Properties of a compiler:

When a compiler is built it should posses following properties:

- The compiler itself must be bug free
- It must generate correct machine code.
- The generated machine code must run fast.
- The compiler itself must run fast (compilation time must be proportional to program size).
- The compiler must be portable (i.e, modular, supporting separate compilation).
- It must give good diagnostics and error msgs.
- The generated code must work well with existing debuggers.
- It must have consistent optimization.

Why should we study compilers ?

Following are some reasons behind the study of compilers :

- The machine performance can be measured by amount of compiled code.
- Various programming tools such as debugging tools, source code validating tools can be developed for the convinience of programmer.
- The study of compilers inspire to develop new programming languages that satisfies the need of programmer.
- The study of compilers helped to develop query languages, object orienled Programming Langs and visual programming languages.
- Various system building tools such as LEX and YACC can be developed by analytical study of compilers.

Features of Good Compiler

Various features of Good compiler are as follows:

- The good compiler compiles the large amount of code in less time.
- The good compiler requires less amount of memory space to compile the source code.
- The good compiler can compile only the modified code segment if frequent modifications are required in the source program [Incremental compiler]
- While handling the hardware interrupts the good compilers interact closely with operating System

Phases of a Compiler:

1. Lexical Analysis

- It is also called scanning
- It is the phase of compilation in which complete code is scanned and broken down into group of strings called 'tokens'.
- A token is a sequence of characters having collective meaning
- A token can be an identifier, keyword, relational operator, reserved word etc

eg:- total := count + rate * 60

The Lexical Analysis phase scans the given string from left to right and broken down into series of tokens.

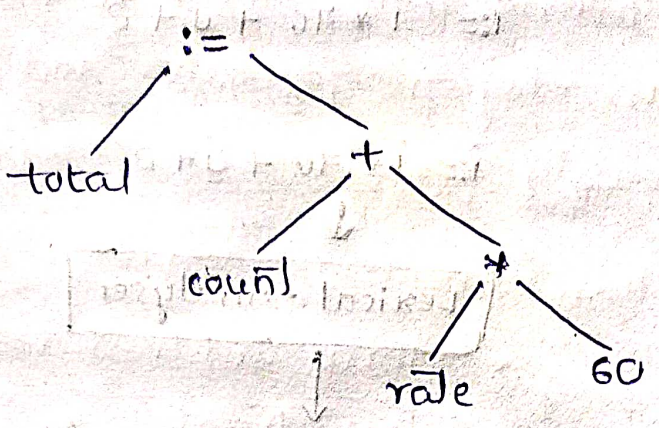
total - Identifier
:= - operator
count - Identifier
+ - operator
rate - Identifier
* - operator
60 - constant

while scanning the string, this phase eleminates all the blank spaces.

2. Syntax Analysis:

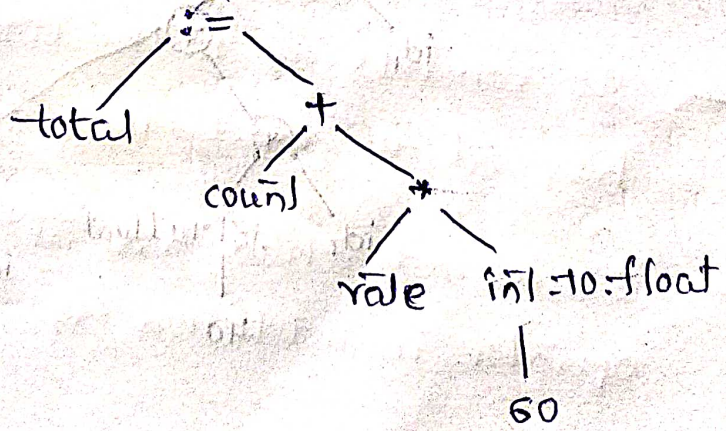
- It is also called Parsing
- In this phase, the tokens generated by Lexical Analyzer are grouped to form a hierarchical structure.
- The hierarchical structure generated in this phase called "Parse Tree" (or) "Syntax Tree".

For the expression total := count + rate * 60
the parse tree can be generated as follows:



3. Semantic Analysis:

- Once the syntax is checked and parse tree is generated.
- The semantic analysis determines the meaning of the source string
- For example, meaning of source string means matching of "parenthesis" in the expression, (or) matching of "if ... else" statements (or) performing arithmetic operations of the expr. that are type compatible (or) checking the scope of operation.



4. Intermediate Code Generation:

- Intermediate code is a kind of code which is easy to generate and this code is easily converted to target code.
- This code is in variety of forms such as
  - * three address code
  - * quadraple
  - * triple
  - * posix
- Here, we will consider three address code as intermediate code.

eg:-

$t_1 := \text{int to float (60)}$

$t_2 := \text{rate} * t_1$

$t_3 := \text{count} * t_2$

$\text{total} = t_3$

There are certain properties which should be possessed by three address code.

- It consists of instructions each of which has at the most three operands.
- Each instruction has at the most one operator in addition to assignment.

- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some instructions may have fewer than three operands for example first and last instructions of above given three address code.

i.e., $t_1 := \text{int to float (60)}$

$\text{total} := t3$

5. Code Optimization:

- The code optimization phase attempts to improve the intermediate code.
- Thus, it is necessary to have a faster executing code (or) less consumption of memory
- optimizing the code means, overall running time of target program can be improved.

eg:

$t_1 := \text{rate} * 60.0$

$t_2 := \text{count} + t1$

$\text{total} := t_2$

6. Code Generation:

- In code generation phase, the target code gets generated.
- The intermediate code instructions are transferred into equivalent sequence of machine instructions.

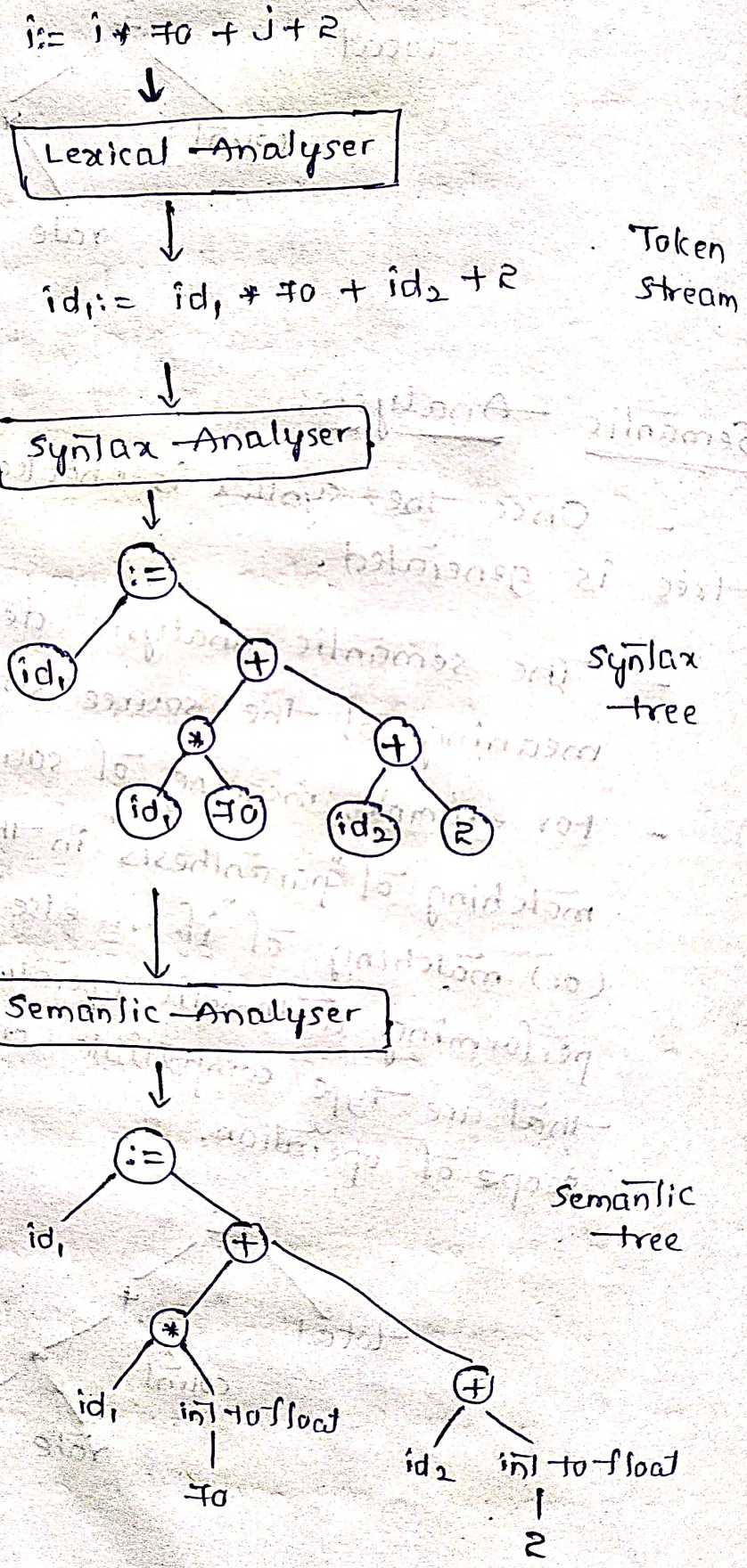
eg:

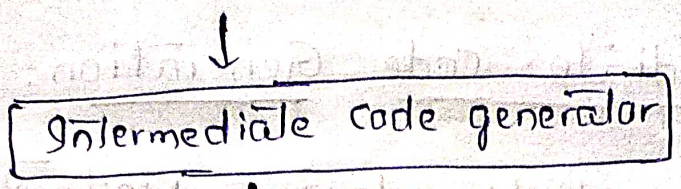
```
MOVF rate, R1
MULF #60.0, R1
MOVF count, R2
ADDF R2, R1
MOVF R1, total
```

Eg:- Show the output generated by each phase for the following expression:

$$i := i * 10 + j + 2$$

sol:





$t_1 := \text{int to float}(\#0)$

$t_2 := id_1 * t_1$

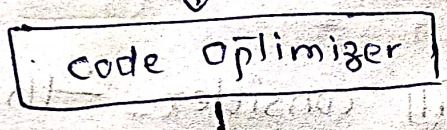
$t_3 := \text{int to float}(2)$

$t_4 := id_2 * t_3$

$t_5 = t_2 + t_4$

$id_1 = t_5$

Intermediate code



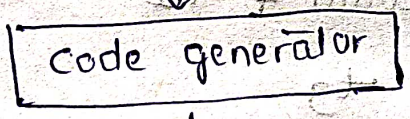
$t_1 := id_1 * \#0.0$

$t_2 := id_2 + 2.0$

$t_3 := t_1 + t_2$

$id_1 := t_3$

Optimized code




```
MOVF  id1, R1
MULF  #0.0, R1
MOVF  id2, R2
ADDF  #2.0, R2
ADDF  R2, R1
MOVF  R1, id1
```

Machine code