IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

Review of Compiler Structure and Processing in Compiler Design

Barkha Gupta[1]

Programme Assistant, Department of Computer, Agriculture University, Jodhpur, Rajasthan, India[1]

Abstract: Computer became integral tool in our lives because it helps in many ways by solving the human problems as and when required but when it comes to the working of computer. It becomes very difficult to understand. In normal and day-to-day scenario, software engineers and computer programmer write the programming or code in high level language, which is understood by human but not by machine and to make that code understable to machine there come the requirement of converting that language to computer understable form and this generates the need of compiler. Compiler is a computer program which translate high level language into the machine understable form. Compiler do this conversion in numbers of phases which will be covered here.

Keywords: Compiler design, compiler phase, syntax analysis, semantic analysis, code generation, code optimization, structure of compiler.

I. INTRODUCTION

A compiler is a set of computer program that is used to convert source code written in any programming language generally termed as source code into another programming language generally termed as target code or an object code. Compiler is generally used to convert any high-level language into lower-level language. The reason behind this process is to create an executable file of a program.

II. BASIC TERMINOLOGIES

(i) **De -compiler**- Reverse process of a complier is known as de-compiler. A set of computer program that translate higher-level programming language into lower-level programming language.

(ii) **Cross-compiler**- A Cross compiler is a compiler that is capable enough to create an executable file for a platform or operating system different from its current one on which it is running. It runs on platform A and is capable of generating executable code for platform B. For example, a compiler that is running on windows operating system but create an executable file for Android operating system.

(iii) **Language Rewriter**- Language rewriter is a computer program that translate the form of expression without changing its programming language.

(iv) **Source to Source Compiler** – Source to source compiler is capable to take the source code of one programming language and translate it into the source code of another programming language.

III. COMPILER PHASE

Compiler has two phases: -

(i) **Analysis Phase** – It is also known as front end analysis. This phase is machine independent phase. It includes following: -

a) Divides the source program into tokens
b) Groups tokens into syntactic structure
c) Verify lexical grammar and syntax error
d) Generates intermediate code and symbol table
e) Perform machine independent optimization

(ii) **Synthesis Phase** – It is also known as back-end analysis. This phase is machine dependent phase. It includes following: -

a) Generates target code
b) Target code optimization

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

IV. STRUCTURE OF COMPILER

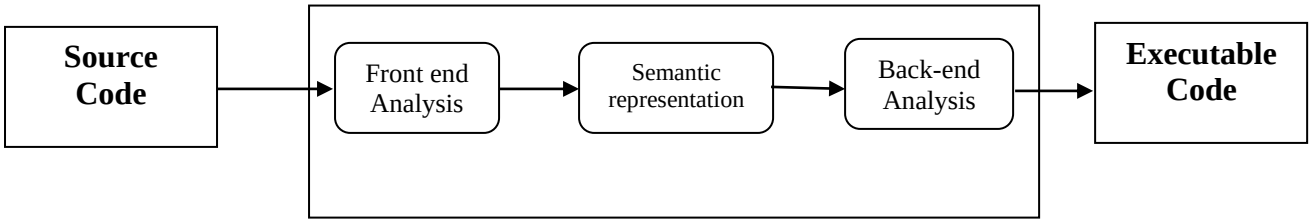


Fig 1: Basic Structure of Compiler

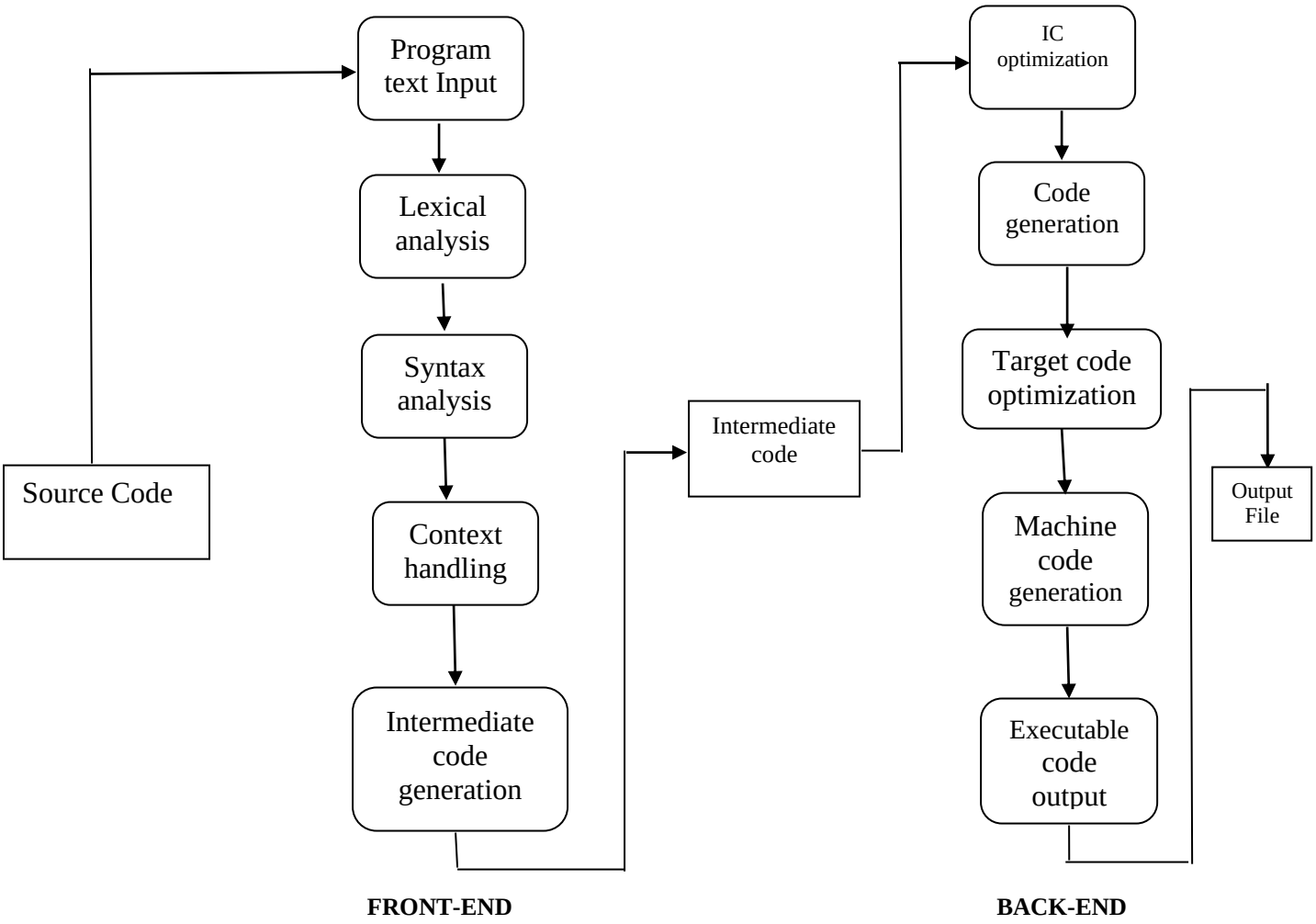


Fig 2: Detailed structure of compiler

Process of compilation is divided into five phases which are as follows: -

(i) **Phase 1 : Lexical Analysis**

Lexical analyzer divides the source program into tokens. It reads one character at a time and group them into the keywords, identifiers, operators which are known as tokens. It removes white spaces and comments from the programming language and maintains the line number. It creates storage for identifier in symbol table. Lexical Analyzer is also known as linear analyzer or scanner. Tokens has a particular pattern which is defined by regular expression. It follows longest match rule and rule priority where reserved word is given the highest priority than user defined word. It generates transition table.

Example: Printf ("you are welcome");

In the above sentence there are 5 tokens which are as follows

Printf	(	" " (counted as one)	you are welcome	)

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

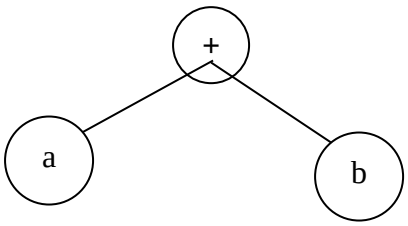
(ii) **Phase 2 : Syntax Analysis**

In this phase of compiler, it groups the tokens into syntactic structures. It builds the symbol table. The parser does the syntax analysis on the groups of tokens. This phase is also known as hierarchical analysis or hierarchical parsing. This phase uses context free grammar to check balancing token which is recognised by push down automata. Output of this phase is a parse tree.

Context free grammar has 4 components: -

a) Non terminal symbol
b) Terminal symbol
c) Set of productions
d) Start symbol (one of non-terminal symbol)

Example a + b then its syntactic structure will be



Derivation here are of two types: -

a) **Leftmost derivations** – Derivation is called leftmost derivation if and only if all steps involved in derivation have leftmost variable replacement only.

Example 1. Consider a context free grammar

S xS | AA, A → YA | xA | xBB | x, B y →

Find leftmost derivation for w = xxxyyxx.

Then solution is

	S xS	→
	S xxS	→
S	xxxS	→
S	xxxAA	→
S	xxxyAA	→
S	xxxyyAA	→
S	xxxyyxA	→
S	xxxyyxx	→

b) Rightmost derivations - Derivation is called rightmost derivation if and only if all steps involved in derivation have rightmost variable replacement only.

Example 1. Consider a context free grammar

S xS | AA, A → YA | xA | xBB | x, B y →

Find rightmost derivation for w = xxxyyxx.

Then solution is

	S xS	→
	S xxS	→
S	xxxAA	→
S	xxAxA	→
S	xxAxx	→
S	xxxBBxx	→
S	xxxByxx	→
S	xxxyyxx	→

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

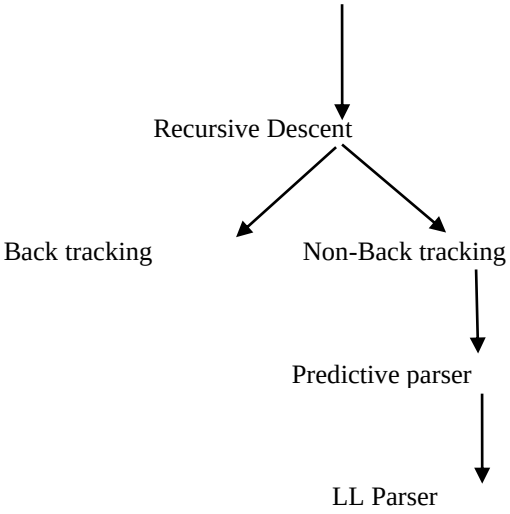
Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

Parser is divided into two parts: -

(i) Top-Down Parsing

Top-Down Parsing



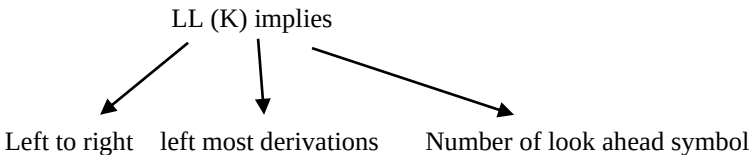
a) **Recursive descent parsing** – It is the top-down parser. It recursively parses input to make parse tree which may or may not require backtracking. This parser has more than one production to choose from single instance of input.

b) **Back tracking** – it is the top-down parser which chooses backtracking till not found the matching string or the desired output.

c) **Predictive parser** - It is the top-down parser which do not suffer from backtracking. Predictive parser uses look ahead pointer which points to the next input symbol. To make this backtracking free, predictive parser put some constraint on grammar and accepts only LL (K) grammar. Predictive parser uses stack and parsing table to parse input and generated parse tree. In predictive parser each step has utmost one production to choose.

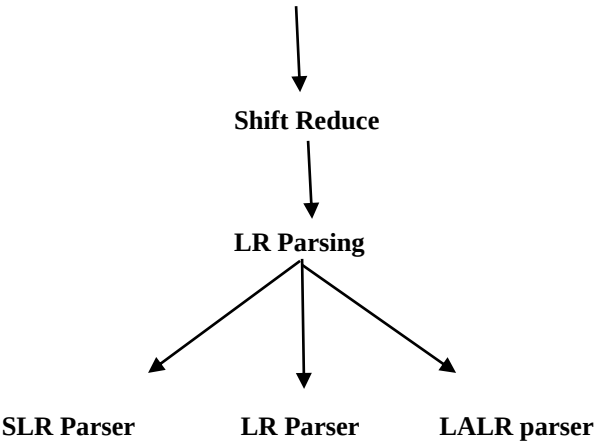
d) **LL Parser** – It is top-down parser. It accepts LL grammar. LL grammar is subset of context free grammar. It is implemented by recursive descent or table driven.

Here



(ii) Bottom-Up Parsing

Bottom-Up Parsing



IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

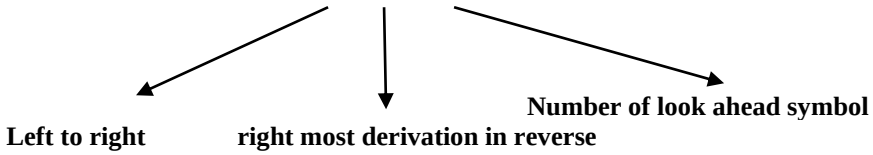
a) **Shift Reduce –** shift reduce has two steps: -

1) **Shift step –** Shift step implies shifting of input pointer to next input symbol and push that onto stack.

2) **Reduce step-** Reduce step implies replacement of left-hand side variable which is done by popping the element from the stack.

b) **LR Parser –** It is the most efficient parser.

Here **LR (K) implies**



LR Parser is divided into three widely used parser: -

1) **SLR Parser –** It is the simplest LR Parser. It works on smallest class of grammar. It has only few numbers of states hence it has small table. It is simple in process and hence fast construction.

2) **LR Parser –** It works on set of LR (1) grammar. It generates large table and large number of states hence it has slow construction.

3) **LALR Parser –** It is look ahead LR Parser. It works on intermediate size of grammar.

Difference between LL Parser and LR Parser

S.No.	LL Parser	LR Parser
1.	Left most derivation	Rightmost derivation in reverse
2.	It starts from the root	It ends with the root
3.	It ends when stack is empty	It starts with empty stack
4.	It builds parse tree while following top-down approach	It builds parse tree while following bottom-up approach
5.	It uses stack for designating what is still to be expected	It uses stack for designating what is already seen or visited
6.	It expands the non-terminal symbol	It reduces the non-terminal symbol
7.	It follows pre order traversal of parse tree	It follows post order traversal of parse tree
8.	It read terminal when it pops one off the stack	It read the terminal while it pushes them to stack
9.	It continually pops a non-terminal symbol off the stack and pushes the corresponding right hand side symbols.	It tries to recognise right hand side on the stack, pop it and pushes the corresponding non terminal symbol.

Phase 2 also includes semantic analysis. (or it can be included into different and separate phase). This phase is also termed as type checker phase. This phase helps to interpret symbols, their types and relation with each other. It judges whether syntax structure constructed in source program derives any meaning or not. It uses syntax directed translation. It produces annotated syntax tree as an output.

Context Free Grammar + Semantic Rules = Syntax Directed Definitions

Here attribute is divided into two parts: -

(i) **Synthesis attribute –** Synthesis attribute is also termed as s-attribute. It evaluates post order traversal. It gets the values from their child nodes. It defined the left-hand side variables.

Example: - S → ABC

Here S is a synthesis attribute.

(ii) **Inherited attribute–**It evaluates pre order traversal. It gets the values from parent or siblings. It defined the right-hand side variables.

Example: - S → ABC

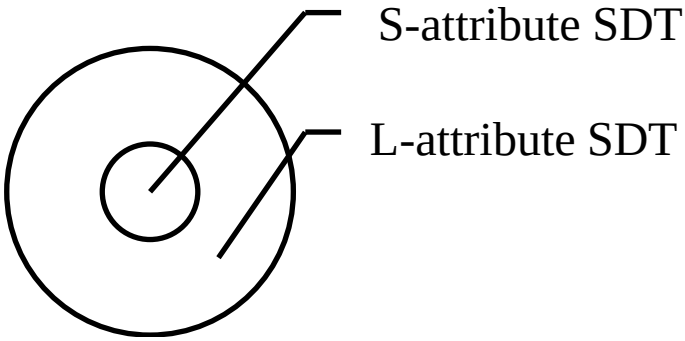
Here A, B, C are inherited attributes.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264



Semantic analyser receives abstract syntax tree (AST) from the previous phase. It attaches attribute information with AST called attribute AST.

S-attribute SDT – This attribute evaluates bottom-up parsing. In this semantics action are placed at right hand side. It uses only synthesis attribute.

L-attribute SDT – This attribute evaluates depth first parsing and moves from left to right side. It uses both synthesis and inherited attribute restricted from parent or left siblings only.
Example - S → ABC
Here S can take the value from A, B, C (synthesis)
A can take the value from S
B can take the value from S, A
C can take the value from S, A, B

(iii) **Phase 3 : Intermediate Code Generation**

In this phase simple instruction has been generated or intermediate code is generated form the syntactic structure generated in its previous phase. Intermediate code can be language specific or language dependent. Intermediate representation of a code is divided into two parts: -

a) **High level intermediate representation** – This representation is close to the source code. It is simple and easy code. Modification in the code can take place here but target code optimization is less preferred here.

b) **Low level intermediate representation - This** representation is close to the target machine code. It is suitable to the registers and memory allocation. Machine code optimization is preferred here.

(iv) **Phase 4 : Code optimization**

This phase is an optional phase. It is from the perspective of saving and utilizing the memory. It is used to reduce the usage of number of registers during the programming language. This is done by code optimizer.

a) **Peephole optimization** – It works locally on the source code to transform into an optimized code. A small portion is analysed and checked for possible optimization.

1) **Redundant instruction elimination**

Example –

```
 int fun (int x)
{
Int y, x;
y = 10;
z = x + y;
return z;
}
```

Must be written as

```
Return x + 10;
}
```

2) **Remove unreachable code**
3) **Flow of control optimization** – where program jump back and forth without performing any task.
4) **Algebraic expression simplification** – like a = a + 1 must be written as a++
5) **Strength reduction** – like x^2 must be written as x<<1

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

b) **Loop optimization** – It includes following: -

1) **Invariant code** – The code which exist inside loop and computes same value at each iteration. So, this code must be moved outside the loop body.
2) **Induction analysis** – if its value is altered within the loop-by-loop invariant value.
3) **Strength reduction**
4) **Dead code elimination**
5) **Partial redundancy**

(v) **Phase 5 : Code Generation**

In this phase target code or output code has been generated which can be executed on any device. It produces target code by deciding its memory location for data, selecting appropriate registers to perform the various computations and selecting code for accessing and generating output. Code generator should consider: -

1) Target language
2) Intermediate representation
3) Register allocation
4) Ordering of instruction
5) Selection of instruction

Code generator has two descriptors:

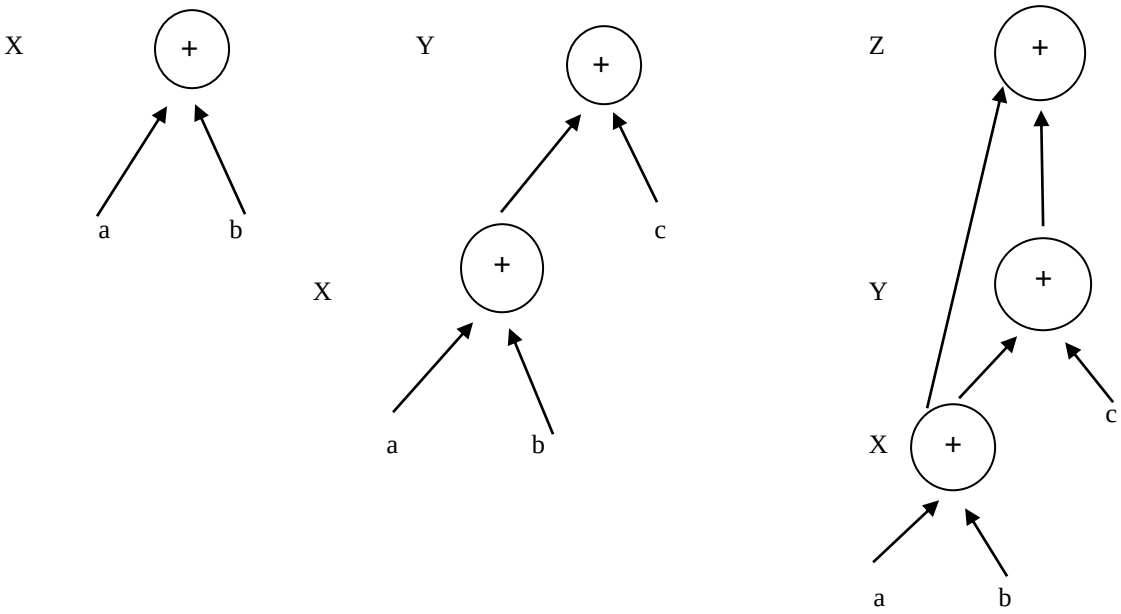
1) **Register descriptors** – which specify the availability of registers
2) **Address descriptors** – which is used to retrieve the value of variable stored at different location, to keep track on it.

Directed Acyclic Graph (DAG) is a tool that provides easy transformation on basic block.

1) **Leaf node** – implies name, identifier and constant. It is also known as exterior node.
2) **Interior node** – operator or result of expression.

Example 1:

X= a + b
Y = X + c
Z = X + Y



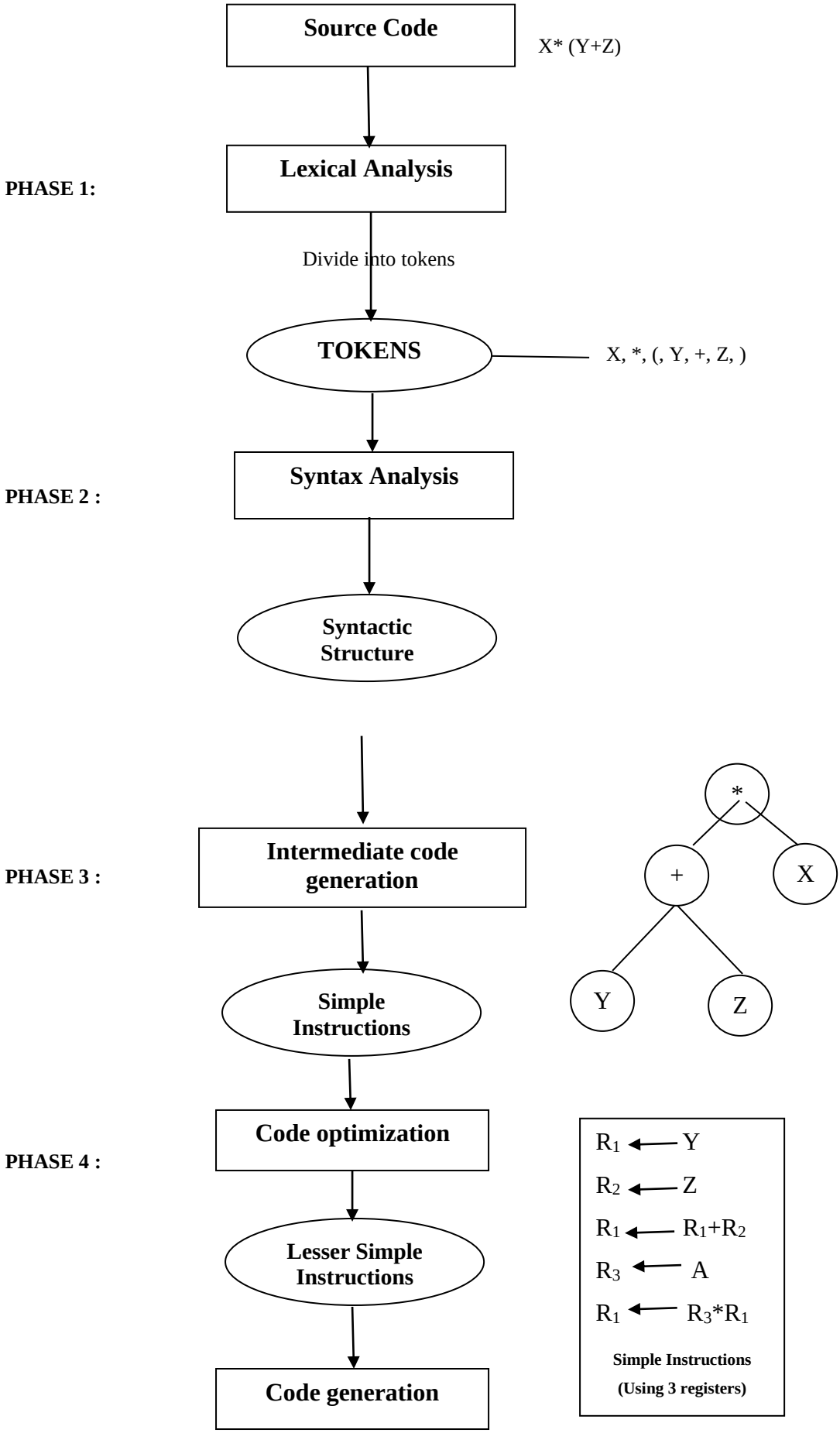
IJARCCE

ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264



IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 10, Issue 12, December 2021

DOI: 10.17148/IJARCCE.2021.101264

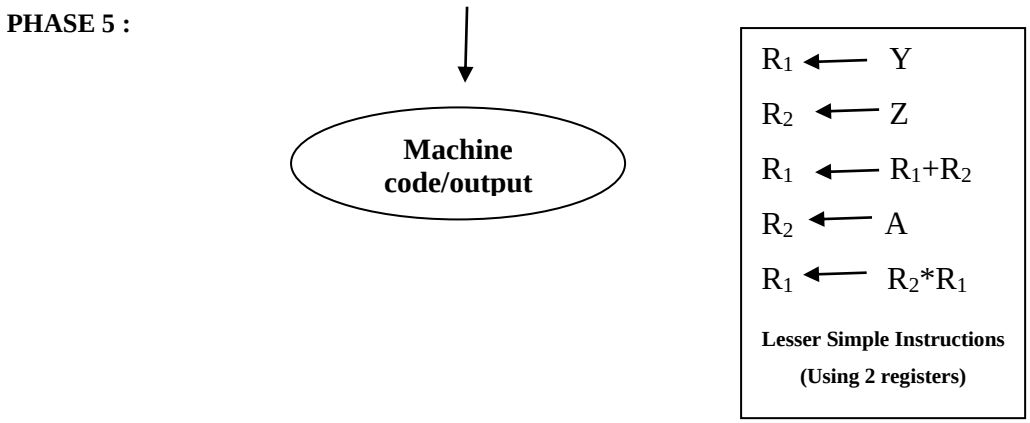


Fig 3: Detailed structure of compiler with its example of working

V. CONCLUSION

A compiler is an important computer program or software used to convert human-readable language into the machine understable language. It consists of various phases such as lexical analysis, syntax analysis, semantics analysis, code optimization, code generation. Each phase has its own and different working. In compiler technology, we need to provide higher level of automation such that we could meet the requirement of virtual machine and reduce the cost of compiler while reusing the code again and again to solve specific problem instead of rewriting the code. Optimizing the compiler is one of the complex tasks which must be efficiently and effectively done by the engineers by integrating various compiles into one form. There is further scope of development and improvement such that we can enable more languages implementation to deploy better interpreters and compilers and hence deliver better computer language performance to more users.

VI. ACKNOWLEDGEMENTS

Author is grateful to Agriculture University, Jodhpur for encouraging writing in my field and increasing the dynamics and knowledge in my stream.

VII. REFERENCES

[1] Alfred V. Aho. Monica S. Lam. Ravi Sethi. Jeffrey D. Ullman. Compilers: Principles, Techniques & Tools. Second Edition. Pearson Publisher
[2] Anuradha A. Puntambekar. Compiler Design. First Edition. Technical Publications.
[3] Ikvinderpal Singh. Compiler Design (Principles, Techniques and Tools). Khanna Publishers.
[4] Adesh K. Pandey. Fundamentals of Compiler Design. Kataria, S. K. & Sons Publisher.
[5] Dr. R, Venkatesh. Dr. N. Uma Maheshwari. Ms. S. Jeyanthi. Compiler Design. Publisher Yes Dee.
[6] Dr. Kavita A. Sultanpure. Compiler Design. Publisher Tec knowledge Publications Course Edition.
[7] Dr. B. S. Charulatha. Dr. J. Stanly Jayaprakash. Dr. A. Kanchana. Compiler Design. Publisher Charulatha Publications Private Limited.