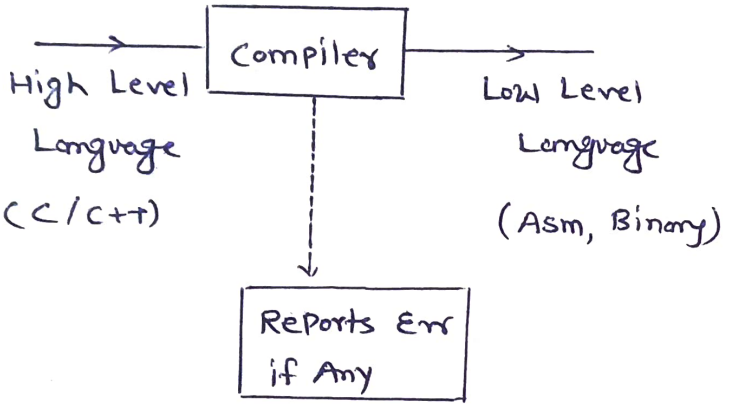
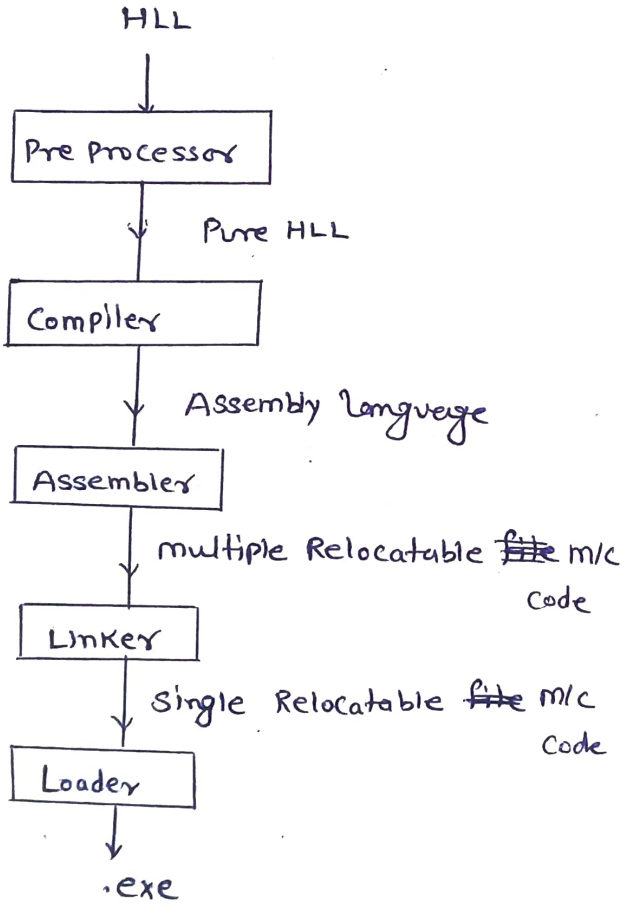
* Compiler Design *

→ A Compiler is a Software that translates Computer code Written in one Programming Language (source) into Another (target).



* Language Processing system -



- **Pre processor -**
  - → Removes All # include directives by including the files
    - ↳ Called file inclusion
  - → Remove all # define directives using macro expansion.

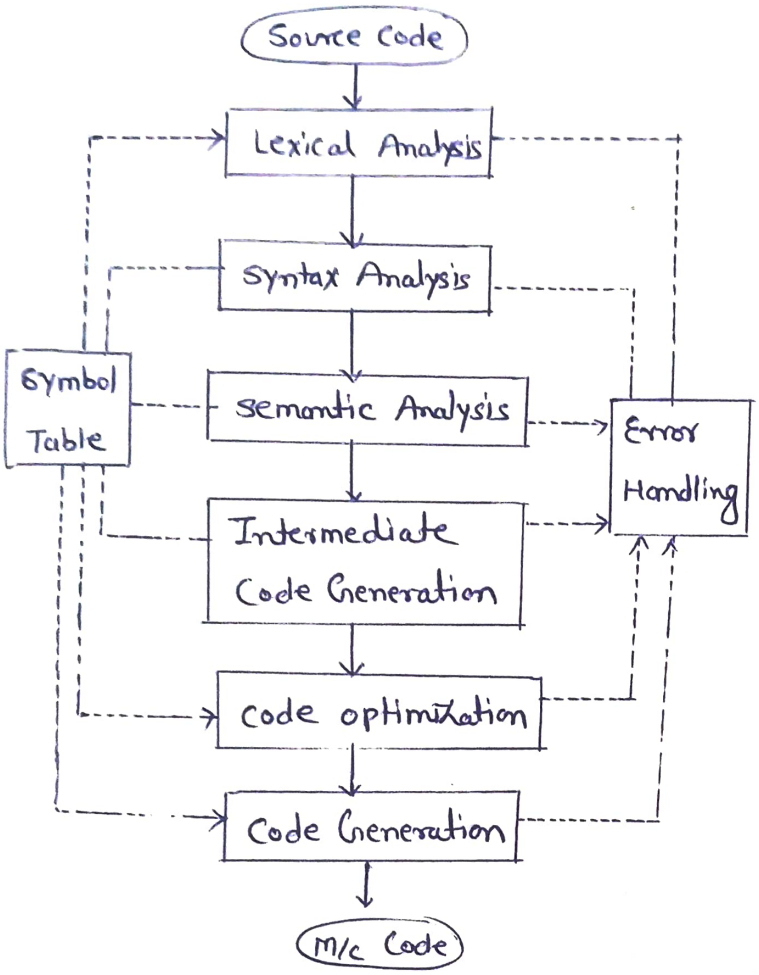
- **Compiler -**
  - → verifies All Limits, Ranges, Error etc
  - → slower than other system software
  - → occupies huge Amount of memory Space.

- **Assembler -**
  - → for Every platform (Hardware + OS) We have an assembler. They are not universal.
  - → The output of Assembler is called object file. It translates Assembly Language to machine code.

- **Linker :-**
  - ↳ Basic Work is to merge the object codes (that have not been Connected), Produced by Compiler, Assembler, Standard Library fn' and OS Resources.

- **Loader :-**
  - ↳ find / Calculate the exact address of memory Locations to Store Code.

* Phase of Compilers-



1) Lexical Analyzer:-

→ Reads the program and Convert it into a sequence of tokens.

⇓

A smallest unit of meaningful data

→ These Tokens Can be represent keywords, identifiers, Constants, Operators.

→ It removes White spaces and Comments.

```
float x,y,z;
x = y + z * 60
```

x - token : identifier
= : operator
y : identifier
+ : operator
z : identifier
* : operator
60 : Constant

2) syntax Analyzer - (Parser)

→ It checks Wheather the code adheres to the language rules.

→ We Construct Parse / Syntax Tree.

→ Uses productions of CFGs to Construct the Parse Tree.

→ It reads all the tokens one by one.

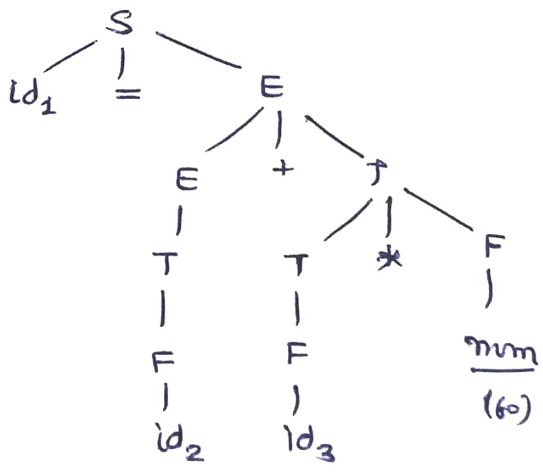
$S \rightarrow id = E$

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id \mid num$

$\Rightarrow id_1 = id_2 + id_3 * 60$



3) Semantic Analyzer -

→ It verifies the Parse tree Wheather it's meaningful or not.

→ It checks Wheather the program has any Semantic Errors, such as type mismatched and undeclared variables.

→ o/p : Annotated Parse tree

4) Intermediate code Generator -

→ Generate Intermediate code.

→ It is not specific to any particular m/c, making it Portable and easier to optimize.

⊕ : 1) Platform Independence

2) Simplifying optimization

- Dead code Elimination
- Loop optimization
- Common subexpression elimination
  ↳ Reusing Previously Calculated values to avoid redundant Calculations

$x = y + z * 60$

$t_1 = z * 60$

$t_2 = t_1 + y$

$x = t_2$

5) Code optimizer

→ This is a process of Improving the Intermediate or target Code, to make the program run faster, use less resources.

→ optimization is classified broadly into 2 types -

Machine Independent

Machine Dependent

6) Code Generation

→ final phase of Compiler, Where the intermediate representation of Source program (E.g.- Three Address Code) is translated into machine Code.

- Symbol Table -

↳ It is a data structure being used and maintained by the Compiler, Consists all the identifier's name along with their types.

→ Information stored in the symbol table about identifier -

name

type

Scope

size

offset

- Error handler

→ It is a subroutine to take care of the continuation of compilation, even any error at any phase occur.

→ Compiler can handle three types of Err -

1) Lexical Errors
2) Syntax Errors
3) Semantic Errors

- Passes :- A pass refers to the traversal of compiler through the Entire program.

1) Single Pass Compiler -

- Reads and processes the source code in a single pass, translating it directly into machine code.
- faster than multi pass Compiler
- memory efficient - use less memory
- Limited optimization

Eg - Tiny C Compiler (TCC)
Pascal

2) multi-Pass Compiler -

- It processes the source code multiple times before producing the final output.
- Advanced optimization
- Context Awareness
- memory usage ↑

- Gcc, LLVM

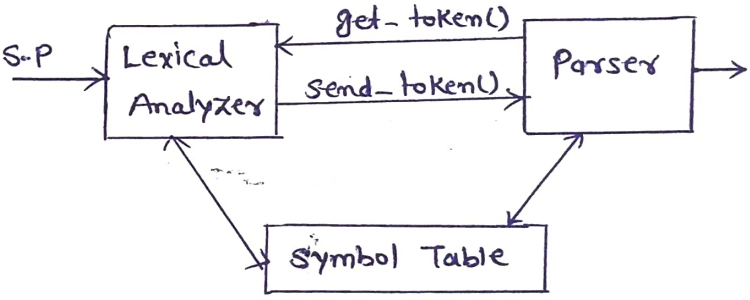
- Cross Compiler -

A Cross Compiler is a Compiler capable of creating executable code for a platform other than the one on which the Compiler is running.

↳ A Cross Compiler executes on machine X and produce machine code for machine Y.

* Lexical Analyzer -

→ Lexical Analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of tokens.



→ Actual Representation or stream of characters is called as Lexemes; and Logical meaning of that lexemes is known as Tokens.

Lexeme	Token
While	While
(	lParen
y	identifier
>=	Comparison
3	Integer
=	Assignment

→ Lexing Can be divided into 2 stages:

A) Scanning : Segments the input string into syntatic units called lexemes and categorizes these into token classes

B) Evaluating : Convert lexemes into processed values.

→ Tokens are normally identifiers, Keywords, operators, Constant and ~~specid~~ special symbols.

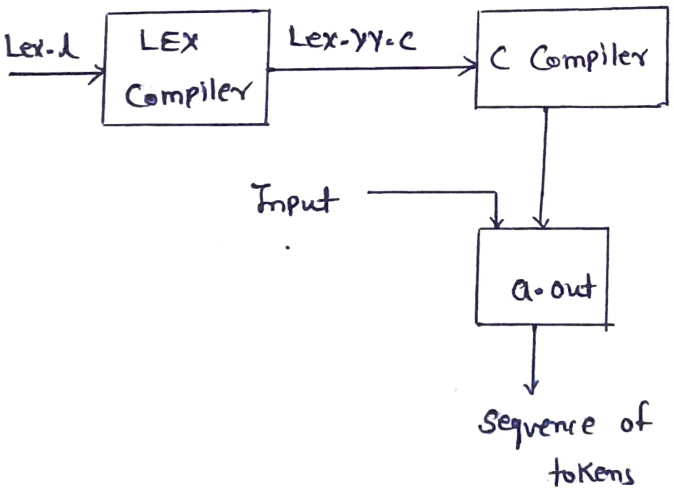
→ Lexical Analyzer uses a DFA to recognize these tokens, As DFAs are designed to identify Regular expression.

Lexical Analysis phase Can be Automated using a tool Called Lex.

- LEX is a Unix utility
- LEX Compiler -

  Takes a specification file (with extension .l) Containing Regular expressions and actions.

  Produce lex.yy.c file.



→ secondary function of Lexical Analyzer

1) Removal of Comments lines
2) Removal of White space
3) Give Error msg Along with line No. and Col No.

* Ambiguous Grammer

→ The Grammer CFG is said to be ambiguous if there are more than one derivation tree for any string i.e. if there exist more than one derivation tree (LMDT or RMDT).

$E \rightarrow E+E \mid E*E \mid id$

$V = \{E\}$

$T = \{+, *, id\}$

$Str = id + id * id$

LMD_1 -

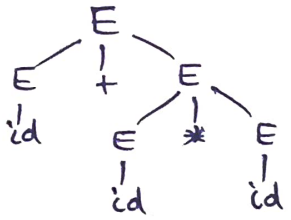
$E \rightarrow E+E$

$id+E$

$id+(E*E)$

$id+id*E$

$id+id*id$



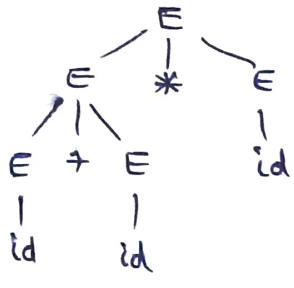
LMD_2 - $E \rightarrow E*E$

$E+E*E$

$id+E*E$

$id+id*E$

$id+id*id$



• Ambiguous to unAmbiguous Grammer -

$E \rightarrow E+E \mid E*E \mid id$

$w = id + id * id$

→ Removal of Ambiguity -

1) Precedence -

if different operators are used, we will consider the precedence of the operators.

- The level at which the production is present denotes the priority of operators.
- high levels ≡ Less priority
  ↳ Close to Roots

  Lower levels ≡ High priority

2) Associativity -

If the same precedence operators are in production, then we will have to consider the Associativity.

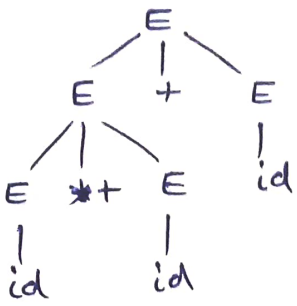
(+, -) → Left to Right Associative

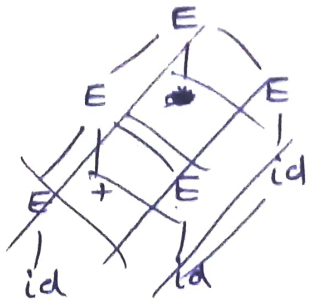
(*, /) → Left to Right Associative
⇓
Left Recursion

(^) → Right to Left Associative

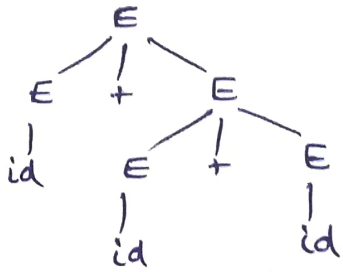
$E \rightarrow E+E \mid E*E \mid id$

$w = id + id * id$





(I)



(II)

According to Associativity, for

$w = (id + id) + id$

tree I is Correct.

↳ We have to break right recursion.

$E \rightarrow E+id \mid id$ (LR)

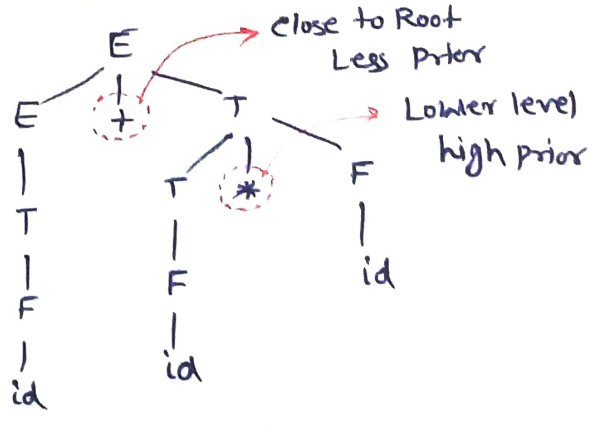
⇒ $E \rightarrow E+E \mid E*E \mid id$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

$w = id + id * id$



Left Recursion : $A \rightarrow A\alpha \mid \beta$

Right Recursion : $A \rightarrow \alpha A \mid \beta$

- Elimination of left Recursion-

$$A \rightarrow A\alpha \mid \beta \equiv \beta\alpha^*$$

$$\Downarrow$$

$$\left.\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \in / \alpha A' \end{array}\right] \equiv \beta\alpha^*$$

1) $E \rightarrow E+T \mid T$ (α = $+T$, β = T)

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow \in / +TE' \end{array}$$

2) ~~$S \rightarrow Sa \mid Sos \mid 01$~~

$S \rightarrow S01sos \mid 01$ (α = $01sos$, β = 01)

$$\begin{array}{l} S \rightarrow 01S' \\ S' \rightarrow \in / 01sossS' \end{array}$$

$A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$

$$\Downarrow$$

$$\begin{array}{l} A \rightarrow \beta_1 A' / \beta_2 A' / \ldots\ldots / \beta_n A' \\ A' \rightarrow \in / \alpha A' \end{array}$$

$A \rightarrow A\alpha_1 / A\alpha_2 / \ldots\ldots / A\alpha_n / \beta$

$$\Downarrow$$

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha_1 A' / \alpha_2 A' / \alpha_3 A' \ldots\ldots / \alpha_n A' / \in \end{array}$$

$A \rightarrow A\alpha_1 / A\alpha_2 / \ldots\ldots / A\alpha_n / \beta_1 / \beta_2 \ldots\ldots / \beta_m$

$$\Downarrow$$

$$\begin{array}{l} A \rightarrow \beta_1 A' / \beta_2 A' / \ldots\ldots / \beta_n A' \\ A' \rightarrow \alpha_1 A' / \alpha_2 A' / \ldots\ldots / \alpha_n A' / \in \end{array}$$

- Left factoring :-

Process of Conversion of Non deterministic grammer into deterministic grammer is known as left factoring

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \alpha\beta_4$

α is a Common prefix

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 / \beta_2 / \beta_3 / \beta_4 \end{array}$$

1) $S \rightarrow iEts / iEtses / a$

$E \rightarrow b$

$$\Downarrow$$

$$\begin{array}{l} S \rightarrow iEts\,s' / a \\ s' \rightarrow \in / es \\ E \rightarrow b \end{array}$$

2) $S \rightarrow assbs / asasb / abb / b$

$S \rightarrow as' / b$

$S' \rightarrow ssbs / sasb / bb$

$$\begin{array}{l} s' \rightarrow ss'' / bb \\ s'' \rightarrow sbs / asb \end{array}$$

1) $S \rightarrow as / b$

$s' \rightarrow ss'' / bb$

$s'' \rightarrow sbs / asb$