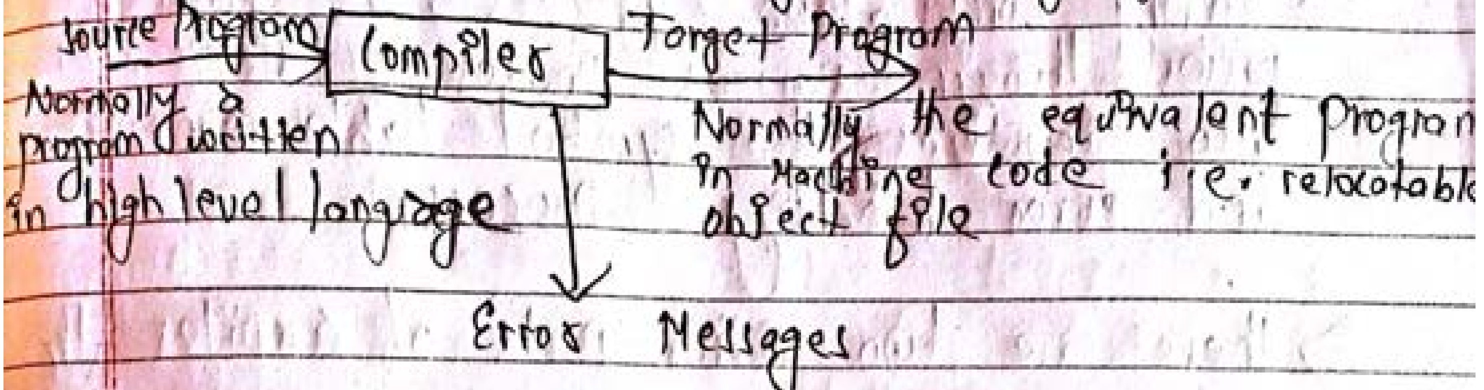
Chapter → 1

Introduction to Compiler

A compiler is a language translator software programs that takes its input in the form of program written in one particular programming language (source language) and produces the output in the form of program in another language (object or target language).



Source Program
Compiler
Target Program
Normally a program written in high level language
Normally the equivalent program in Machine code i.e. relocatable object file
Error Messages

→ The compiler is a special type of computer program that translates human readable text file into a form that a computer can more easily understand.

→ At its most basic level, a computer can only understand two things i.e. 0 and 1. At this level, a human will operate very slowly and find the information contained in the long string of 0's and 1's incomprehensible. A compiler is actually a computer program that bridges this gap.

(#) List of compilers:

→ ALGOL → Ada compilers → BASIC compilers

→ C compiler → C++ compilers → COBOL com
→ Pascal compilers → FORTRAN compilers →
JAVA compilers.
→ Python compilers → C# compilers.

(#) Phases of compilers: (or steps)
→ During compilation process, a program passes through various phases or steps. It also involves the symbol table management and error handling process

→ There are two major parts of compiler & they are:-
(1) Analysis part
(2) Synthesis part

→ In analysis part, an intermediate representati is created from the given source program. This part of compiler is also called front end part of the compiler. It mainly consist of following four parts:-
i) Lexical Analysis
ii) Syntax Analysis
(iii) Semantic Analysis
(iv) Intermediate code generation

→ In synthesis part, the equivalent target progr

is created from intermediate representation of a programme created by analysis part. This part of compiler is also called "backend" of the compiler. This part mainly consist of following two phases:-
(i) Code optimization
(ii) Final Code Generation.

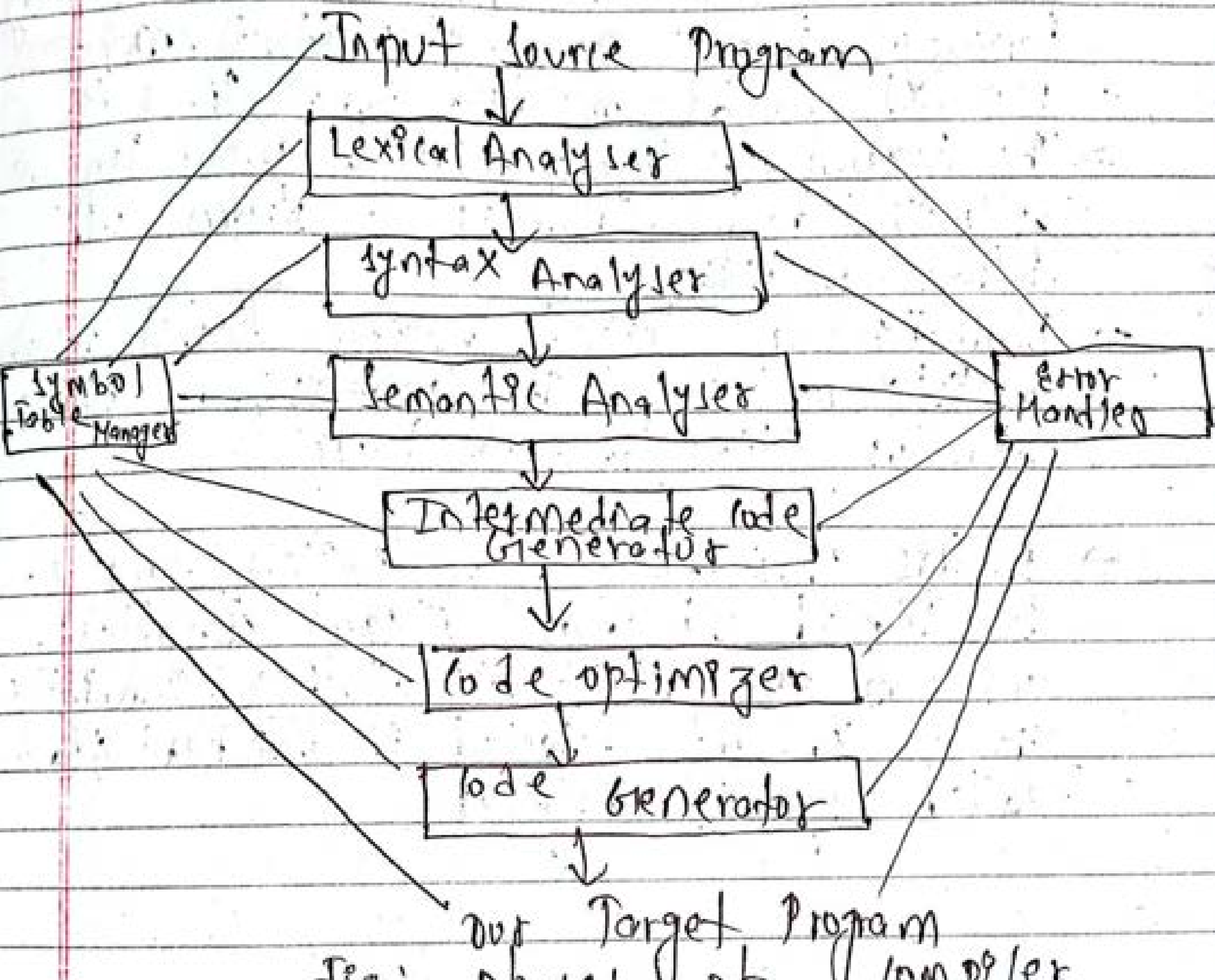


Fig: phases of compiler

Date.............
Page.............

⇒ The phases of compiler illustrated in the given figure are discussed below :-

(i) Lexical Analyzer (scanning) :

→ Lexical Analysis or scanning is the process where the source program is read from the left to right and grouped into "tokens".

→ Tokens are the sequences of characters with a collective meaning. In any programmin language, tokens may be constants, operators, reserved words words, punctuations, etc.

→ The lexical analyzer takes a source program as input and produces a stream of tokens as output.

→ Normally, a lexical analyzer doesn't return a list of tokens ; it returns a token separator (like semicolon) etc.

→ In this phase, only few limited errors can be detected such as illegal characters, unrecognized symbols, etc. Other remaining errors can be detected in the next phase, called syntax analyser.

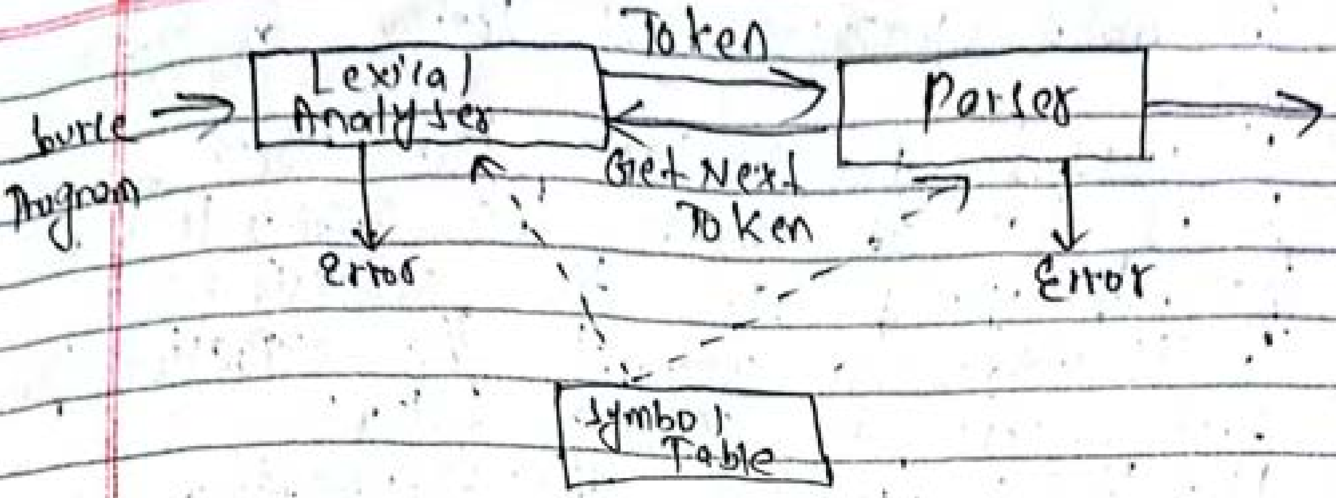


Fig: Lexical Analysis in Compiler Design

Example:
while (i>0)
i=i-2

Tokens	Description
while	while keyword
(	left parenthesis
i	identifier
>	greater than symbol
0	constant
)	right parenthesis
i	identifier
=	equals
i	identifier
-	minus
2	integer constant
;	semi colon.

Date:
Page:

⇒ The main purposes of lexical analyzer are :-

(i) It is used to analyse the source code.

(ii) It is able to remove the comments and the whitespaces in the expression.

(iii) It is used to format the expression for easy access i.e. create tokens.

(iv) It begins to fill the information in the symbol table.

(2) Syntax Analyzer (or Parser)

- The second phase of the compiler design is syntax analysis.
- Once lexical Analysis is completed

- A syntax analyzer creates the "syntactic structur generally called a "parse tree" of the given source program.
- syntax analyzer is also called the "parser" an its job is to analyze the source programme based on the definition of its syntax.
- Thus, it is responsible for creating a parse

tree of the source code.
- A syntax of a language is specified by context Free Grammar (CFG). The rules in CFG are mostly recursive.

The main purposes of Syntax Analyzer are:-
(i) It analyzes the tokenized code for the structure
(ii) It is able to groups with "typed information"

(#) Demonstration
newval := old val + 12

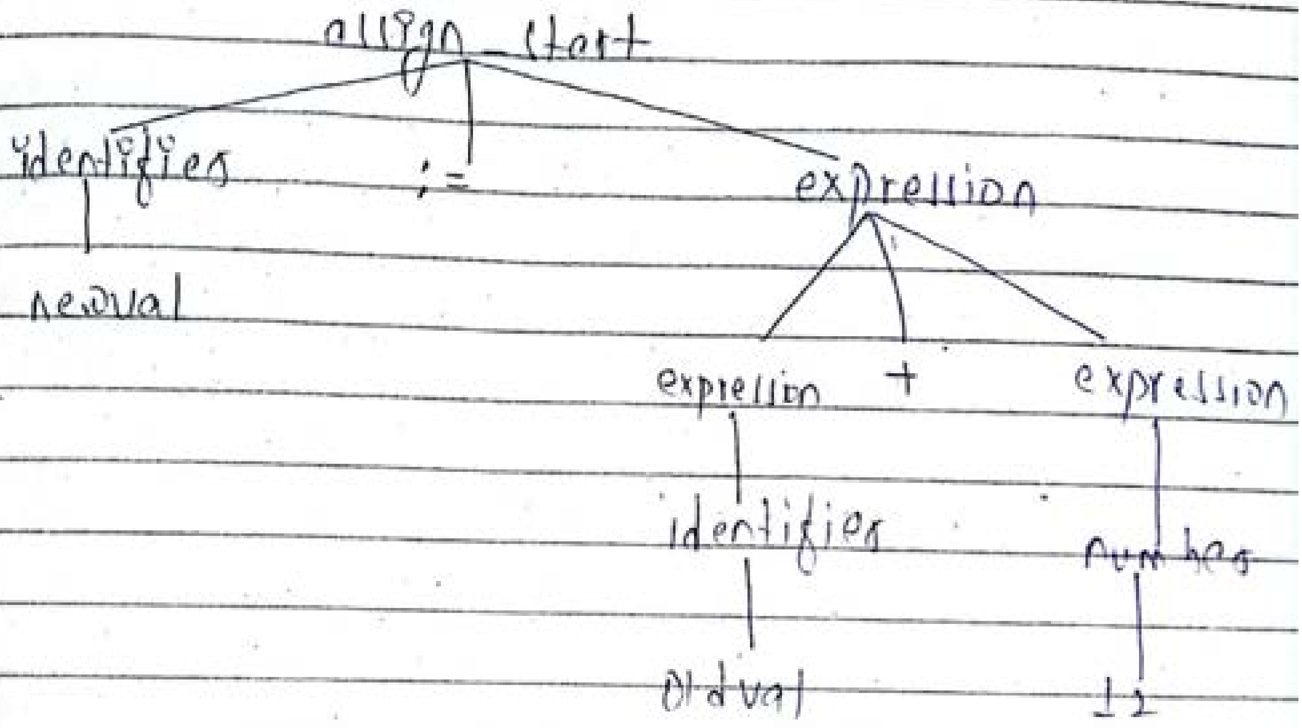


Fig: A demonstration of a parse tree.

→ In parse tree, all the terminals are at leaves and all the inner nodes are non terminals in a CFG

Date.........
Page.........

(3) Semantic Analyzer.

→ The next phase is the semantic analyzer, which performs a very important role to check the semantics (i.e. meaning) of the rules of the expression according to the source language. The previous phase output i.e. syntactically correct expression is the input of the semantic analyzer.

→ It is required when the compiler may require performing some additional check such as determining a type of expressions and checking that all the statements are correct with respect to the typing rules, that variables have been properly declared before they are used, the functions are called with the proper numbers of parameters, etc.

→ This phase is carried out using the information from the parse tree and symbol table.

→ The parsing phase only verifies that the program consists of tokens arranged in a syntactically valid combination. Now, semantic analyzer checks whether they form a sensible set of instructions in the programming language or not.

(*) Examples of semantic Analyzer.

→ The "type" of the right hand side expression of the assignment statement should match the type of the left-side expression.
i.e. newval := oldval + 12
The type of the expression (oldval + 12) must match with the type of variable (newval)
→ The parameter of a function should match the arguments of a function call in both number and type.
→ The variable name used in the program must be unique, etc.

(X) The main purposes of semantic analyzer are :-
(i) It is used to analyse the parsed code for meaning.
(ii) It fills in assumed or missing information.
(iii) It tags groups with meaningful information.

(4) Intermediate Code Generator.
→ If the program is syntactically and semantically correct then intermediate code generator generates a simple machine-independent language.
→ The intermediate language should have the following two properties.
(i) It should be simple and easy to produce.
(ii) It should be easy to translate the target program.

Date.........
Page.........

→ The main purpose of Intermediate Code (
ration is used to generate the intermedia
Code of source code.

→ Some compilers may produce an explicit inte
mediate code representing the source code.
These intermediate codes are generally
machine independent but the level of interm
ediate code is close to the level of
machine codes.

→ Intermediate code generation is done by
the use of three address code generation
For Example:

$a = b + c * d / f$

→ Sol^n
Intermediate code for above example is
represented by three address code as follow:-

$t_1 = c * d$
$t_2 = t_1 / f$
$t_3 = b + t_2$
$a = t_3$

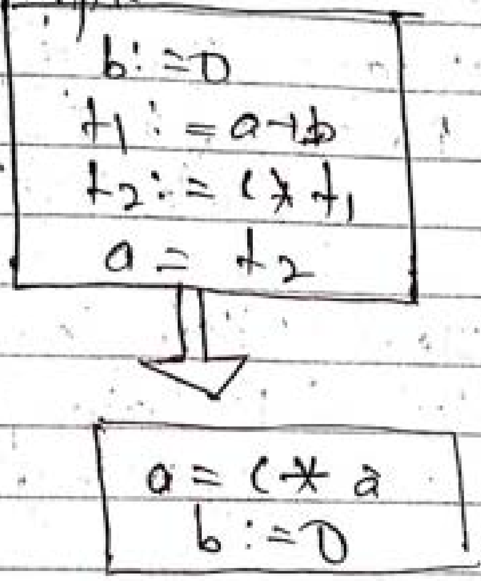
(5) Code optimization:

→ Optimization is the process of transform
ing a piece of code to make more effici
(either interms of time or space) without
changing its output or side effects.

→ The process of removing unnecessary part

of code is known as code optimization.
⇒ Due to code optimization process, it decreases time and space complexity of a program i.e.
(i) Detection of redundant function calls.
(ii) Detection of loop-invariants
(iii) Common sub-expression elimination.
(iv) Dead code detection and elimination.

⇒ The main purpose of code optimization is that it examines the object code (target code) inorder to determine whether there are more efficient means of execution.
For example:



```
b := 0
t1 := a+b
t2 := c * t1
a = t2
```

```
a = c * a
b := 0
```

(6) Code Generator
⇒ It generators assembly code for the target CPU from an optimized intermediate code

representation of the program.
→ For example:
Assume that we have an architecture with instructions in which at least one of its operands is a machine register.

A = b + c * d / f.

```
MOVE  c, R1
MULT  d, R1
DIV   f, R1
ADD   b, R1
MOVE  R1, A
```

(#) One pass compiler vs Multi-Pass Compiler

One Pass Compiler	Multi-Pass compil
(i) Here, all the phases are combined into one pass.	Here, different phase of compiler are groupe into multiple phases.
(ii) Here, intermediate representation of source code is not created.	Here, intermediate representation of source code is created.
(iii) It is also called narrow compiler.	It is also called wide compiler.
(iv) It is faster than Multipass compiler.	It is slightly slower than one pass compiler

(v) A single Pass compiler takes more space then Multi-pass compiler.	A Multi-Pass compiler takes less space because in multi-pass compiler, the space used by the compiler during one pass can be reused by the subsequent pass.
(vi) Pascal's compiler is an example of one Pass compiler	C++ compiler is an example of Multi Pass compiler.

(#) Symbol Table:

→ They are the data structure that are used by the compilers to hold information about source program constructs.

→ The information is collected incrementally by the analysis phase of compiler and used by the synthesis phase to generate the target code.

→ Entries in the symbol table contain information about an identifier, such as its type, position in storage and any other relevant information.

→ Symbol tables typically need to support multi the declaration of some identifier with in a program.

→ The lexical analyser can create a symbol table entry and can return token to a parser, say (Id), along with a ~~the~~ pointer to the lexeme.

→ Then the parser can decide whether to use a previously created symbol table or create new one for the identifier.

⇒ The basic operations defined on a symbol table include:

(i) allocate → To allocate a new empty symbol table.

(ii) free → To remove all the entries and free the storage of a symbol table.

(iii) Insert → To insert a name in a symbol table and return a pointer to its entry.

(iv) lookup → To search for a name, and return a pointer to its entry.

(v) set-attribute → To associate an attribute with a given entry.

(vi) get-attribute ⇒ To get an attribute associated with a given entry.

⇒ The possible entries in a symbol table are as follows :-

(i) Name : a string
(ii) Attribute :
- Reserved word
- Variable name
- Type name
- Procedure name
- Constant name
- - - - - - - - -

(iii) Data Type
(iv) Scope information : where it can be used
(v) Storage allocation, size, ...
- - - - . . . .

Example : Let us take a portion of a program as follows :-

```
  void  fun ( int A, float B)
{
      int D, E;
       D = 0;
      E = A 1 round (B);
     if ( E>5)
```

Date.........
Page.........

```
    {
        print D;
    }
 }
```

⇒ It's symbol table is created as follow

Symbol	Token	DataType	Initialization
fun	Id	function name	No
A	Id	Int	Yes
B	Id	Float	Yes
D	Id	Int	No
E	Id	Int	No

Also, we can write it as :-

Symbol	Token	DataType	Initialization
fun	Id	function name	No
A	Id	Int	Yes
B	Id	Float	Yes
D	Id	Int	Yes
E	Id	Int	Yes

8) Error Handling in compiler:

→ Error detection and reporting of errors are the important functions of the compiler. whenever an error is encountered during the compilation of the source program, an error handling is invoked.

→ Error Handler generates a suitable error reporting message regarding the encountered error. The error reporting message allows the programmer to find out the exact location of error.

→ Errors can be encountered at any phase of the compiler during compilation of the source program for several reasons, such as:

(i) In lexical Analysis phase, errors can occur due to the misspelled tokens, unrecognized characters, etc. These errors are mostly the typing errors.

(ii) In Syntax Analysis phase, errors can occur due to the syntactic violation of the language.

Date.........
Page.........

(iii) In Intermediate Code Generation Phase, errors can occur due to the incompatibility of operands, type for an operator, etc.

(iv) In code optimization phase errors can occur during the control flow analysis due to some unreachable statements.

(v) In Code Generation Phase errors can occur due to the incompatibility with the computer architecture during the generation of code.

(vi) In symbol table, errors can occur during the book-keeping routine, due to the multiple declaration of an identifier with an ambigouis attribute.

(#) Language Processing System:

→ We know that, any computer system is made up of hardware and software.
→ The hardware understands a language which humans cannot understand. So, we write Program in high level language which is easier for us to understand and remember.
→ These programs are then fed into a series

CamScanner

of tools and of components to get the desired code that can be used by the machine. This is known as language processing system.

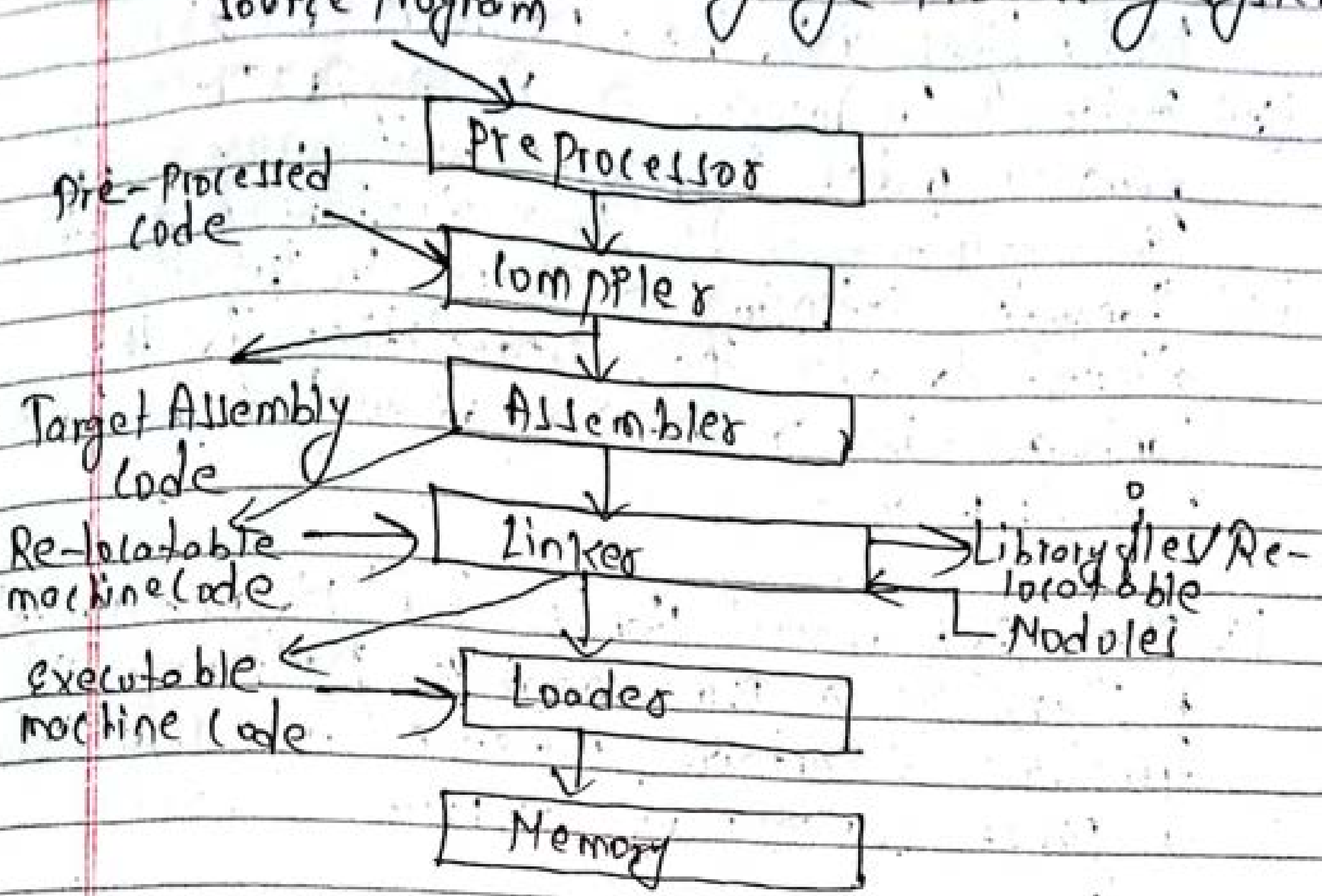


Fig: Language Processing System.

(#) Pre Processor ⇒ A preprocessor, generally considered as a part of compiler, is a tool that produces input for the compilers.

→ PreProcessor deals with macro-processing, augmentation, file inclusion, language extension, etc.

→ Pre Processor may perform the following functions:

(i) Macro processing :→ A preprocessor may allow a

user to define macros that are shorthands fo
the longer constructs.
(ii) File inclusion ⇒ A Preprocessor may include header files into the program text.
(iii) Rational Preprocessor ⇒ These preprocessors augment older languages with more modern flow-of-control and data structuring faciliti
(iv) Language Extension ⇒ These Preprocessors attempt to add the capabilities to the language by a certain amount to the buil In-macros.

(#) Macros ⇒ A "macro" stands for "Macro-instructi A Macro is a programmable pattern which translates a certain sequence of input in a preset sequence of output.

→ Macros can make tasks less repetitive by representing a complicated sequence of key strokes & mouse movements, commands or other types of inputs.

→ A Macro-instruction makes writing the code more convenient.

→ For every occurrence of macro, the whole macro body or macro block of statements, gets expanded in the main source code.

(#) Features of Macro-Processor:

(i) Macro represents a group of commonly used statements in the source programming language.
(ii) Macro-Processor replaces each macro instruction with the corresponding group of source language statements. This is known as expansion of macros.
(iii) Using macro instructions, programmer can leave, the mechanical details to be handled by the macro-processor.
(iv) Macro Processor design is not directly related to computer Architecture on which it runs.
(v) Macro Processor involves definition, expansion, invocation, etc.

(#) Compiler Construction Tools
→ For the construction of compiler, the compiler writer uses different types of software tools that are known as compiler construction Tools.

→ These tools make the use of specialized languages for specifying and implementing specific components, and most of them use

Sophisticated algorithm.
→ These tools should hide the details of the algorithm used and produce component in such a way that they can be easily integrated into the rest of the compiler.

⇛ Some of the most commonly used compiler Construction Tools are :-
(i) Scanner Generators
(ii) Parser Generators
(iii) Syntax - Directed Translations Engines
(iv) Code Generators
(v) Data - flow Analysis Engines.
(vi) Compiler - Construction Toolkits.

(i) Scanner Generators :- They automatically produ lexical analyzers or scanners. For Example: flex lex.

(ii) Parser Generators :- They produce syntax - analyzers or parsers. for Example Bison, Yacc.

(iii) Syntax - Directed Translations Engines
→ They produce a collection of routines, which traverse the parse tree and generates the intermediate code.

(iv) Code Generators :- ⇒ They produce a code generator from a set of rules that translates the intermediate language instructions into the equivalent machine language instructions for the target machine.

(v) Data-flow Analysis Engines ⇒ They gather the information about how the data is transmitted from one part of the program to another.
⇒ For code optimization, Data-flow Analysis is a key part.

(vi) Compiler - Construction Toolkits.
⇒ They provide an integrated set of routine for the construction of the different phase of the compiler.