~~Lexical Phase Errors~~

- Classification of errors

- errors
  - Compile Time
    - Lexical Phase Errors
    - Syntactic Phase error
    - Semantic Errors
  - RunTime Errors

- Lexical Phase Errors
  1. spelling errors . incorrect tokens
  2. exceeding length of identifier or numeric constant
  3. Appearence of illegal characters

- Example

```
switch (choice)
{ ....
}
```

Swtich is ~~misp~~. misspelled hence identified as an Identifier, where it is actually a keyword.

- Cascading of characters.

```
switch (choice)
{   case1 : get_data();
            break;
}
```

here case1 will be detected as valid identifier as there is no space between case & 1

- Fortran go →

DO 5 I = 1.25

Whether it is DO5I or any command?

Classification of errors

error...

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

2 conditions

(1) 2 Non terminals should not be there

(2) It can be applied for some of the ambiguous grammar

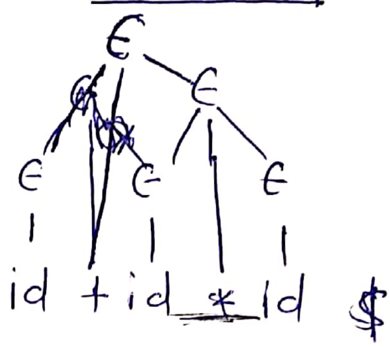
check in this dir^n.

	id	+	*	\$
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	accept

Note → Precedence should be from row to column.

< push
>. pop

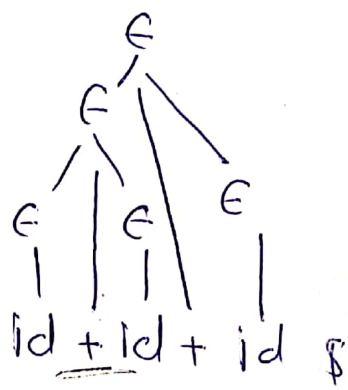
id + id * id \$



\$ id + id * id

try for id + id + id \$

\$ id + id . + id.



example

→ id + id * id \$

→ id + id + id \$

~~Phase errors~~

- Classification of errors

Symbol Table structure :-

Attributes :

1. Variable Name :- Variable stored in symbol Table by its nam
2. constant : -
3. Data types : - Data type & associated variable is stored in Symbol Table.
4. compiler generated temperories
5. Function names .
6. Scope information

Data structure Used :-

1. **Linear List :** - Simple
   - New names added in the order as they arrive
   - Pointer available is maintained at the end of all stored records

Name 1	Info 1
name 2	Info 2
name 3	Info 3

empty
start of empty slot.

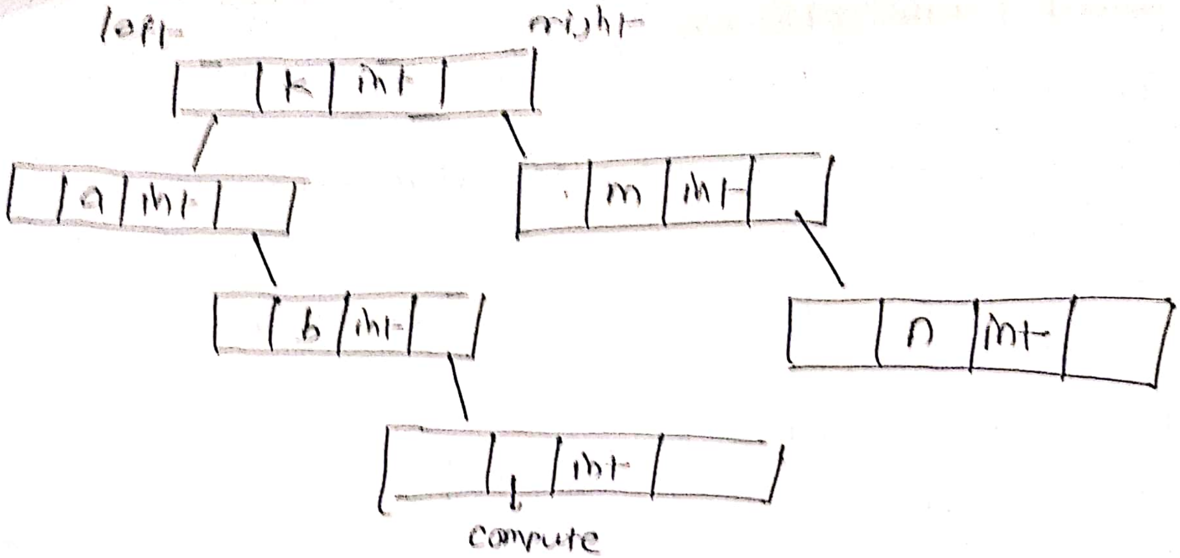
2. **Binary Trees :** -

Node structure

left child	Symbols	Information	Right child

↳ left symbol stores addr of previous symbol & Right child stores addrs of next symbol.

6 7 ... applied for

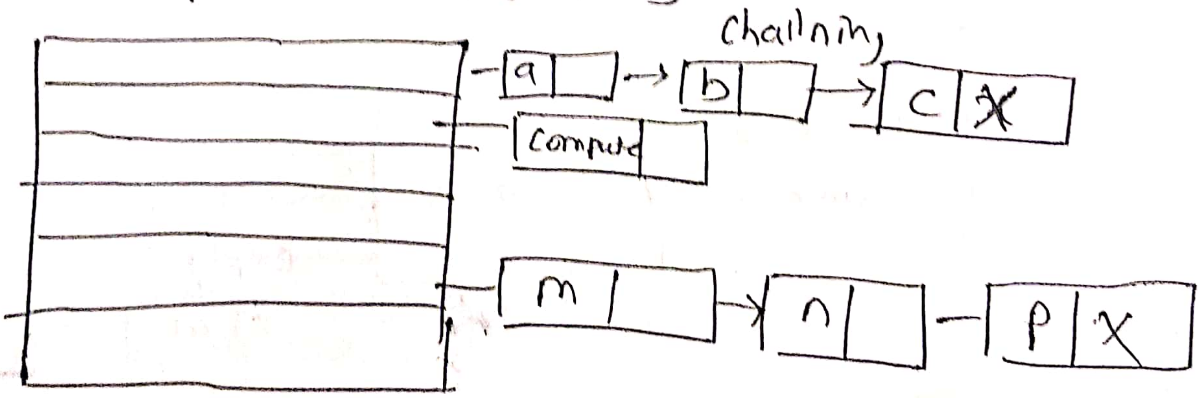


③ **Hash Table :-**

Consists of 'k' entries from 0, 1 to k-1
these entries are pointers to ST.

For position in hash table we use hash function 'h'
h(name) will result any integer betn 0 to k-1

position = h(name)

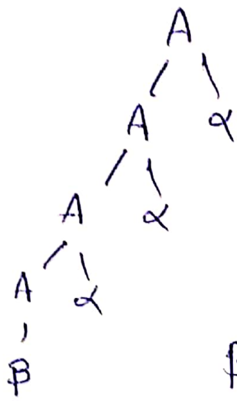


Left Recursion :-

A grammer is a left recursive if it has a nonterminal A such that there is a derivation for some string α. [A itself present in RHS]

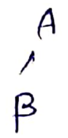
$$A \to A\alpha \mid \beta$$

~~$A \to A\alpha$~~



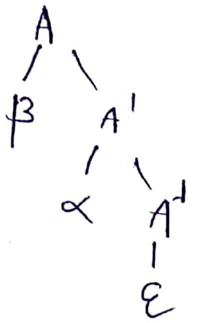
This is known as left recursion

2nd tree can be



$\beta\alpha^*$ $\Rightarrow$ $A \Rightarrow \beta A'$

$A' \Rightarrow \alpha A' \mid \epsilon$



example:

① $S \to Sb \mid c$

$S \to cS'$

$S' \to bS' \mid \epsilon$

② $E \to E+T$

$T \to id$

$E \to TE'$

$E' \to +TE' \mid \epsilon$

Left factoring :- A grammer may not be suitable for Recursive descent parsing even if there is no left ~~factoring~~ recursion.

Factoring out the common prefix is known as left factoring

Let $A \to \alpha\beta_1 \mid \alpha\beta_2$

then $A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

example :-

$S \to iEtS \mid \underline{iEtSeS} \mid a$

$E \to b$

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

Ambiguous grammer

$E \rightarrow E+E$
$\quad / E*E$
$\quad / id$

Terminals : +, *, id
Non Terminals = E

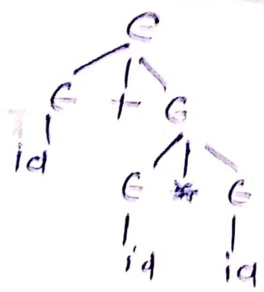
id + id * id →

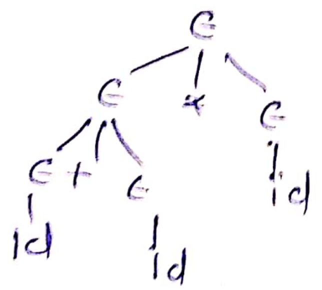
$E \Rightarrow E+E$

→ Leftmost derivation ← Rightmost Derivation

Leftmost	Rightmost
$E \Rightarrow E+E$	$E \Rightarrow E+E$
$\Rightarrow id + E*E$	$E \Rightarrow E+E*E$
$\Rightarrow id + id * E$	$\Rightarrow E+E*id$
$\Rightarrow id + id * id$	$\Rightarrow E+id*id$
	$\Rightarrow id+id*id$

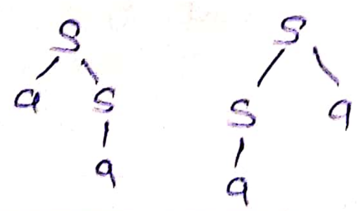
LMD	RMD
$E \Rightarrow E*E$	yes.
$\Rightarrow E+E*E$	$E \Rightarrow E*E$
$\Rightarrow id+E*E$	$\Rightarrow E*id$
$\Rightarrow id+id*E$	$\Rightarrow E+E*id$
$\Rightarrow id+id*id$	$\Rightarrow E+id*id$
	$\Rightarrow id+id*id$





grammer is ambiguous.

$S \rightarrow aS / Sa / a \qquad w = aa$



— ambiguous grammer

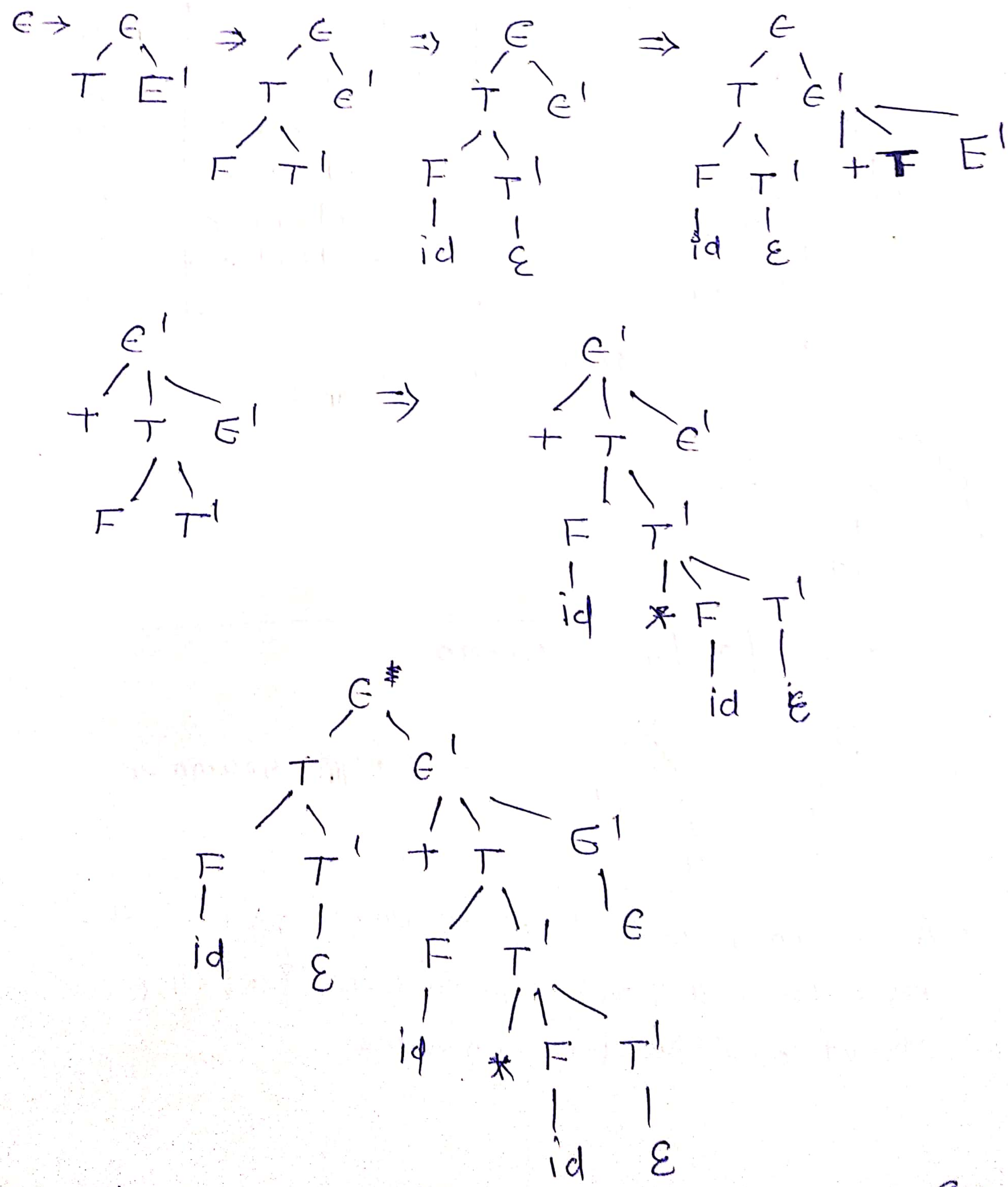
A grammer is said to be ambiguous when there are more than 2 possible parse trees ~~for~~. These ambiguouity should be reduced or eliminated

Top Down Parsing

Constructing a parse tree for the input string starting from root & creating nodes of parse tree in preorder.

id + id * id

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$



$S \rightarrow cAd$

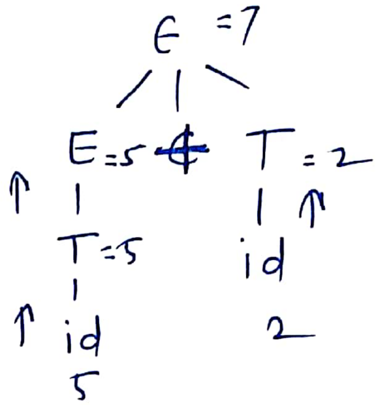
Synthesized attributes -

A synthesized attribute of the non terminal on the left hand side of a production.

Attributes can take the value ~~of~~ only from its children.

example

(1) $E \Rightarrow E+T$
$\;\;\;\;\;\;\;\;\;\;\; / \; T$
$T \Rightarrow id$



* (2) $A \to BC$ → A is synthesized attribute

i/p 5+2

Inherited attributes :-

An attribute of a non terminal on a right hand side of a production is called inherited attribute. The attribute can take value from its parent or from its siblings

* $A \to BC$: B is dependant on ~~value~~ ~~of~~ A ~~&~~ as well as C then it will be inherited attribute

S-attributed grammer (SDT)

- If SDT uses only synthesized attributes it is called as S-attributed SDT.
- S attributed SDT uses evaluation in bottom up parsing, as the values of parent node is dependant on values of child nodes.

example - same as synthesized attributes

L-attributed SDT -

If SDT uses Both synthesized & inherited attributes with a restriction that inherited attributes can inherit values from left sibling only, it is called L attributed SDT.

example

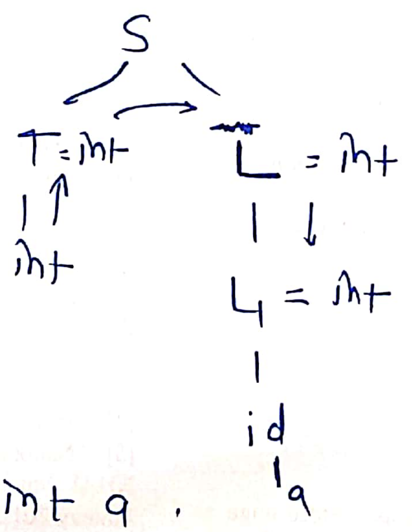
$S \to TL$ { $L.type = T.type$ }

$T \to int$ { $T.type = int$ }

$T \to char$ { $T.type = char$ }

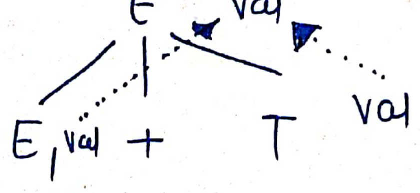
$L \to L_1$ { $L_1.type = L.type$ }

$L_1 \to id$ { $L_1.type = id.lval$ }



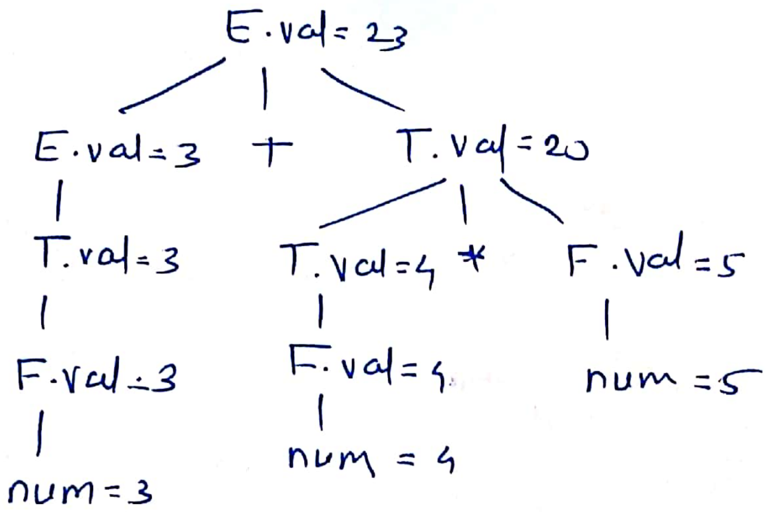
i/p = int a ;

dependency graphs → it depicts the flow of information among the attribute instances in a particular parse Tree.



Annotated or decorated parse Tree

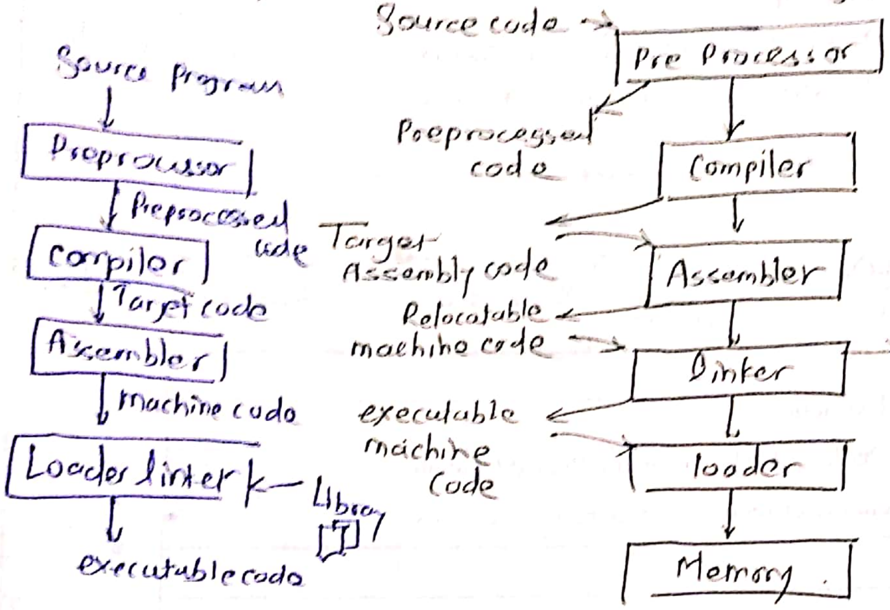
*5



A parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree

* Language Processing System

- We need to write a program in high level language. These programs are then fed into a series of tools & os components to get desired code that can be used by machine. This is known as language Processing System.



1. User writes program (C program) (high level)
2. C compiler compiles the program & translates it to assembly program (low level lang)
3. An assembler then translates the assembly program into machine code (object)
4. linker tool is used to link all parts of program (executable machine code)
5. loader loads all of them into memory & then program gets executed,

① Preprocessor - It is part of compiler. It is tool that produces i/p for compilers. It deals with macro processing, augmentation, file inclusion.

② Interpreter -

It translates high level lang. to low level lang.

The difference is :- The way they read the source code or input.

* Compiler Reads Whole source code at once, creates tokens checks symantics, generates intermediate code, executes the whole program.
* Interpreter Reads a statement from i/p, converts it into intermediate code, executes it, then takes next statement
* When error occurs -
  - Interpreter stops execution & reports it
  - Compiler reads program ~~#~~ even if encounters some error.

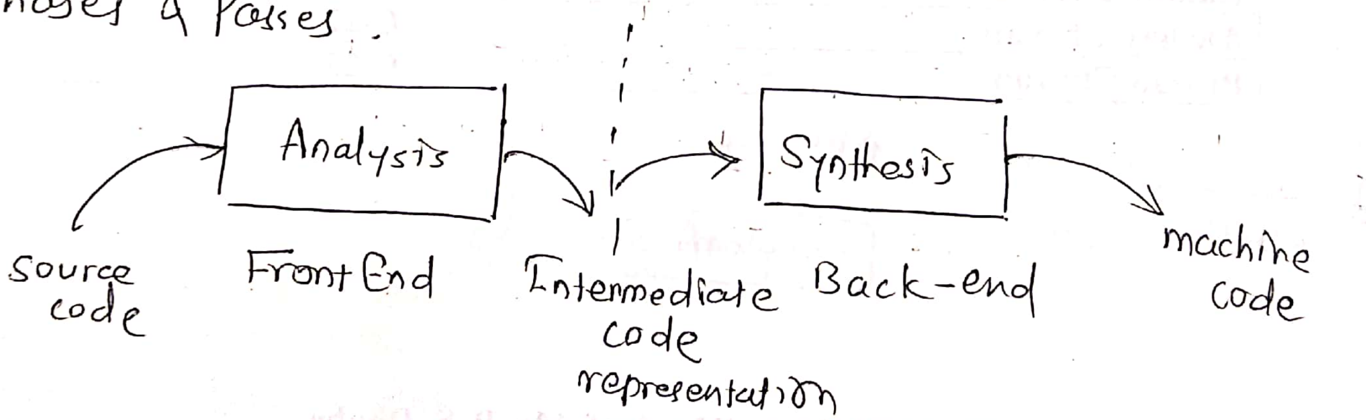
(3) Assembler.

Assembler Translates ~~as~~ assembly language into machine code

(4) linker - Computer Program - merges various object files.

(5) loader - loading into the memory

* Phases & Passes.



Analysis phase - It reads program, devides into core part & then checks for lexical, grammer & syntax errors O/p of this is intermediate code

Synthesis Phase - generates the target Program with the help of intermediate code & symbol Table

* ① Pass → Pass is traversal of compiler through entire program

② Phase – Phases are distinguishable stages which takes i/p from previous stage, processes & o/p can be given to next phase

A pass can have more than one phase

* Compilation process of source code through phases :–

example · Position = initial + rate * 60

↓

$\boxed{\text{lexical Analyzer}}$

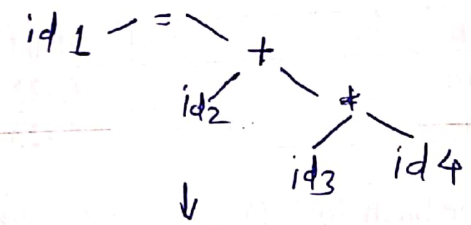
↓

Tokens <id1> <=> <id2> <+> <id3> <*> <id4>

↓

$\boxed{\text{Syntax Analyzer}}$

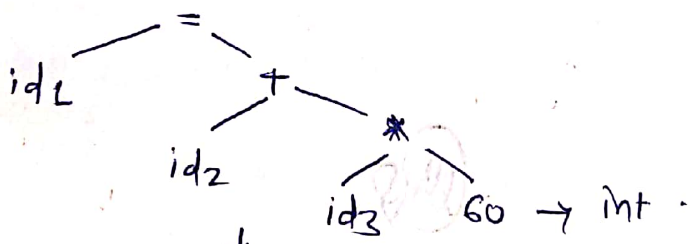
↓



Symbol Table	
position	
initial	
rate	

↓

$\boxed{\text{Semantic Analyzer}}$



↓

$\boxed{\text{intermediate code generator}}$

↓

temp 1 = int · to real (60) (float)
temp2 = id3 * temp 1
temp 3 = id2 + temp2
id1 = temp 3.

↓

$\boxed{\text{Code optimizer}}$

temp 1 = id3 * 60 .
id1 = id2 + temp1

↓

$\boxed{\text{Code generator}}$

↓

MOVF id3, r2
MULF 60, r2
MOVF id2, r2
ADDF r2, r1
MOVF r1, id1

<id,1> <=> <id,2> <+> <id,3> <*> <60>

(1) lexeme = (token name , attribute Value)

The position is lexeme that would be mapped in token (id,1)

identifiers — Table entry in symbol Table

(2) Symbol - '=' will be mapped into token <=> it is having no attribute value.

(3) initial is lexeme that is mapped into token <id,2>

(4) + is lexeme that mapped into token <+>

(5) rate is lexemes = <id, 3>

(6) * will be mapped as <*>

(7) 60 is mapped as <60>

* Symbol Table

```
int sum ( )
{
    double sum = 0;
    int
    for (int i = 0 ; i<n; i++)
    sum = a+b;
    return sum;
}
```

Symbol Name	Type	Scope
a, b	int	
sum	int	local, global or extern or function scope
i	int	
	int	