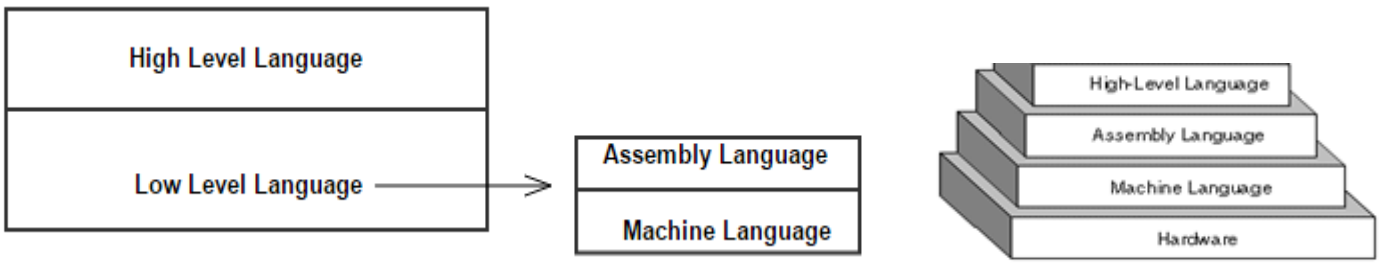
Levels Of Languages



High Level Language: high-level programming languages, are generally portable across multiple CPU .Examples FORTRAN, ALGOL ,COBOL, C ,C++ , …….

Low –Level Languages: **A machine language and assembly language**is known as Low-Level Language.

Assembly Language(Symbolic Language): In assembly language we use **alphanumeric** code instead of **numeric** codes. EXAMPLE:- using **ADD** instead of **1110**

Machine Language (Binary Language or object code):- Strings of **0s and 1s** .The only difficulty is to learn it and to edit it if errors occur . A machine language instruction has two parts

Opcode	Operand

Low-level languages (machine ,assembly) are closer to the hardware whereas high-level programming languages are closer to human languages

*Computer only understand **Machine language .** So all the languages need to be converted to Machine language.*

Asembler,Compiler&Interpreter,Linker,Loader

Assembler: **Assembly language** is converted into **machine Language** by assembler .

Compiler and Interpreter: Both compiler and Interpreter translates a high level language program into a machine language program . Both are intelligent than an assembler. Compiler can translate only those programs that have been written in a language for which the compiler is meant **i.e** Fortran compiler can translate programs written in Fortran . **(** In old days compiler translates a high level language into assembly language . But now a days both compiler & Assembler are merged together so high level language is directly converted to machine language **)**

Interpreter	**Compiler**
An interpreter is a program which reads only one statement of program, translates it Then it reads the next statement of the program again translates. In this way it proceeds further till all the statements are translated.	A compiler goes through the entire program and then translates the entire program into machine codes.
An interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system which has limited memory space.	
	By the compiler, the machine codes are saved permanently for future reference whereas it is not saved in case of interpreter .Therefore there is no need to repeat the compilation process every time we wish to execute the program only it is required whenever the program is modified
	Compiler is 5 to 25 times faster than an interpreter

Linker: In high level languages functions are linked to the ***header files or libraries*** by a program called Linker. If Linker does not find a header file of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program .

Loader: Loader is a program that loads machine codes of a program into the system memory.

Infix Prefix Postfix

Infix L(R)R

Prefix (R)LR

Postfix LR(R)

Infix to any
Postfix to any
Prefix to any
→ Tree बनाके

Constructing Tree with the Infix

Paranthise the given Infix expression with the help of priority, associativity.

सबसे पहले paranthise the operands which are along with highest priority operator i.e paranthise operand in decreasing order of priority. If priority is same then check for associativity.

Highest priority :- exponential ($, ^)

Next Highest priority :- *, /

Lowest priority :- +, -

exponential operator is right associative
remaining 4 are left associative

Draw tree from the Infix given

$\wedge * + -$

$\wedge$ (right associative remaining left)

$a+b*c-d\wedge e\wedge f$

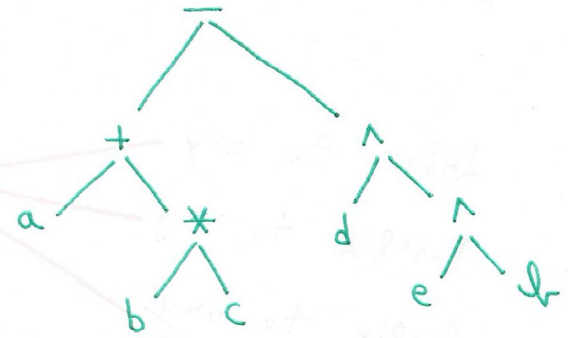
$a+b*c-d\wedge(e\wedge f)$

$a+b*c-(d\wedge(e\wedge f))$

$a+(b*c)-(d\wedge(e\wedge f))$

$(a+(b*c))-(d\wedge(e\wedge f))$

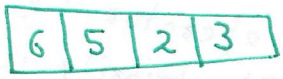
In one step only one parenthesis is introduced

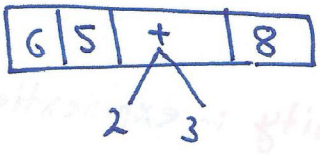


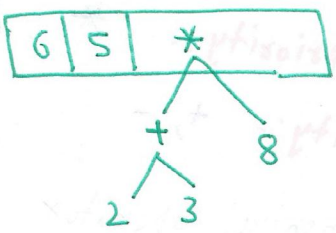
Constructing Tree with Postfix

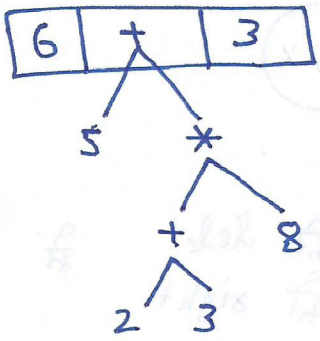
Q

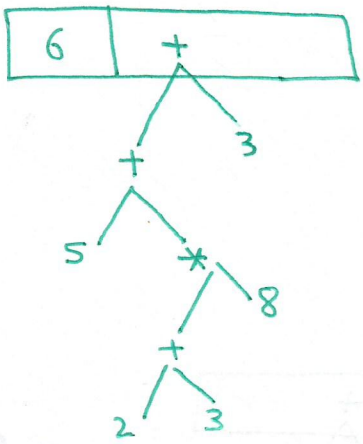
6523+8*+3+*

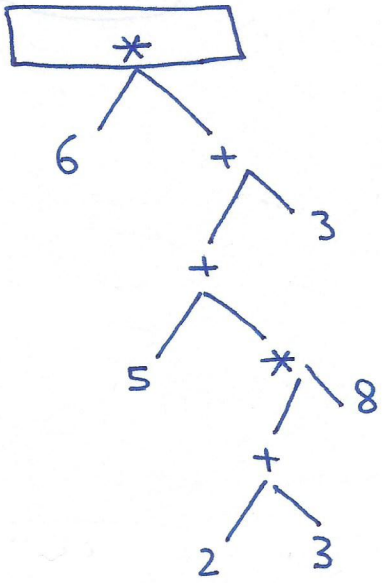







6
+
3
5
*
+
8
2
3


6
+
3
+
5
*
8
+
2
3


*
6
+
3
+
5
*
8
+
2
3

Constructing Tree with Prefix

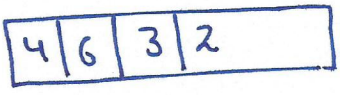
Step1:- Reverse the given Prefix expression

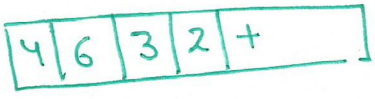
Step2:- While constructing tree right वाले को left में जोड़ो
left वाले को right

Q-> + * + 2 3 6 4

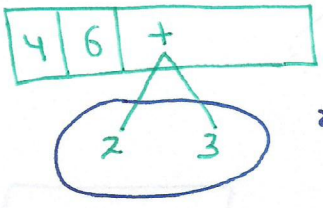
After reversing we get

4 6 3 2 + * +

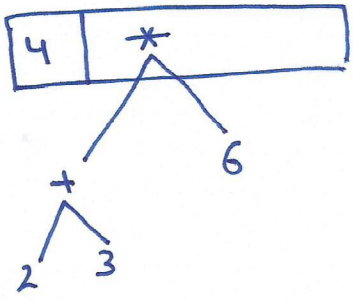


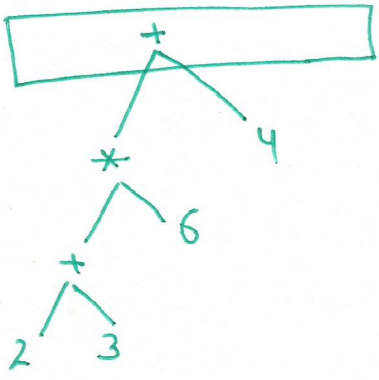


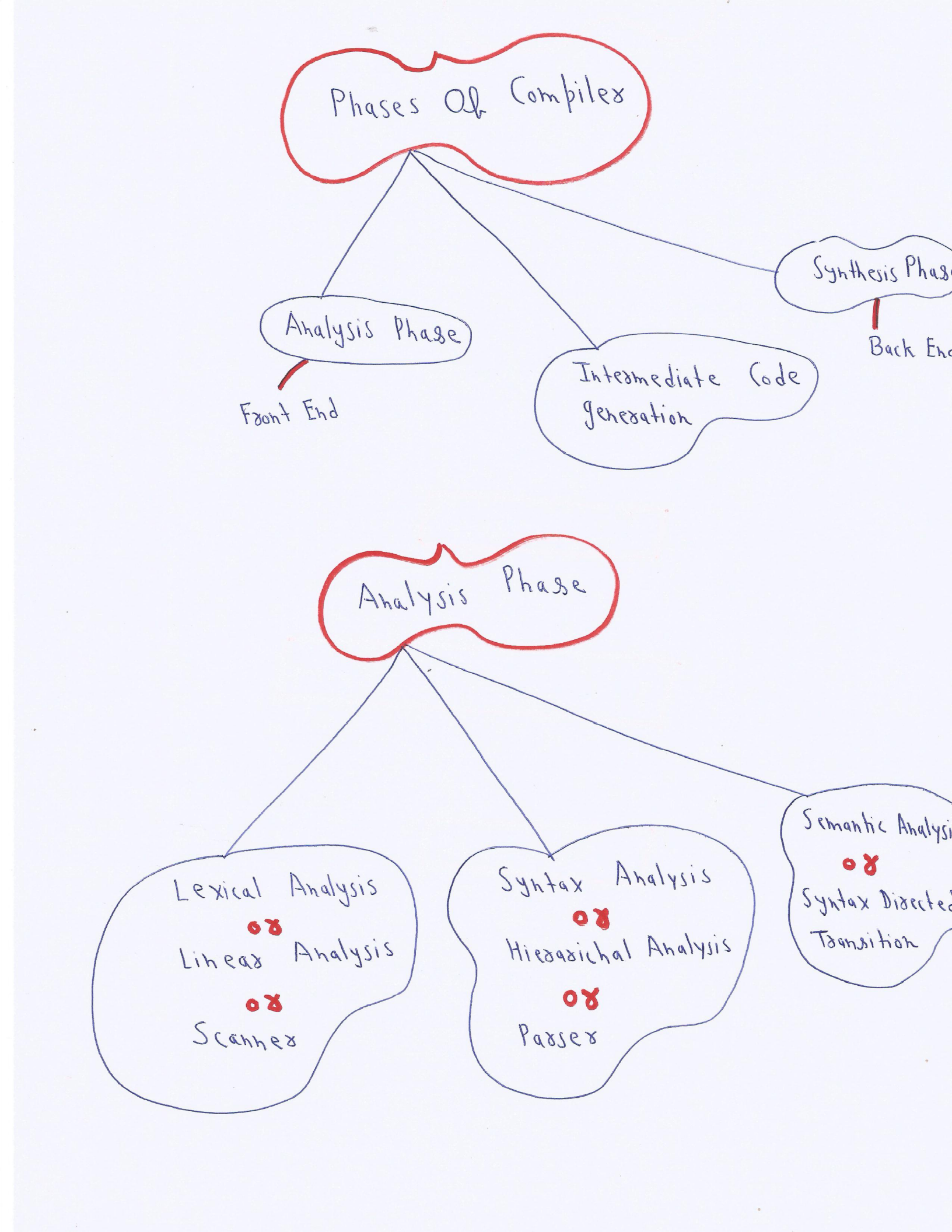
=

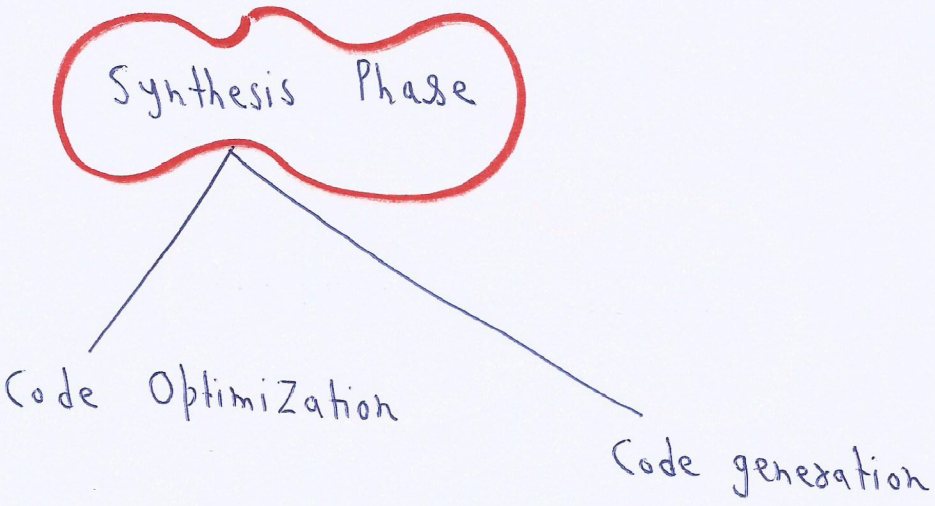


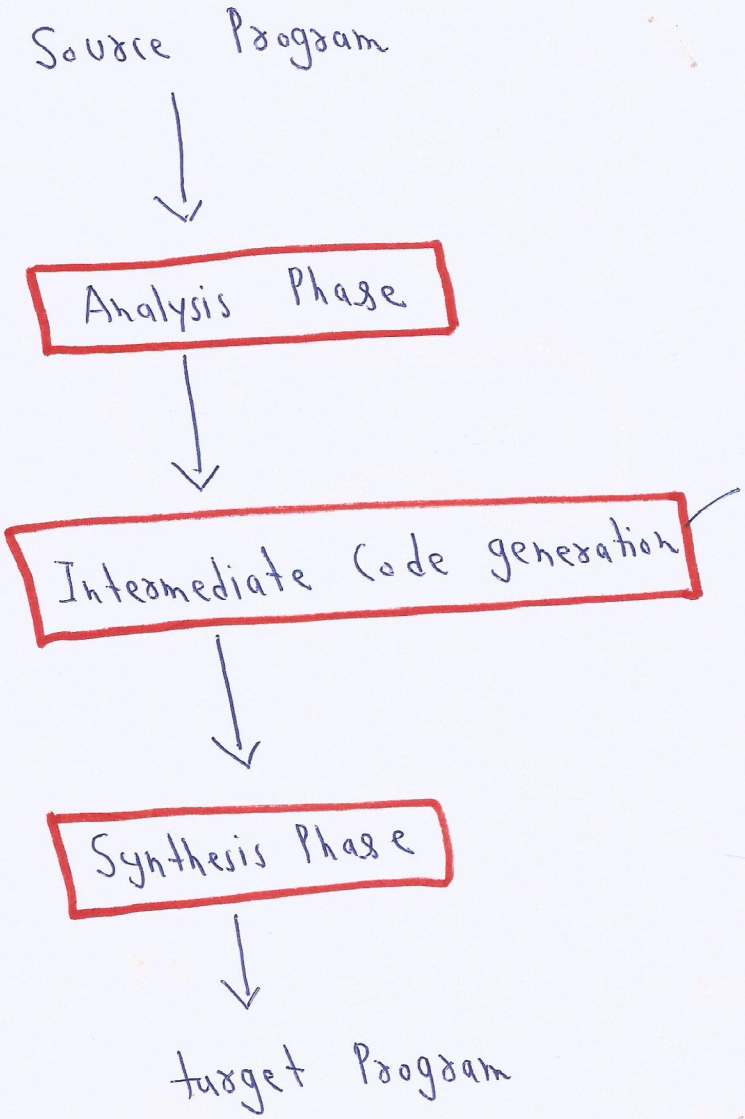
right वाला left
left वाला right



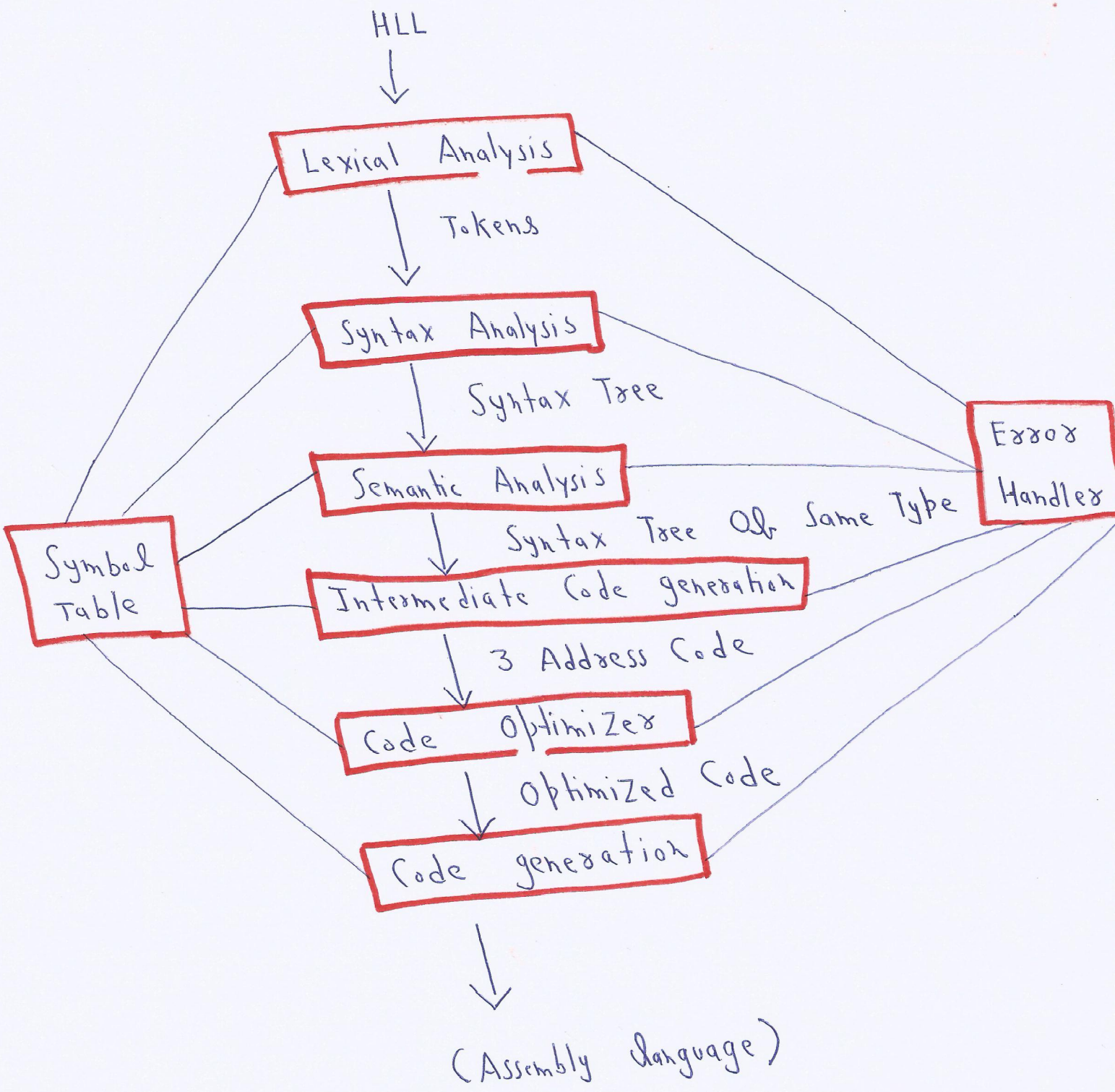



Phases Of Compiler
Analysis Phase
Front End
Intermediate Code Generation
Synthesis Phase
Back End
Analysis Phase
Lexical Analysis
or
Linear Analysis
or
Scanner
Syntax Analysis
or
Hierarchal Analysis
or
Parser
Semantic Analysis
or
Syntax Directed
Transition





Intermediate code is a m/c independent code which makes it possible to be devised for different processors.


HLL
Lexical Analysis
Tokens
Syntax Analysis
Syntax Tree
Semantic Analysis
Syntax Tree Of Same Type
Intermediate Code generation
3 Address Code
Code Optimizer
Optimized Code
Code generation
(Assembly language)
Symbol Table
Error Handler

Lexical Analysis

Source Program is scanned from left to right to find tokens.

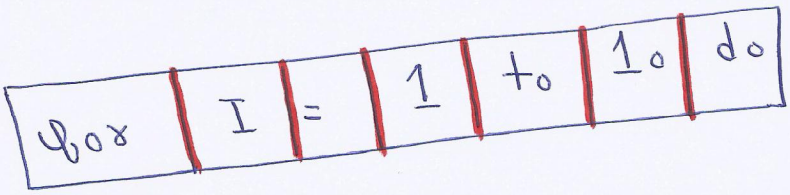
Token :- Sequence of characters having a collective meaning. Characters forming token is called lexeme.

Token :- Keywords (if, int), Identifier (Variable name, Constant name — pi, number — 1, 2, ..., function name), Operator, Character, String

Non Token :- Preprocessor directives (# include <stdio.h>), Space, Tab, Enter, Comments

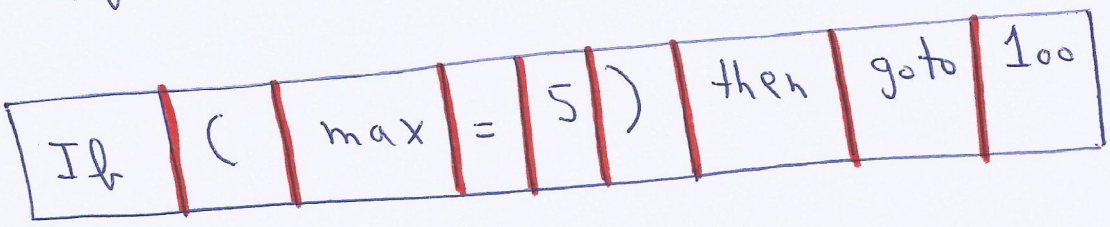
Q Identify Tokens

① for I = 1 to 10 do



7 tokens

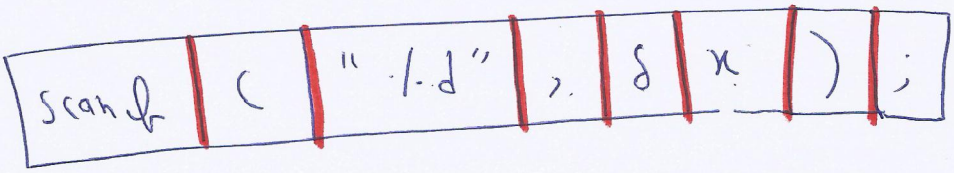
② If (max = 5) then goto 100



9 token

③ scanf ("%d", &x) ;

String



8 tokens

④

```
int MAX (int a, int b)
{
    if (a > b)
    return a;
    else
    return b;
}
```

24 tokens

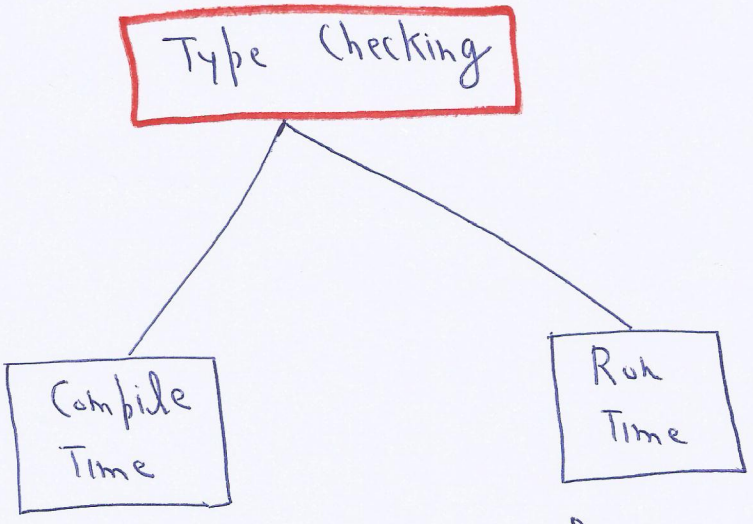
Syntax Analysis

In Syntax Analysis first of all a tree is formed which is known as parse Tree.

From parse tree syntax tree is formed which contains tokens only and whose internal nodes are keywords, operators.

Semantic Analysis

In Semantic Analysis type checking is done



Intermediate Code generation

3 Address Code.

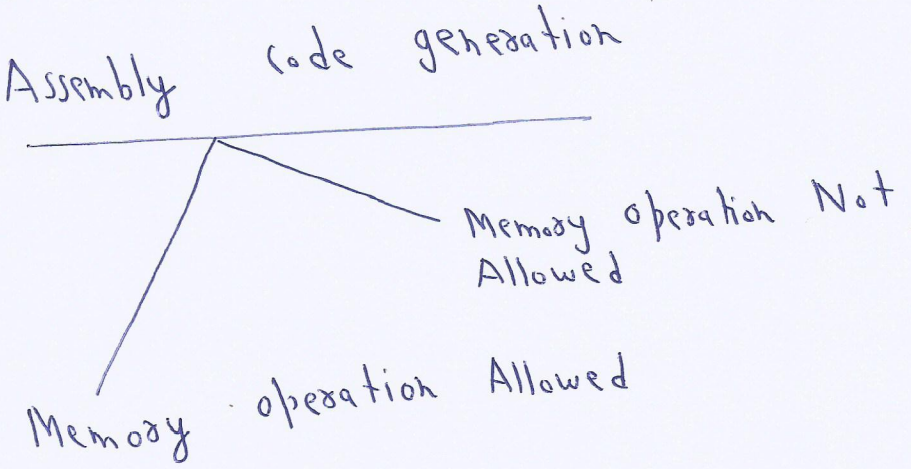
At most three operands

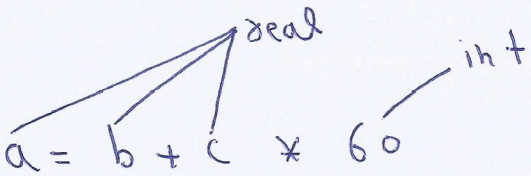
i.e 3 Address Code can have at most one operator other than assignment operator.

Code Optimization :- Removal of unnessary temproary variables keeping in mind

3 Address Code.

Code generation :-

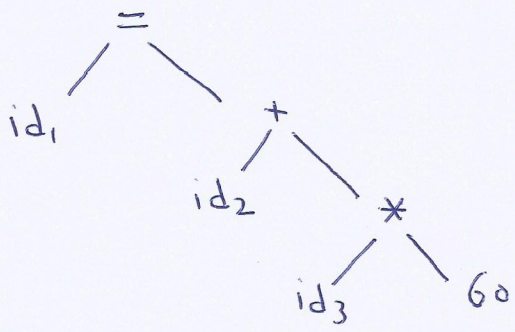




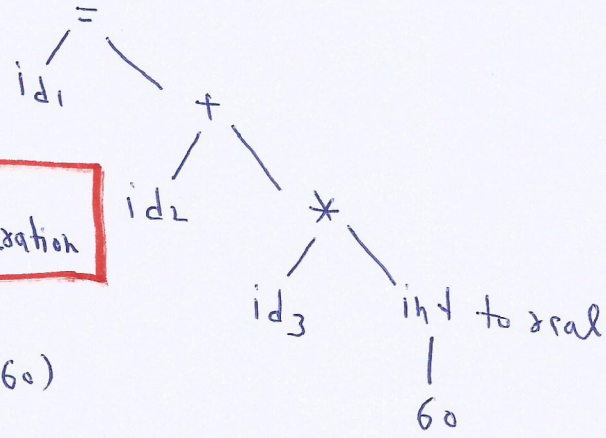
Lexical Analysis

$id_1 = id_2 + id_3 * 60$

Syntax Analysis



Semantic Analysis



Intermediate Code generation

temp 1 = int to real (60)

temp 2 = id_3 * temp 1

temp 3 = id_2 + temp 2

id_1 = temp 3

Code Optimization

Removal of temp1, temp3

```
temp2 = id3 * int to real (60)
id1 = id2 + temp2
```

Code generation

```
MOV   R1 , 60.0
MOV   R2 , id3
MUL   R1 , R2
MOV   temp2, R1
MOV   R1 , temp2
MOV   R2 , id2
ADD   R1 , R2
MOV   id1 , R1
```

Let Memory o/p not allowed

Minimum Number Of Registers Reqd to generate Assembly Code

- For given expression (Labelling Algorithm)
- For given code (Interference graph)

Labelling Algorithm

① First lable the leaves

left → 0
right → 1

if both the variables are not reqd to be in register for operation

i.e memory operation is allowed

left → 1
right → 1

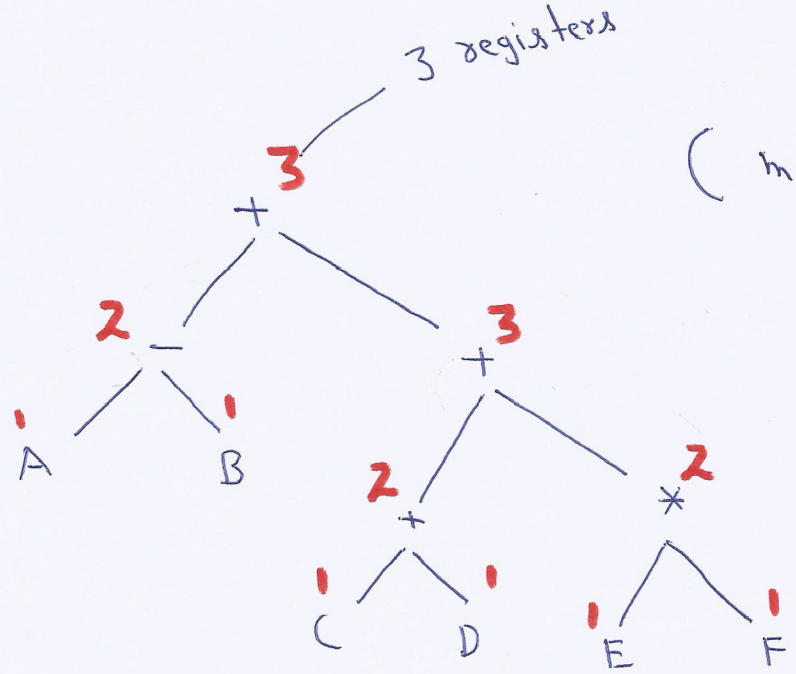
memory operation is not allowed

② After labeling leaves lable parent

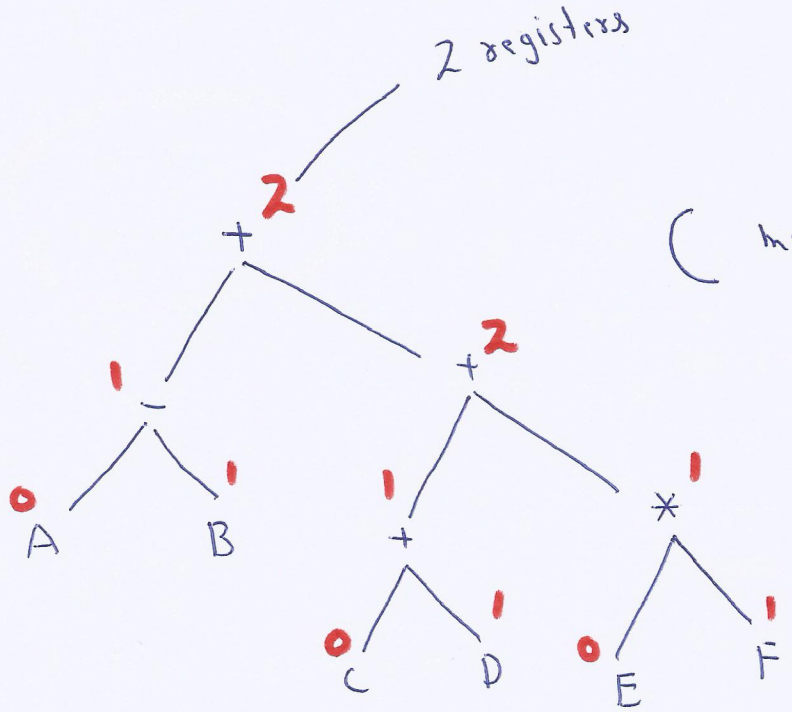
$$\text{lable (parent)} = \begin{bmatrix} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l + 1 & \text{if } l_1 = l_2 \end{bmatrix}$$

=

$$(A-B) + \Big(((C+D) + (E*F)) \Big)$$



(memory operation not allowed)



(memory operation allowed

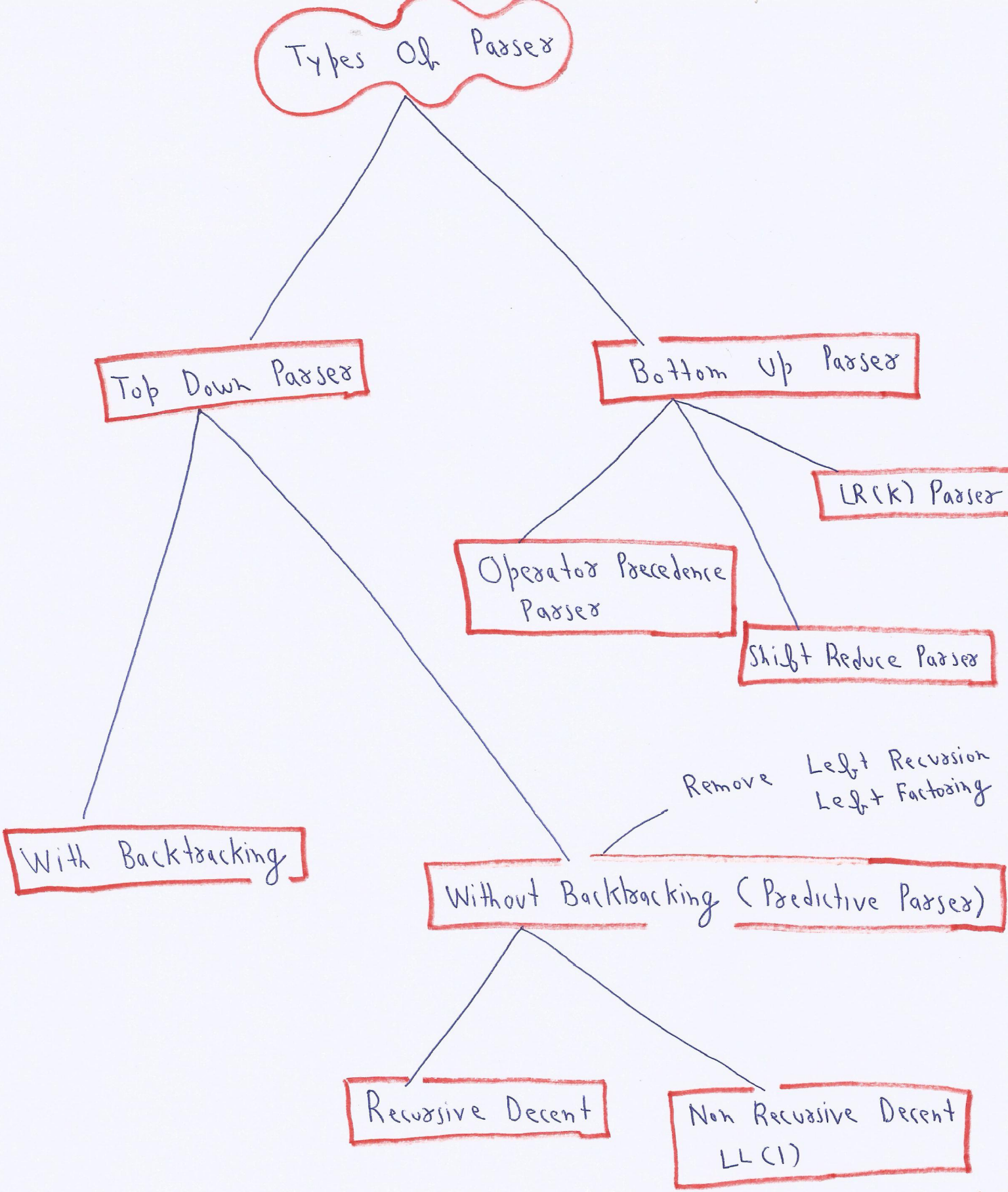
Parser

Parser takes string as input varifies wheather the string can be generated by the grammar given or not.

Context Free grammar

If yes than parse tree is generated.

If no than error is generated.


Types Of Parser
Top Down Parser
Bottom Up Parser
LR(K) Parser
Operator Precedence Parser
Shift Reduce Parser
With Backtracking
Remove Left Recursion
Left Factoring
Without Backtracking (Predictive Parser)
Recursive Decent
Non Recursive Decent
LL(1)