Compiler Design

Computer Science
&
Information Technology (CS)

20 Rank under AIR 100

Postal Correspondence

- ✓ Examination Oriented Theory, Practice Set
- ✓ Key concepts, Analysis & Summary

ENGINEERS INSTITUTE OF INDIA
India's Best Institute for IES, GATE & PSUs
A Team of IES & GATE Toppers

CONTENT

1. INTRODUCTION OF COMPILER DESIGN & LEXICAL ANALYSIS 03-33
2. SYNTAX ANALYSIS 34-89
3. SYNTAX DIRECTED TRANSLATION 90-120
4. CODE GENERATION 121-145
5. GATE PRACTICE SET 146-153

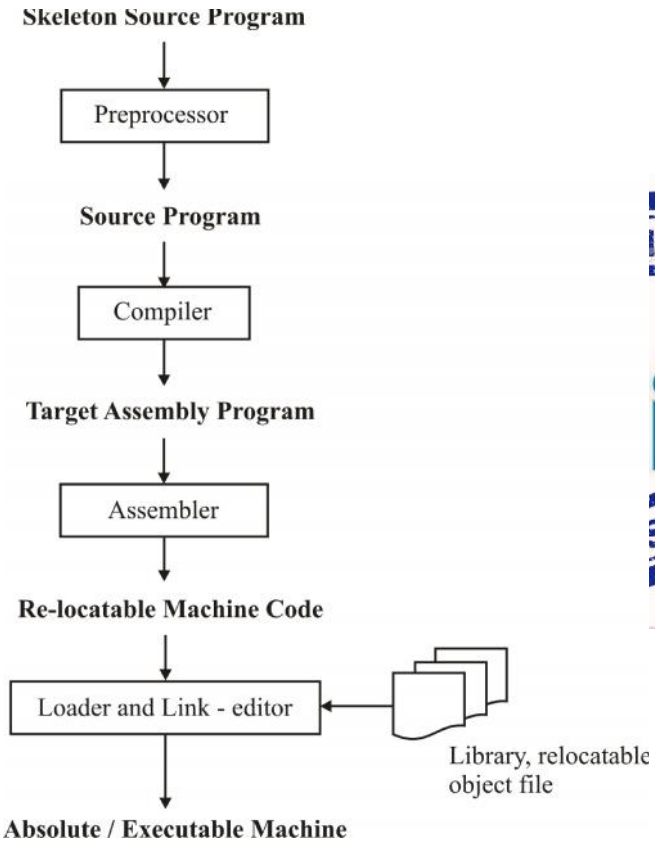
Syllabus:
Compiler Design: Lexical analysis, Parsing, Syntax directed translation, Runtime environments, Intermediate and target code generation, Basics of code optimization.

CHAPTER-1

INTRODUCTION OF COMPILER DESIGN AND LEXICAL ANALYSIS

Language Processing System

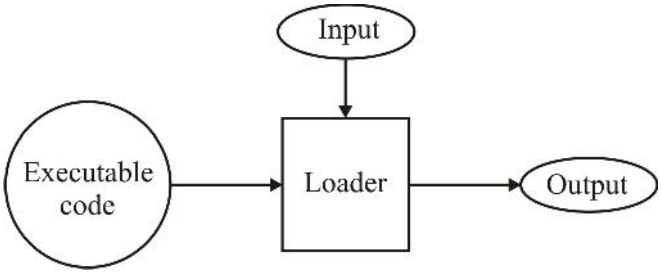
- Language processing is the ability of a computer to understand a program written in high level language by converting into executable program (binary code).
- These are the following steps in the language processing system:



After these steps, loader puts the program into the computer memory.

The program is run by operating system.

Input are read from I/O devices and from memory. Output is written to I/O devices and to memory. This is shown in below figure.



Now we will understand the each of the steps of language processing system.

1. **Preprocessor:**

   Preprocessor produce input to compiler.

   They may perform following function

   (a) **Micro processing:** A preprocessor may allow a user to define macro that are short hands fro longer constructs

   (b) **File inclusion:** A preprocessor may include header file into the program text. For example C processor causes, the context of the file < global.h > to replace the statement # include < global.h > when it processes a file, containing this statement.

   (c) **Rational Preprocessor :-** These processor augment older languages with more modern flow of control and data structuring facilities, for example, such a preprocessor might provide user with built in macros for construct like while statement or if – statements, where none exists in the programming language itself.

   (d) **Language Extensions :** These processors attempt to add capabilities to the language by what amounts to built I macros.

Fig: Role of Preprocessor.

2. **Assemblers :-** Some compilers produce the assembly code as output which is given to the assemblers as an input. The assemblers is a kind of translator which takes the assembly program as input and produces the machine code as output.

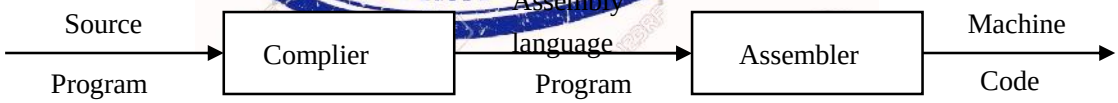


Fig: Role of assembler

An assembly code is a mnemonic version of machine code. The typical assembly instructions are given below.

```
MOV   a,R1
MUL   # 5,R1
ADD   #7, R1
MOV   R1,b
```

The assembler converts these instructions in the binary languages which can be understood by the machine. Such a binary code is often called as machine code. This machine code is a re-locatable machine code that can be passed directly to the loader / linker for execution purpose.

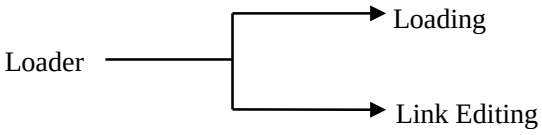
The assembler converts the assembly program to low level machine language using two passes. A pass means one complete scan of the input program. The end of second pass is the re-locatable machine code.

3. **Loaders and Link Editors :-**

A program called a loader performs loading and link editing. The process of loading consists of taking re–locatable machine code altering the re-locatable addresses and placing altered instruction and data in memory at the proper location.

Linker editor allows us to make a single program from several files of re-locatable machine code and planning the altered instruction and a data in memory at the proper location.

The link editor allows us to make a single program from several files of re-locatable machine code. These files may have been the result of several different compilations and one or may be library files of routine provided by the system and available to any program needs them



Translator

- Computer can understand only machine level language (binary, 0 & 1)
- It is difficult to write and maintain programs in machine level language. The programs written in the code of high level language and low level language need to be converted into machine level language using translator for this purpose.
- Translator are just computer programs which accept a program written in high level or low level language and produce an equivalent machine level program as output.
- Translators are of three types:
  - (i) Assembler
  - (ii) Compiler
  - (iii) Interpreter
- Assembler is used for converting the code of low level language (assembly language) into machine level language.

  Compiler and interpreter are used to convert the code of high level language into machine language. The high level program is known as source program and the corresponding machine level program is known as object program is known as object program. Although both compilers and Interpreters perform the same task but there is a difference in their working.
- A compiler searches all the errors of a program and lists them. if the program is error free then it converts the code of program into machine code and then the program can be executed by separate commands.
- All interpreter checks the error of a program statement by statement. After checking are statement, it converts that statement into machine code and then executes that statement. The process continuous until the last statement or program occurs.

Compiler

Compiler is a program which takes the source program written in high level language (C, C++, Java) as input and translates it into an equivalent another language C Assembly language)

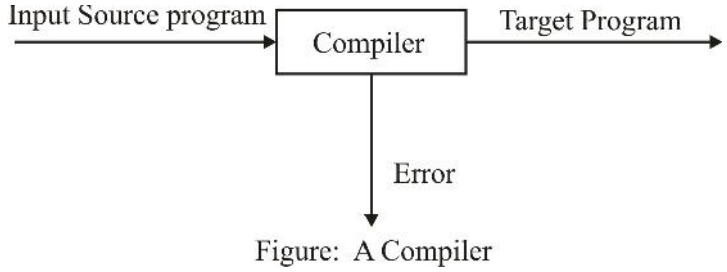


Figure: A Compiler

Note: Output of compiler may be assembly code or re-locatable object code or even an executable program or if something goes wrong, output is a bunch of error message.

→ During the compilation if some error are encountered the compiler displays the error messages.

The compile time error include three errors.

(i) Syntax error

(ii) Type checking errors

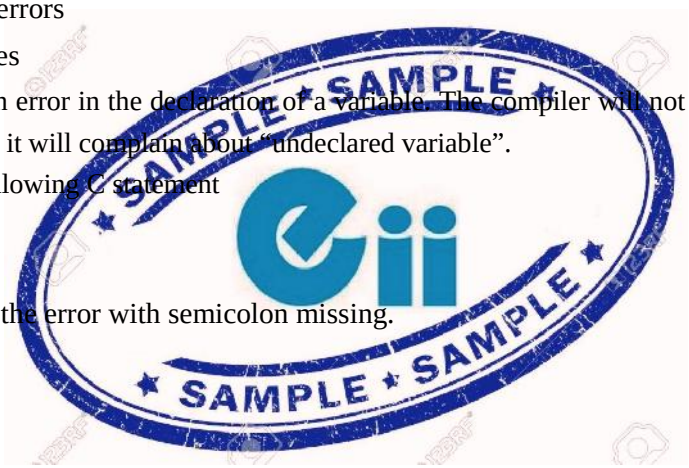
(iii) Compiler crashes

Example 1: If there is an error in the declaration of a variable. The compiler will not recognize the variable as such when it is used, and it will complain about "undeclared variable".

Example 2: Consider following C statement

```
int c
```

– Here compiler give the error with semicolon missing.



Types of Compiler

(1) Cross – compiler

→ It is a compiler which generates code for a platform different than the platform on which the compiler itself runs.

→ Compilers for embedded targets are almost always cross-compiles, since few embedded targets are capable of hosting the compiler itself.

→ Cross compilers requires a build system which does not assume that the host and target system are compatible, i.e., you cannot run a target executable at build time.

→ Cross-compilation can also be applied to the compile itself. This is referred to as Canadian cross compilation, which is a technique for building a cross compiler on host different from the one the compiler should be run on. In this case we have three platforms.

(i) The platforms the compiler is built on (build).

→ The platform which hosts the compiler (host).

→ The platform for which the compiler generates code (target)

Cross platform

It is refers to the things which are working on many different platforms (different operating system or different processor or bit size)

Incremental Compiler

– An incremental compiler is one that can recompile only those portion of a program that have been modified.

– Ordinary compilers must process entire modules or programs.

Anatomy of compiler

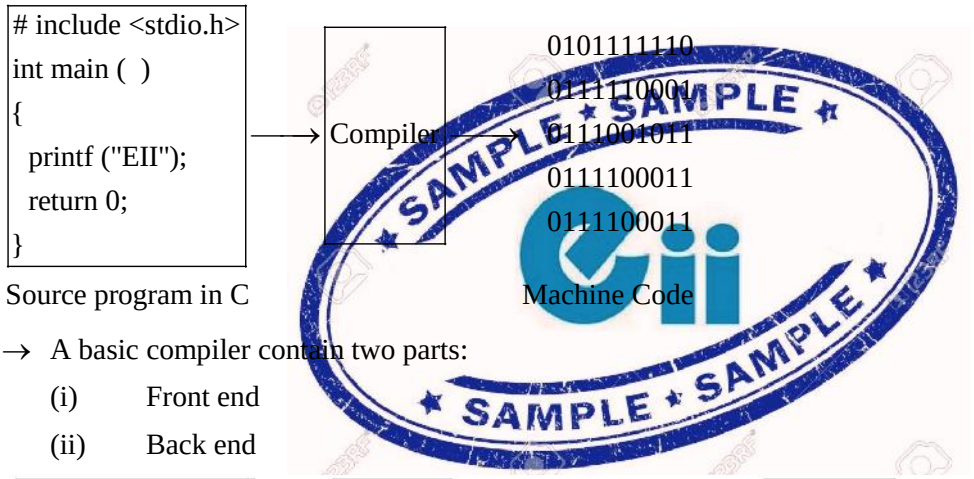
– Computer's can't directly understand our program (written in c, c++, Java).

– Computer's only understand machine code (sequences of instructions expressed as ones an zeroes.

⇒ Compiler are programs that translates our program into machine code that a computer can understand.

Note: Compiler output can be assembly code/machine code or reloctable object code.

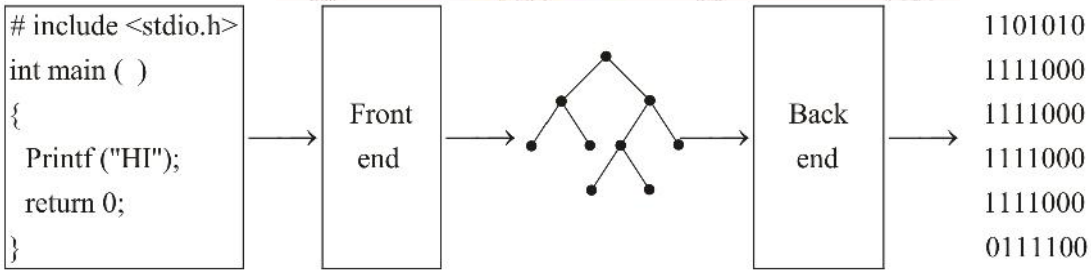
Example:



→ A basic compiler contain two parts:

(i) Front end

(ii) Back end

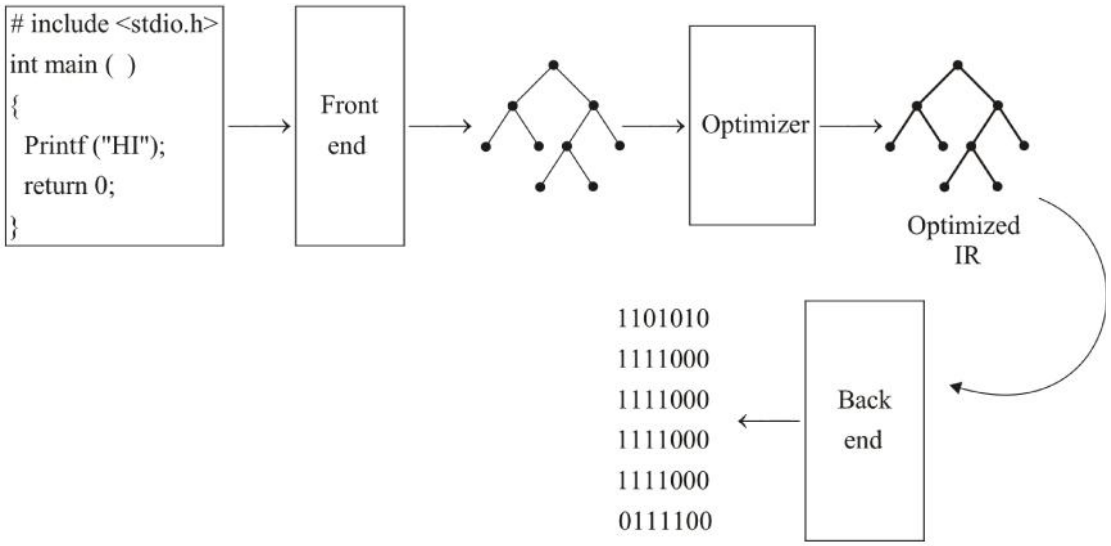


– The front end takes program code and converts it to an intermediate code.

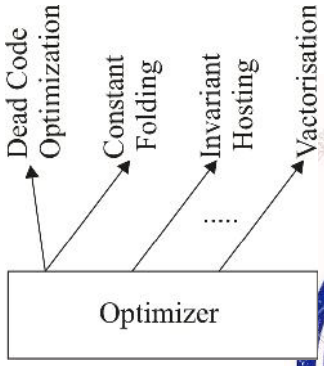
– The back end takes intermediate e code and converts it to machine code.

– Some compiler output will be assembly code

– An optimizer can be used to transform intermediate code to make it more efficient.



Note: Optimizers are not monolithic they often consists of 20 or more self contained optimization phases.



The Phases of Complier :

As we have discussed earlier the process of compilation is carried out in two parts. "Analysis and synthesis". Again the analysis is carried out in three phases: lexical analysis, syntax analysis and semantic analysis. And the synthesis is carried out with the help of intermediate code generation, code generation and code optimization. Let us discuss these phases one by one.

1. **Lexical Analysis :-**
   - → This lexical analysis is also called scanning.
   - → It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group of strings called token
   - → A token is a sequence of characters having a collective meaning.
   - → For example if some statement in your source code follows,

     total = count + rate *10

     Then in lexical analysis phase this statement is broken up into series of token as follows.

1. **The identifier** total
2. **The assignment symbol**
3. **The identifier** count
4. **The plus sign**
5. **The identifier** rate
6. **The multiplication sign**
7. **The constant** number 10

Note: The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

2. **Syntax Analysis:-**

→ The syntax analysis is also called parsing

→ In this phase the tokens generated by the lexical analyzer are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the tokens together.

→ The hierarchical structure generated in this place called parse tree or syntax tree.

→ For the expression total = count + rate ∗ 10 the parse tree can be generated as follows.

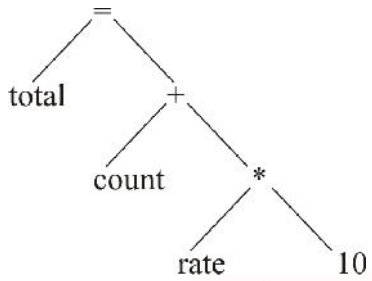


Figure Parse tree for total = count + rate∗10. In this statement 'total = count + rate∗10' first of all rate∗10 will be considered because in arithmetic expression the multiplication operation should be performed before the addition. And then the addition operation will be considered. For building such type of syntax tree the production rules are to be designed. The rules are usually expressed by context free grammar. For the above statement the production rules are

(1) E ← identifier

(2) E ← number

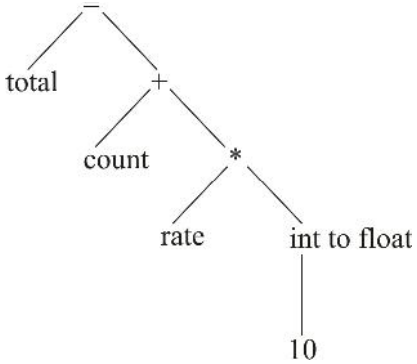
(3) $E \leftarrow E_1 + E_2$

(4) $E \leftarrow E_1 * E_2$

(5) E ← (E)

3. **Semantic Analysis :-**

Once3 the syntax is checked in the syntax analyser phase the next phase i.e the semantic analysis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if …… else statements or performing arithmetic operations of expressions that are type for example



Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

4. **Intermediate Code Generation :**

The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is variety of forms such as three address code, quadruple, triple, posix. Here we'll consider an intermediate code in three address code form. This is like an assembly language. The address code consists of instructions each of which has at the most three operands. For example

$t_1 := \text{int to float}$

$t_2 := \text{rate} \times t_1$

$t_3 := \text{count} + t_2$

$\text{total} := t_3$

There are certain properties which should be possessed by the three address code and those are,

1. Each three address instruction has at the most one operator in addition to the assignment. Thus the complier has to decide the order of the operations deviced by the three address code.
2. The complier must generate a temporary name to hold the value computed by each instruction.
3. Some three address instructions may have fewer than three operands. For example first and last instruction of the above given three address code. i.e

   $t_1 :=$ int to float (10)

   $\text{total} := t_3$.

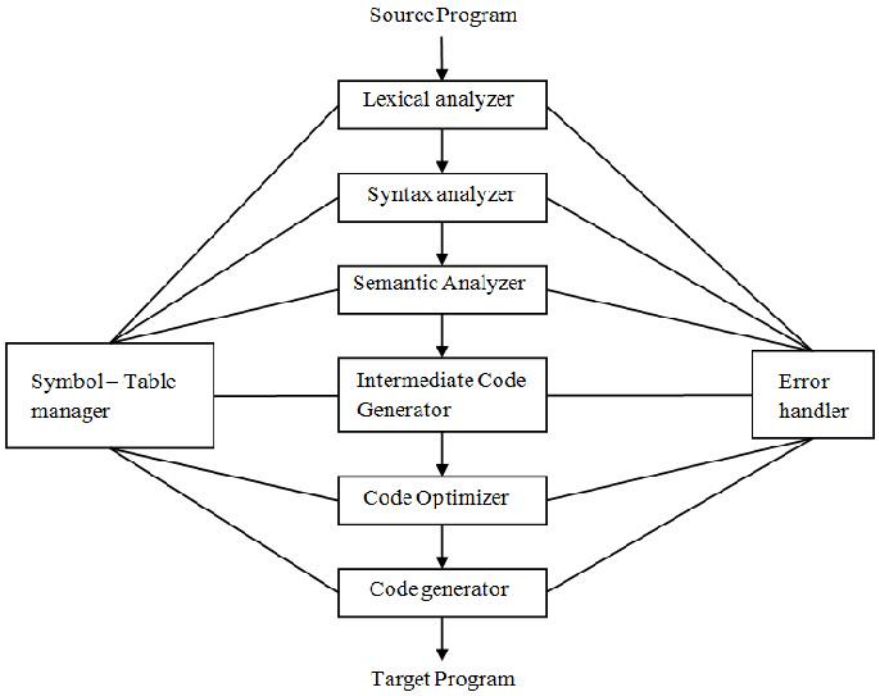
5. **Code Optimization :-**

The code optimization phase attempts to improve the intermediate code. This is necessary to have faster executing code or less consumption of memory. Thus by optimizing the code the overall running time of the target program can be improved.

6. **Code Generation :-**

In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions

```
MOV   rate, R1
MUL   # 10.0,R2
MOV   count, R2
ADD   R2, R1
MOV   R1,total
```



Symbol Table Management:

→ To support these phases of complier a symbol table is maintained. The task of symbol table is to store identifiers (variables) used in the program

→ The symbol table also stores the information about attributes of variable used in the program. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it.

→ The symbol table also stores information about the 'Subroutines' used in the program. In case of subroutine, the symbol table stores the name of the subroutine, no. of arguments passed to it, type of these arguments, the method of passing these arguments (may be call by value or call by reference) return type if any.

→ Basically symbol table is a data structure used to store the information about identifiers.

→ The symbol table allows us to find the record for each identifier quickly to store or retrieve data from that record efficiently.

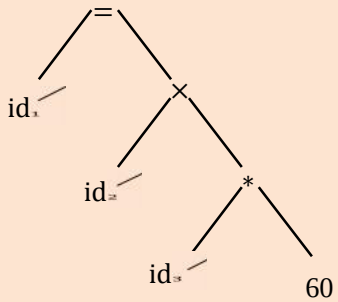
→ During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer cannot determine all the attributes are entered by remaining phases of complier.

→ Various phases can use the symbol table in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what type of identifiers are. Then during code generation typically information about how much storage is allocated to identifier is seen.

Error detection and handling:-

As programs are written by a human beings therefore they can't be free from errors. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form of "message". When the input characters from the input do not form the token, the lexical analyzer detects it as error. Large no. of errors can be detected in syntax analysis phase. Such errors are popularly called as "Syntax Errors". During semantic analysis ; type mismatch kind of error is usually detected.

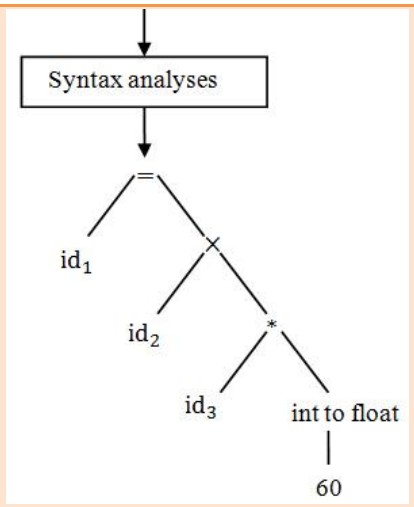
Example:- Show how to an input a = b + c * 60 get processed in compiler. Show the output at each stage of complier. Also show the contents of the symbol table.

	Input processing in complier	**Output**
	a = b + c * 60 ↓ Lexical analyser ↓ $id_1 = id_2 + id_3 * 60$ ↓ Syntax analyser ↓ 	Token stream

a	.....
b	.....

c	.....
⋮ ⋮ ⋮	.....

Symbol Table



Syntax Tree

Intermediate code generator

$t_1 := \text{int to float}(60)$

$t_2 := id_3 * t_1$

$t_3 := id_2 * t_2$

Code optimizer

$t_1 := id_3 * 60.0$

$id_1 := id_2 + t_1$

Code generator

MOVF id3, R2

MOVF # 60.0, R2

MOVF id2, R1

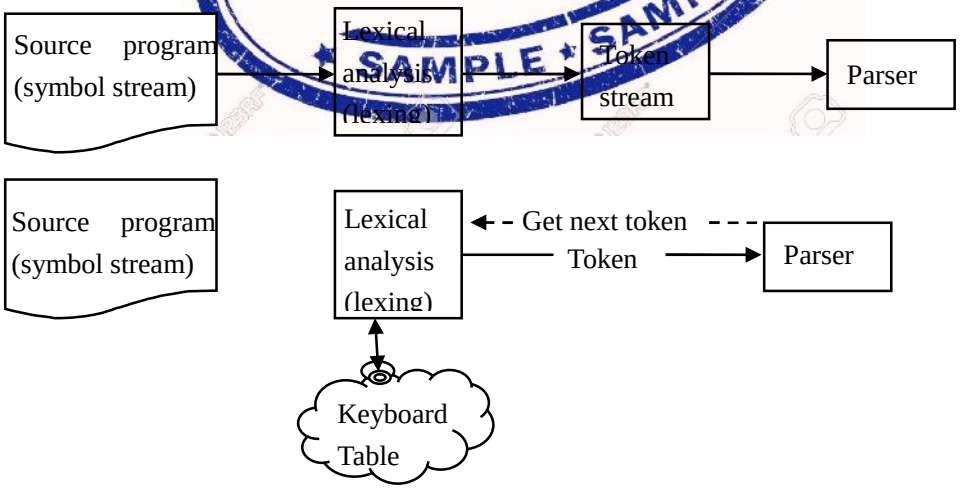
Semantic tree

		Intermediate code Optimized code Machine code

LEXICAL ANALYSIS

Scanning Perspective :-

→ Purpose transform a stream of symbols into a stream of tokens.



Lexical Analyzer Responsibilities :-

Lexical analyzer (Scanner)

→ Scan input
→ Remove white spaces
→ Remove comments
→ Manufacture tokens
→ Generate lexical errors
→ Pass token to parser

Modular design :-

Rationale

→ Separate the two analysis
  - High cohesion | low coupling

→ Improve efficiency
→ Improve portability / maintainability
→ Enable integration of third – party lexers

Lexer = lexical analysis tool.

Lexemes : Smallest logical units (words) of a program, such as A, B, 1.0, true, +, < = ……..

Tokens : classes of similar lexemes, such as identifier, constant, operator

Pattern : An informal or formal description of a token, such as "an identifier" is a string of atmost 8 characters, in which the first character is an alphabet, and the successive characters are either digits or alphabets.

Examples :-

Token	Sample lexemes	Informal Description of pattern
const	const	const
if	if	if
relation	<,< =, =, < >, >, > =	< or < = or = or < > or > = or >
id	p[illegible]ount, D[illegible]	letter followed by letters and digits.
num	3. 1416, 0, 0 6.02 E 23	any numeric constant any characters between "and" & "except"