Phases of Compilers:-

Compilation process is partitioned into some subprocesses called phases.

Inorder to translate a high level code to machine code, it is needed to go phase to phase, with each phase doing a particular task and parsing out its output for the next phase.

Lexical Analysis or Scanning:-

- ✓ It is the first phase of the compiler.
- ✓ The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

Example:-

Consider the statement if(a<b)

In this sentence the tokens are,

if, (, a, <, b,)

Number of Tokens: 6

Identifiers : a, b

Keywords : if

operators : (, <,)

Syntax analyzer or Parser:-

- ✓ Tokens are grouped hierarchically into nested collections with collective meaning.
- ✓ A Context Free Grammar (CFG) specifies the rules or productions for specifying identifying constructs that are valid in a programming language.

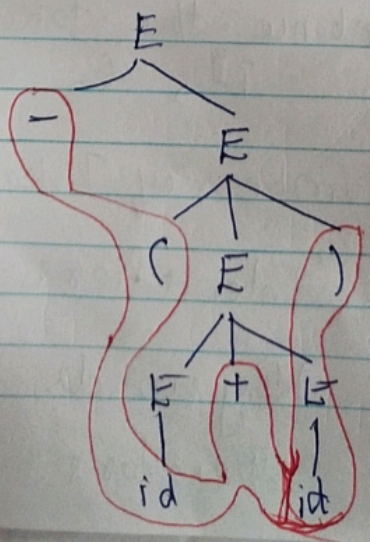
The output is a parse/syntax/Derivation tree.

Example:- Parse tree for $-(id + id)$ using the following grammar.

$E \rightarrow E+E$
$E \rightarrow E * E$
$E \rightarrow -E$
$E \rightarrow (E)$
$E \rightarrow id$

} productions or rules.

Parse Tree



$-(id + id)$

Semantic Analysis :-

- It checks the Source program for Semantic errors.
- Type checking is done in this phase.

  Where the compiler checks that each operator has matching operands for Semantic Consistency with the language definition.
- Gather the type information for the next phases.

Example 1 :-

The bicycle rides the boy.

This Statement has no meaning. but it is Syntactically correct.

Example 2 :-

```
int a;
bool b;
char c;
c = a+b;
```

We can't add integer with a boolean variable and assign it a character variable.

Intermediate Code Generation:-

It has two important properties.

(1) It should be easy to produce

(2) Easy to translate into the target progeam.

"Three address code" is one of the common forms of intermediate code.

"Three address code" Consists of a sequence of instructions, each of which has at most three opeands.

Example:-

$id_1 := id_2 + id_3 * 10$;
$t_1 := \text{inttoreal}(10)$;
$t_2 := id_3 * t_1$;
$t_3 := id_2 + t_2$
$id_1 := t_3$

Code optimization:-

The output of this part will resultin faster running machine code.

Example:- The optimized code

$t_1 := id_3 \times 10.0$
$id_1 := id_2 + t_1$

{ t_2, t_3 regesters are eliminated.

Code generation :-

→ Target code is generated in this phase.

→ Generally the target code can be either a relocatable machine code or an assembly code.

→ Intermediate instructions are each translated into a sequence of machine instructions.

→ Assignment of register will also be done.

Example: Assembly code.

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Symbol Table Management :-

Symbol Table is a data structure containing a record for each variable name, with fields for the attributes of the name.

Uses of Symbol Table :-

1. Record the identifiers, which are used in the source program.

2. Records the type and scope.

3. If it is the procedure name - then the number of arguments, type of arguments, the method of passing [parse by reference] and the type returned.

Error Detection and Reporting:-

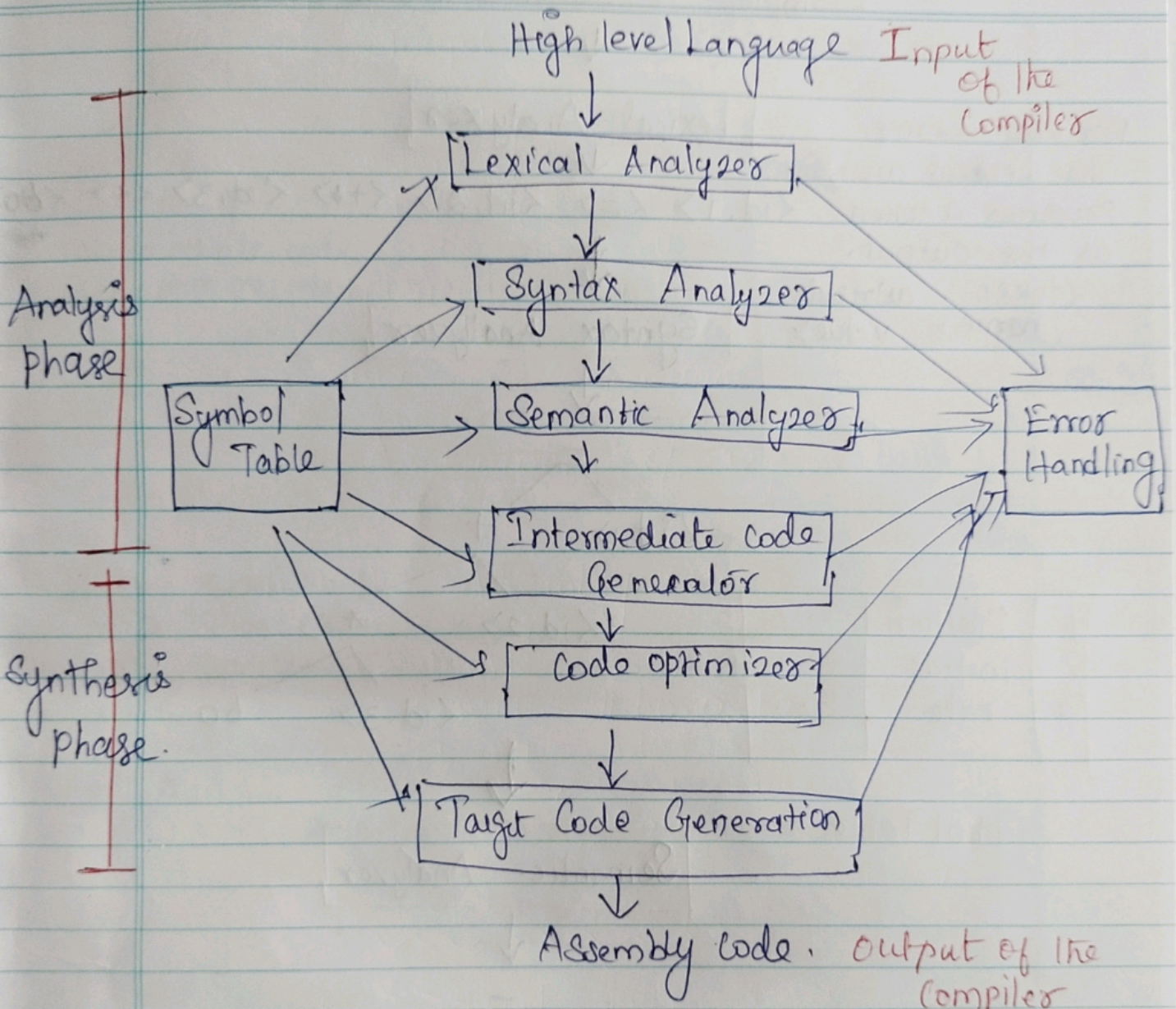
(i) Lexical phase can detect errors where the characters remaining in the input "do not form any token".

(ii) Errors of the type, "violation of syntax" of the language are detected by Syntax analysis.

(iii) Semantic phases tries to detect constructs that have the right syntactic structure but no meaning.

Example: adding two array names.

Phases of Compiler



Example:- Position = initial + rate * 60

↓

Lexical Analyzer

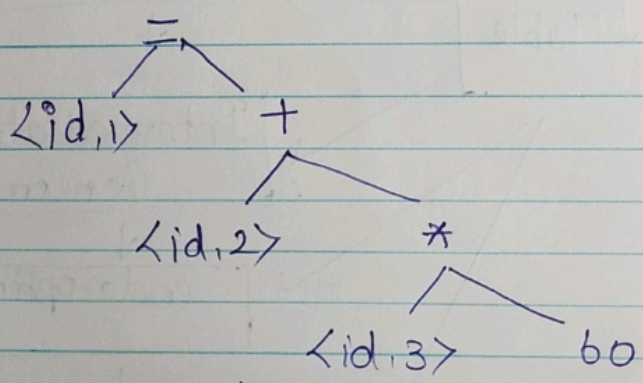
↓

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

↓

Syntax Analyzer

↓



For each Lexme, the Lexical analyzer produces a token as the output.

<token name, attribute value>

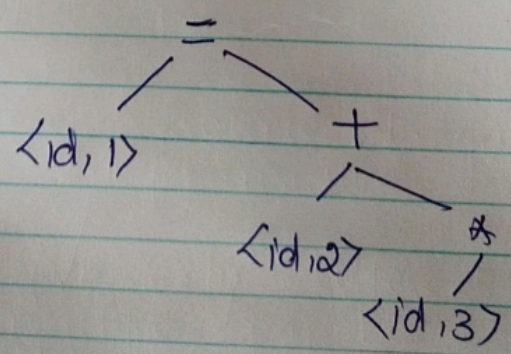
1	Position	...
2	initial	...
3	rate	...

Symbol Table

↓

Semantic Analyzer

↓



Three address code

Intermediate Code generator

Syntax tree is a form of intermediate code Generator representation

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

The language specification may permit some type conversion called coercions.
→ Binary arithmetic between integer & floating point number, the compiler may convert or coerce the integer into a floating point number.

Code optimizer

machine Independent.

Objective of code optimizer
1. Faster
2. Shorter
3. Less power.

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

Assignment of Registers.

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```