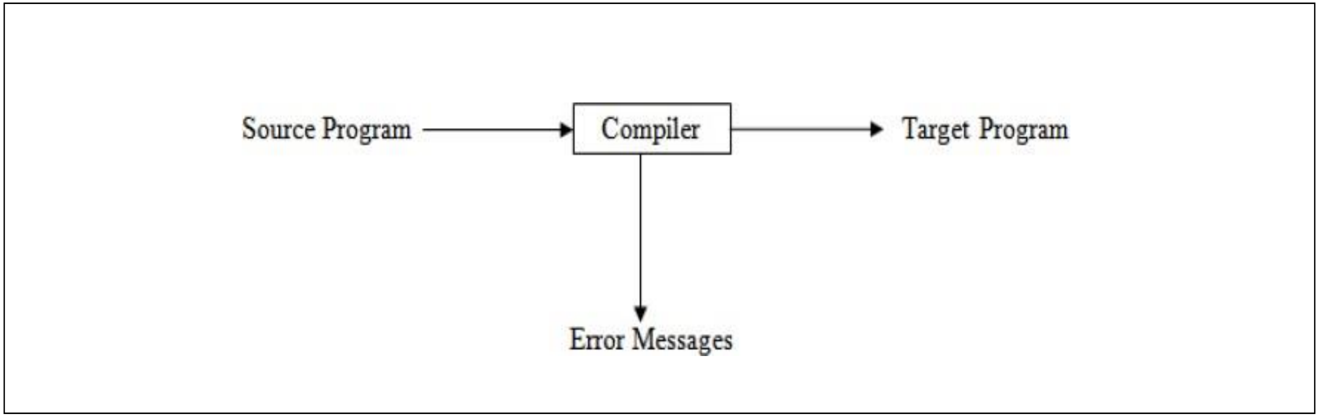

Fig.1 Structure of compiler

Compiler is a translator which is used to convert programs in high level langue to *low level language*. It translates the entire program that is group of statements at a time and also reports the errors in source program encountered during the translation.

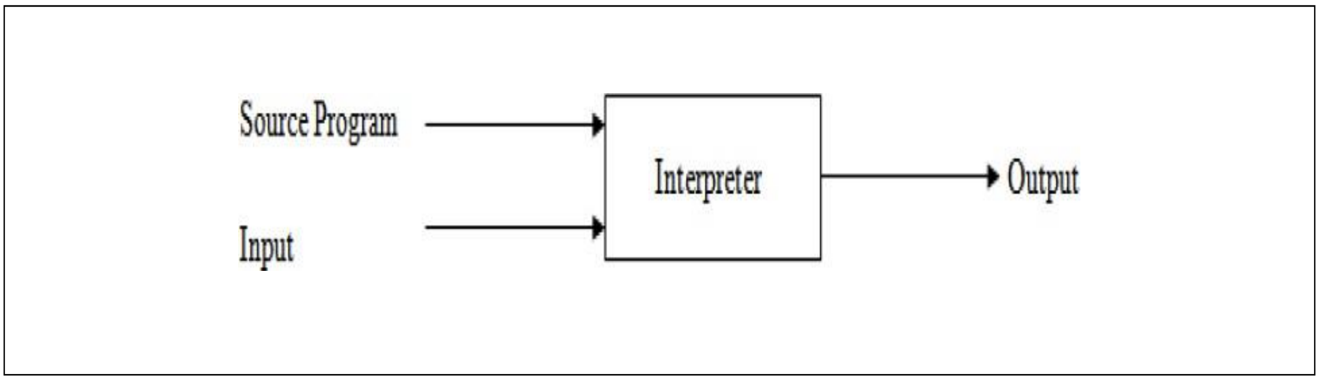

Fig.2 Structure of Interpreter

Interpreter is a translator which is used to convert programs in *high level language* to *low level language*. *Interpreter* translates line by line statements and reports the error once it encountered during the translation process. It gives better error diagnostics than a compiler.

Assembler is a translator which is used to translate the *assembly language* code into machine language code.

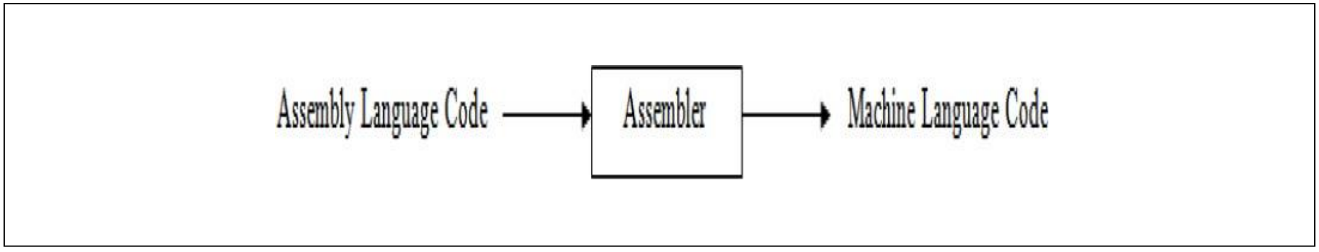

Fig.3 Structure of Assembler

IV. WORKING OF COMPILERS [TASKS]

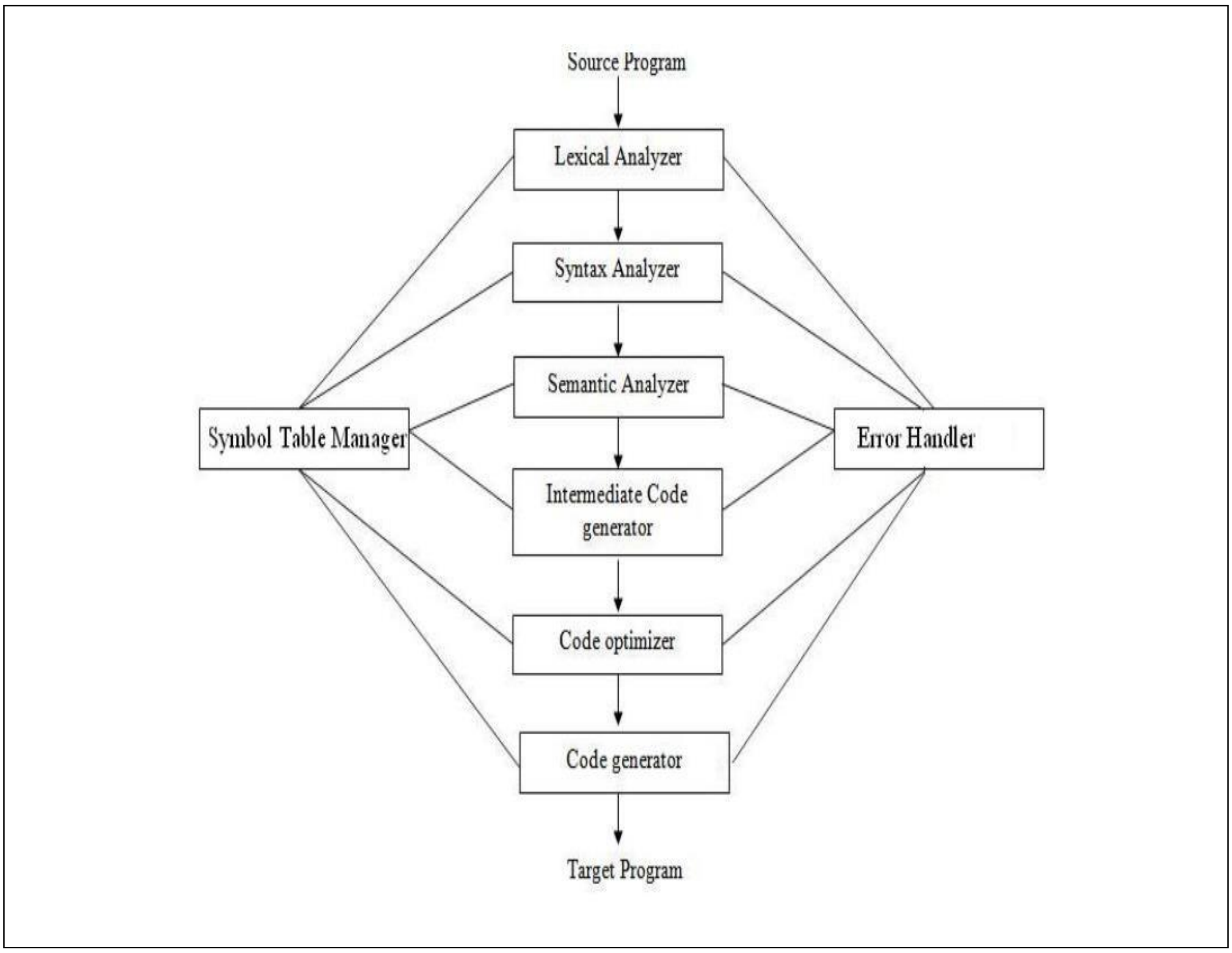

Fig.4 Tasks Performed By Compilers

The compilation process is a sequence of various phases such as *Lexical analyzer*, Syntax analyzer, Semantic analyzer, Intermediate code generator, Machine independent code optimizer, Code generator, Machine dependent code optimizer. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

The different phases of compilation process are as:
1) Lexical analysis 2) Syntax analysis 3) Semantic analysis 4) Intermediate code generation 5) Code optimization 6) Code generation

1. Lexical analysis:
This phase is the first phase of compiler process which scans the source program as a stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output. Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, and identifiers. Lexeme is instance of token that is group of characters forming a token. Form of tokens as Example c = a + b * 5; Here, c, =, a, +, b, *, 5 are lexemes a, b, c are identifiers, = is assignment symbol, + is addition symbol, * is multiplication symbol and 5 is number. These are called tokens. Hence, < = > < +> < 5 >

2. Syntax analysis:
This phase is the second phase of compiler process which is also called as parsing. Parser converts the tokens produced by lexical analyser into a tree like representation called parse tree. A parse tree describes the syntactic structure of the input.

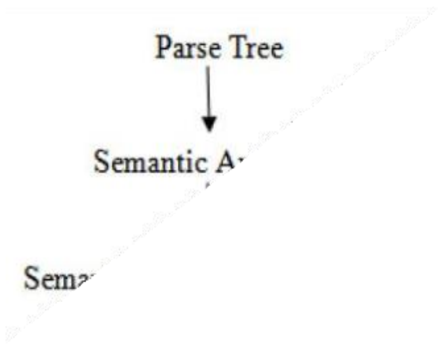

Fig. 5 Parse Tree

Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator. Consider an expression c = a + b * 5;

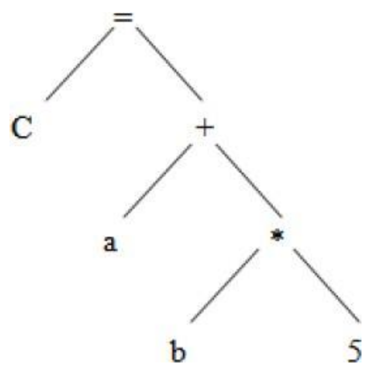

Fig. 6 Parse Tree

3. Semantic analysis:

Semantic analysis is the third phase of compiler. This phase checks for the semantic consistency.
Type information is gathered and stored in symbol table or in syntax tree.

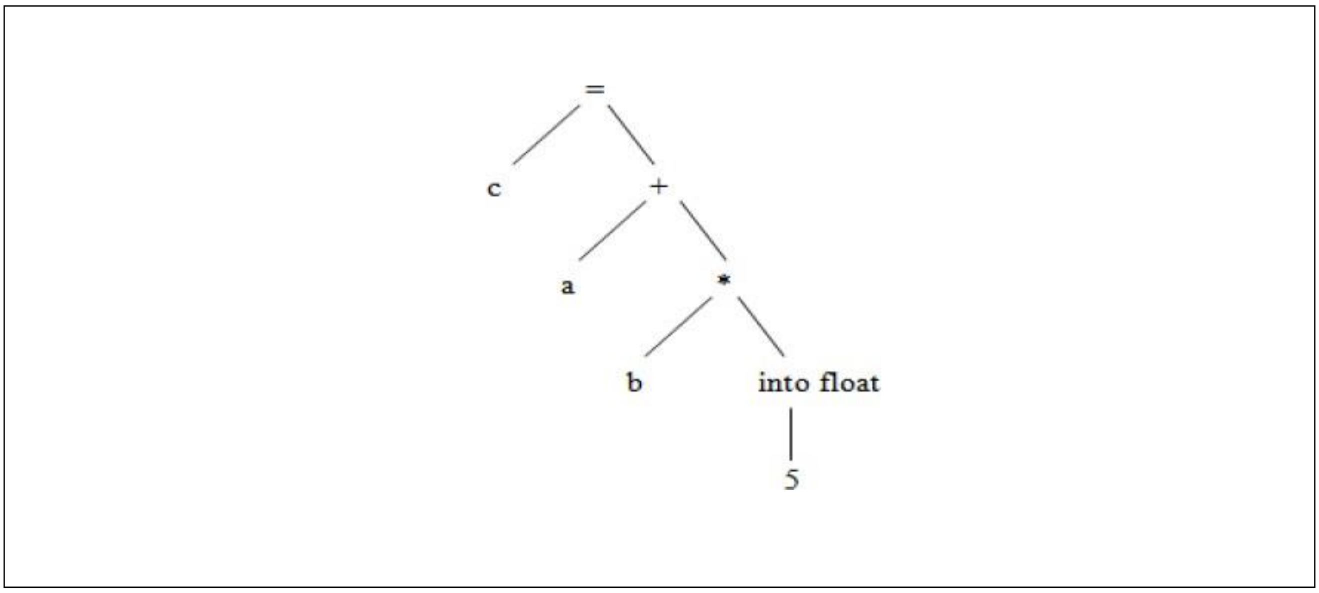

Fig. 7 Syntax Tree

4. Intermediate code generator:

Immediate code generation produces intermediate representations for the source program which are of the following forms-

- Postfix notation
- Three address code
- Syntax tree

Most commonly used form is the three address code. t1 = int to float (5) t2 = id3 * tl t3 = id2 + t2 id1 = t3 Advantages of Intermediate code a) It should easy to produce. b) It should be easy to translate into target program

5. Code optimization:

Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output. It results in faster running machine code. It can be done by reducing the number of lines of code for a program. This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result. During the code optimization, the result of program is not affected. To improve the code generation, the optimization involves-

a. Deduction and removal of dead code (unreachable code).
b. Calculation of constants in expressions and terms.
c. Collapsing of repeated expression into temporary string.
d. Loop unrolling.
e. Moving code outside the loop.
f. Removal of unwanted temporary variables. t1 = id3 * 5.0 id1 = id2 + t1

6. Code generation:

This phase is the final phase of compilation process. This phase gets input form code optimization phase and produces the target code or object code as result. Intermediate instructions are translated into a sequence of machine instructions that performs the same task. The code generation involves-

a. Allocation of register and memory.
b. Generation of correct references.
c. Generation of correct data types.

Generation of missing code. LDF R2, id3 MULF R2, #5.0 LDF R1, id2 ADDF R1, R2 STF id1, R1

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

State Level Conference – AITCON 2K24

Adarsh Institute of Technology & Research Centre, Vita, Maharashtra

Vol. 11, Special Issue 1, March 2024

V. CONCLUSION

On above research documentation i would like to prefer the conclusion is that using lexical analyazer generates the machine code that is the byte data provides to the computers and compilers /interpreters are the mediators between the human readable data to the machine readable data

REFERENCES

[1]. https://www.geeksforgeeks.org/
[2]. An infrastructure for research on parallelizing and optimizing compilers.
[3]. [3]Evolution of Programming Languages Dr. Jivtode.
[4]. Compilers: Principles, Techniques, & Tools 2nd Edition" by Alfred V Aho and Ravi Sethi
[5]. Compiler Design" by K Muneeswaran
[6]. Online website : https://en.wikibooks.org/wiki/Compiler_Construction/Case_Study_1B
[7]. Hunter, R.B. (1981) *The Design and Construction of Compilers*, Wiley, New York.
[8]. Holmes, J. (1995) *Object-Oriented Compiler Construction*, Prentice-Hall, Englewood Cliffs, NJ.
[9]. Wirth, N. (1996) *Compiler Construction*, Addison-Wesley, Wokingham, England.
[10]. Grosch, J. (1989) Efficient generation of lexical analysers, *Software - Practice and Experience*, **19**(11), 1089- 1103.
[11]. Aho, A.V., Sethi, R. and Ullman, J.D. (1986) *Compilers: Principles, Techniques and Tools*, Addison- Wesley, Reading, MA.