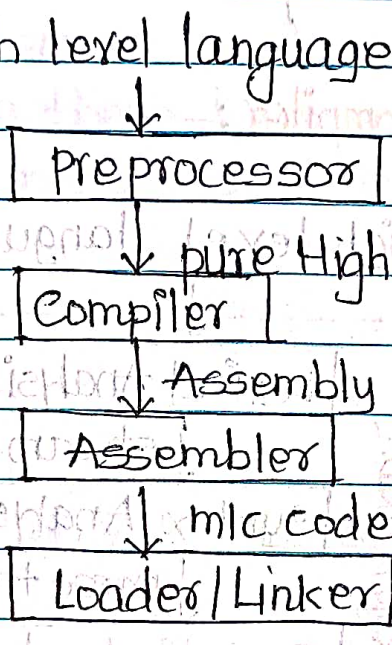
Chapter - 06

Compilers

Introduction to Compilers :- Compiler is a system softwa
-re that convert High level language into Low level language.



Preprocessor :-

It takes input as a high level program that includes preprocessor directives (#include, # define) OR Macros. Preprocessor process the high level language program in such a way that in output it gives high level language program that do not include preprocessor directives or Macros.

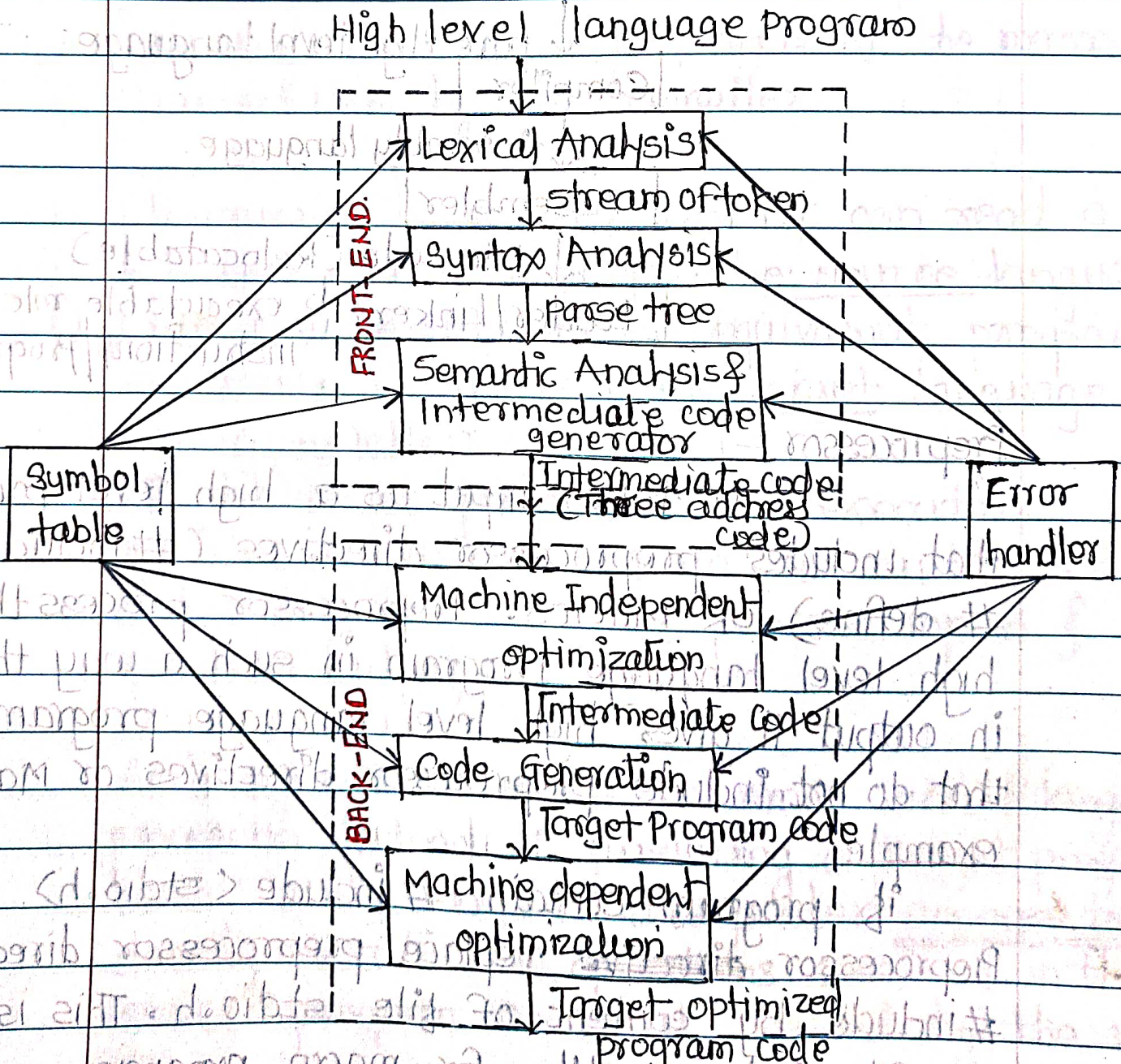
example

if program contain #include <stdio.h>. Preprocessor ~~directive~~ replace preprocessor directive #include by content of file stdio.h. This is know as file inclusion. Ily for macro expansion.

Preprocessor directives direct the preprocessor What to do ?

Example of Compiler:- GCC , Turbo C compiler

Phases of Compiler:-



1] Lexical Analysis :- (Scanner)
- Lexical Analysis works as a text scanner.
- This phase scans the source code as a stream of characters & converts it into meaningful lexemes.
- This phase remove white spaces & comment lines.
- It represents these lexemes in the form of tokens as <token_name, attribute_value>.
- Using regular expression the lexical analyzer identify the tokens.

2] Syntax Analysis :- (Parsing)
- It takes the tokens produced by lexical analysis as input & generate a parse tree or syntax tree.
- Tokens(arrangements) are checked against the source code grammer or CFG i.e the parser checks
- if the expression made by the tokens is syntactically correct.

- Syntax error is detected by syntax analyzer if input is not given as per the syntax grammer.

3] Semantic Analysis :- (type checking)
- It checks whether the parse tree constructed follows the rules of language.

For example: ① assignment of values is between compatible data types
② adding string to an integer.

- Also, It keeps track of identifiers, their types & expressions: whether identifiers are declared

before use or not.

- Parse tree is semantically verified it means whether parse tree is meaningful or not.

4] Intermediate Code Generation :- (Three address code)
- After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
- It represents a program for some abstract machine.
- It is in between the high-level language & the machine language.
- This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

5] Code Optimization :-
- Optimization can be assumed as something that removes unnecessary code lines, & arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

6] Target Code Generation :-
- It generate target code which can understood by assembler.

Practically, we don't have all these phases, most of the phases or job are done by syntax analyzer itself. syntax analysis phase is heart of compiler.

7] Symbol table :- is a data structure.
- All the identifier's names along with their types are stored here.
- The symbol table make it easier for the compiler to quickly search the identifier record & retrieve it.

Example :-

$x = a + b * c$

⇩

Lexical Analysis.

$id = id + id * id$

⇩

Syntax Analysis.

statement → id = E
(expression)

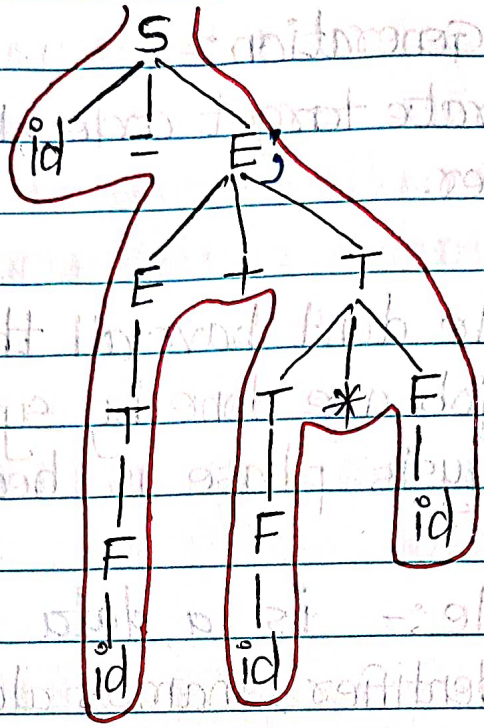
CFG

$S \rightarrow id = E;$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$



⇩

Semantic Analysis

(No semantic error)

⇩

(3 address code) Intermediate Code Generation (ICG)

⇩

```
t1 = b*c
t2 = a+t1
x  = t2
```

⇩

code optimization

```
t1 = b*c
x  = a+t1
```

Target Code Generation.

```
MULT   R1, R2        | a → R0
ADD    R0, R2        | b → R1
MOV    R2, X         | c → R2
```

Comparison between Assembler, Compiler & Interpreter

1] Assemblers:-

i) It is language processors based on language translation linking - loading model.

ii) Generate object program as an output.

iii) Program execution is separate from assembly process & performed only after the entire output (object code) program is produced.

iv) object code executes independently & generally does not need the presence of assembler in the memory.

v) An erroneous program will not generate object code.

vi) Need re-assembly of the program for generating afresh object code after each modification in the source program.

vii) TASM , MASM.

2] Compiler

i) It is a language processor based on language translation – linking – loading model.

ii) Generate object program as an output.

iii) — " —

iv) — " —

v) — " —

vi) — " —

vii) ex. C & C++ Compiler.

3] Interpreters:-

i) It is a class of language processor based on interpretation model.

ii) Do not generate any output program.

iii) Program execution is a part of interpretation & is performed on a statement-by-statement basis.

iv) Exist in the memory during interpretation.

v) Evaluate & execute program statement until an error is found.

vi) Independent of program modification issues as it processes the source program each time during execution

vii) ex. VB, LISP, MATLAB, HTML, Python.

ex 2] result = cost - rate * 66 ... 3) A = B+C

Lexical Analysis

id = id - id * Literal

⇓

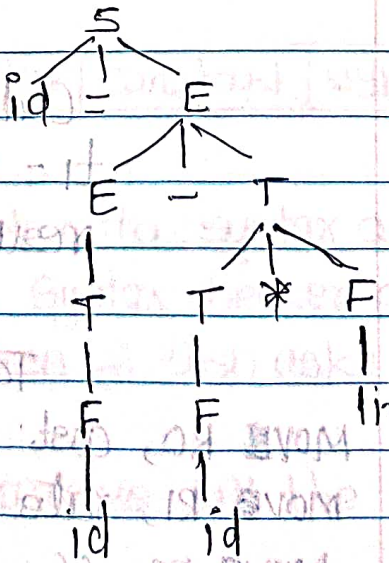
Syntax Analysis

S → id = E

E → E - T | T

T → T * F | F

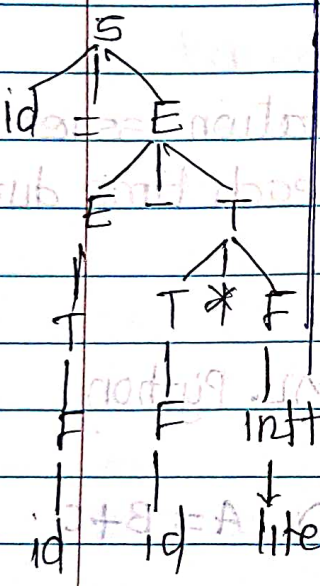
F → id | Literal



↓↓

Semantic Analysis (type checking)

If rate is a float, the integer literal 66 should be converted to a float before multiplying.



ex

float a;	float a, b=5;
a=5/2;	a=b/2;
printf("\na=%f", a);	printf("\na=%f", a)
⇒ a=2.0000	⇒ a=2.5000

↓↓

Intermediate code generation

t1 = rate * 66

t2 = cost - t1

result = t2

↓↓

Code optimization

t1 = rate * 66

result = cost - t1

↓↓

Target code generation

MOVE R0, cost

MOVE R1, rate

MOVE R3, 66

MULT R1, R3.

SUB R0, R3

MOVE R0, result