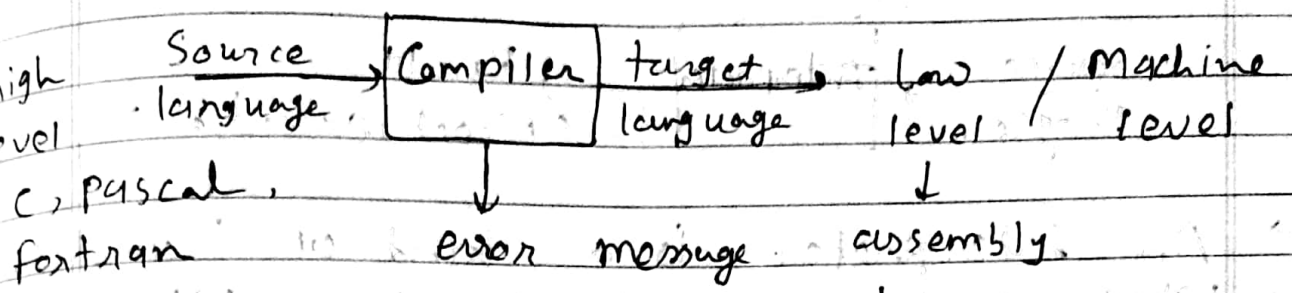
# Chapter: 1 Overview of translation process

Compiler:




→ Compiler is a program that can read a program in one language - source language - and translate it into an equivalent program in another language.

→ Compiler converts higher language into lower level language or machine language.

for example:
There is a C program: Called Hellowcrld.c

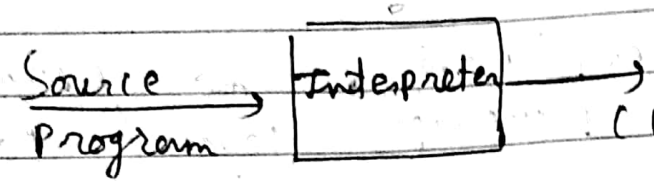
```
#include <stdio.h>
{ printf("Helloworld"); }
```

So, Computer can not understand higher level language. So, Compiler converts it into binary language in the form of 0 and 1. means Compiler create object file. linking loader create the executable file from object file.

example of Compiler:
gcc, Javac, Microsoft visual Studio.

line by line execution

Interpreter : Convert one line at a time




→ An Interpreter is a kind of translator which produces result directly when Source language and data is given to it.

Example of Interpreter :
Ruby, Python.

difference between Compiler and Interpreter

| | | |
|---|---|---|
| 1 | In the process of Compilation the program is analyzed only once and then code is generated | |
| 2 | The Source program gets Interpreted every time it is to be executed | |
| 1 | Compiler is efficient than interpreter | |
| 2 | Interpretation is less efficient than Compiler | |
| 1 | Compilers produce object code. | |
| 2 | Interpreters do not produce object code. | |
| 1 | Complex program | |
| 2 | Simple program | |

1 Source Program → [Compiler] → Object Code → [Loader] → executable Code → [Compiler] → output

2 Source Program → [Interpreter] → output
Data → [Interpreter]

1 example : Ups debugger

2 example : Turbo C Compiler

Cousins of Compiler : / Translator :
language processing System : typical Compilation

1 Context of Compiler :

Source Program
↓
[Preprocessor]
↓ modified source program
[Compiler]
↓ target assembly program
[assembler]
↓ relocatable machine code
[loader/linker]
↓
Absolute machine Code

1 Pre processor :
A Source program may be divided into modules stored in separate files. The task of Collecting the Source program is Sometimes entrusted to Separate program Called pre processor.

Preprocessors produce input to Compilers.

example :
C preprocessor : is not a part of the Compiler, but is Separate step in the Compilation process.

All Preprocessor Commands begin with # Symbol.
#include : insert a peiticular header from another file.
#include <stdio.h>
These directives tell the cpp to get stdio.h from System libraries and add text to the current Source file.

→ Preprocessor also allow a user to define macros that are shorthands for longer Constructs.

2 Compiler : The modified Source program is then fed to the Compiler. The Compiler may produce an assembly language program as its output because assembly language is easier to produce as output and easier to debug.

object Code : is portion of machine code that has not been linked yet into tenget program.

3 Assembler :

Some Compilers produce assembly Code as in figure (a). that is passed to the assemble for further processing.

→ other Compilers performs the job of the assembler producing relocatable machine Code that can be directly passed to the loader / link-editor.

→ Assembly Code is mnemonic version of the machine Code in which names are used instead of binary codes for operations and names are also given to memory addresses.

```
MOV a, R1
ADD #2, R1
MOV R1, b.
```

→ Assembler produces relocatable machine Code as output.

relocatable Machine Code : object Code.

PAGE No.
DATE

4 Loader and link editors

→ loader performs the two functions: loading and link editing.

→ The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at the proper locations.

→ The link editor allows us to make a single program from several files of relocatable machine code.

→ linker / loader then generates absolute / target machine code:

absolute machine code: executable code. or target program.

Hierarchy of language:

There are three types of language:

1 Machine language } low level language
2 Assembly language
3 High-level language

low level languages are closer to the language used by computer, while high-level languages are closer to human languages.

language : specifies a set of instructions that can be used to produce various kinds of output.

1 Machine language :

→ Machine language is a collection of binary digits or bits that the computer reads and interprets.

→ Machine languages are the only languages understood by computers.

→ But Machine languages are almost impossible for humans to use because they consist entirely of numbers.

Example :
let us say that an electric toothbrush has a processor and main memory. Processor can rotate the bristles left and right, and can check the on/off switch. The machine instructions are one byte long and correspond to the following machine operations :

| Machine Instruction | Machine operation |
|---|---|
| 0000 0000 | Stop |
| 0000 0001 | Rotate bristles left |
| 0000 0010 | Rotate bristles right |
| 0000 0100 | go back to start of program |
| 0000 1000 | Skip next instruction if switch is off. |

2 Assembly language :

→ is a low level Programming language for a computer or other programmable device or microprocessors.

assembly language consist of series of mnemonic instructions followed by a list of data, arguments or parameters.

→ These are translated by an assembler into machine language instructions. that can be loaded into memory and executed.

3 High level language:

→ High level languages allow us to write computer code using instructions resembling everyday spoken language (for ex: print, if, while) which then translated into machine language to be executed.

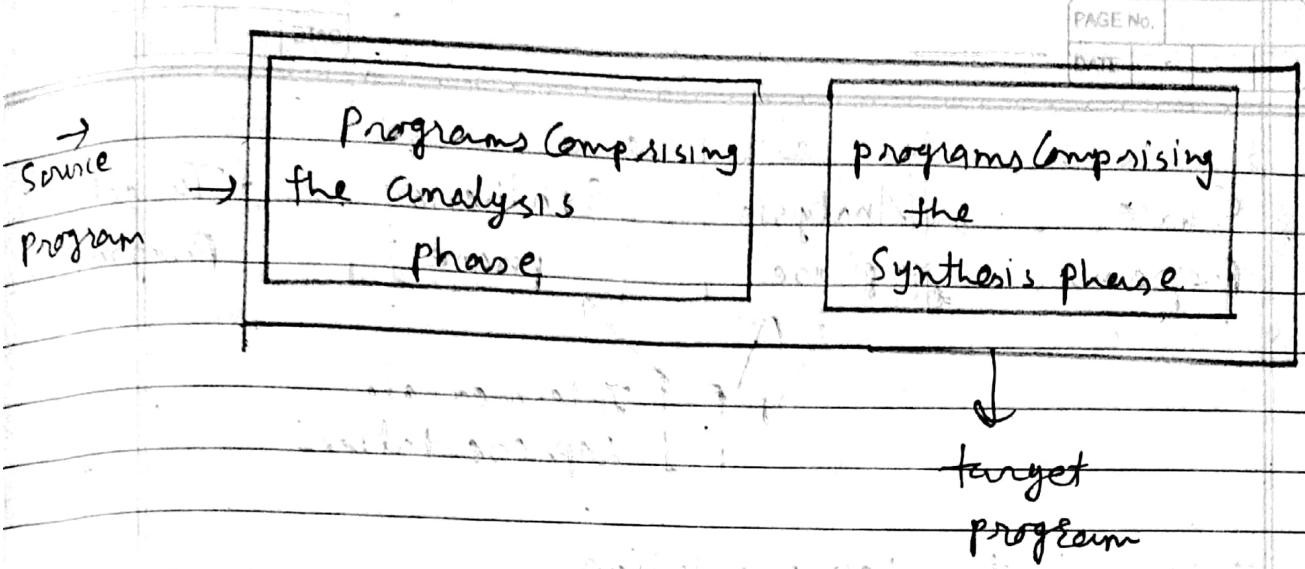
→ programs written in high level language need to translated into machine language before they can be executed.

→ Some programming language use Compiler to perform this translation and others use an interpreter.

passes of Compiler:

1 Single pass Compiler:
2 Multi pass Compiler:

Pass: one complete scan of source program is called pass.




→ In a Single pass Compiler analysis of Source program is Immediately followed by Synthesis of equivalent target statement/ program.

→ It is difficult to Compile the Source program into Single pass due to :

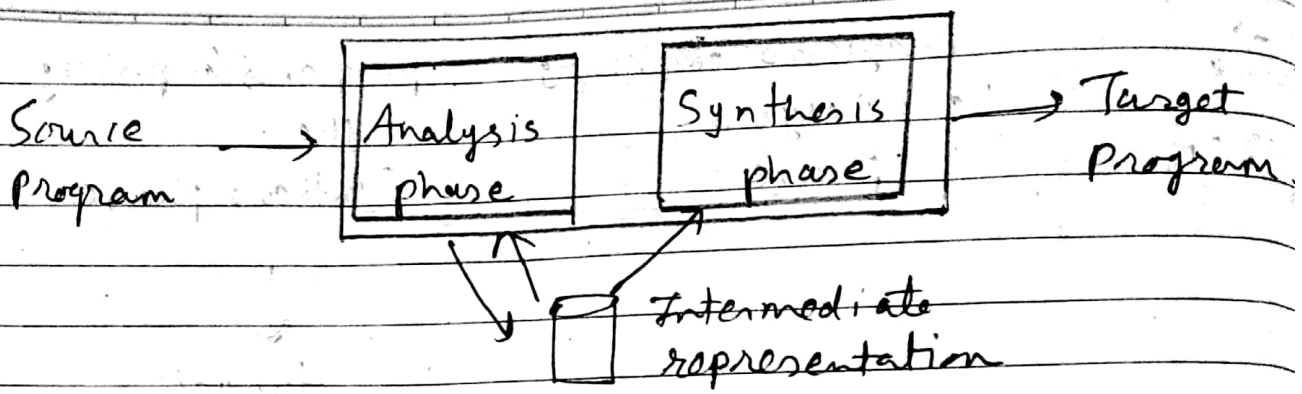
Forward Reference : a forward reference entity is a reference to the entity which precedes its definition in the program.

→ using an identifier before declaration.

Multi pass Compiler :

→ In pass I : perform analysis of the Source program and note relevant information.

In pass II : generate target code using information noted in pass I / perform Synthesis of the target Program.




Intermediate representation:
Compiler generate an easy to represent form of source language which is called IR.

→ difference between Single pass and multi pass Compiler:
1 2

| | 1 | 2 |
|---|---|---|
| 1 | One pass Compiler is a Compiler that passes through the Source Code of each Compilation unit only once. | A multi pass Compiler is a type of Compiler that processes the Source Code or AST of program several times. |
| 1 | faster | |
| 2 | Slower | |
| 1 | Narrow Compiler | |
| 2 | Wide Compiler | |

1 language like pascal can be implemented with single pass compiler

2 languages like Java requires a multi-pass compiler.

difference between compiler and Assembler

1 high level language to machine code
2 mnemonic op code to machine code

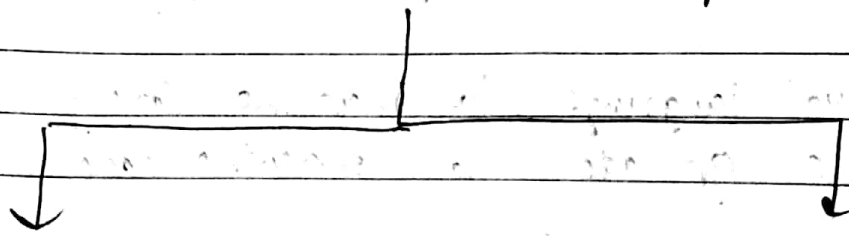
1 Types of Compiler : Single pass compiler
Multi pass compiler
2 Types of assembler : single pass assembler
multi pass assembler
1 C Compiler
2 8085, 8086 instruction set

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.

Compiler which run on one machine and produces the target code for another machine

Analysis Synthesis model of Compilation / Structure of Compiler
Phases of Compiler
Write output of phases of a Compiler. for a = a + b * c * 2 ; type of a, b, c are float.

→ two Parts of Compilation process.



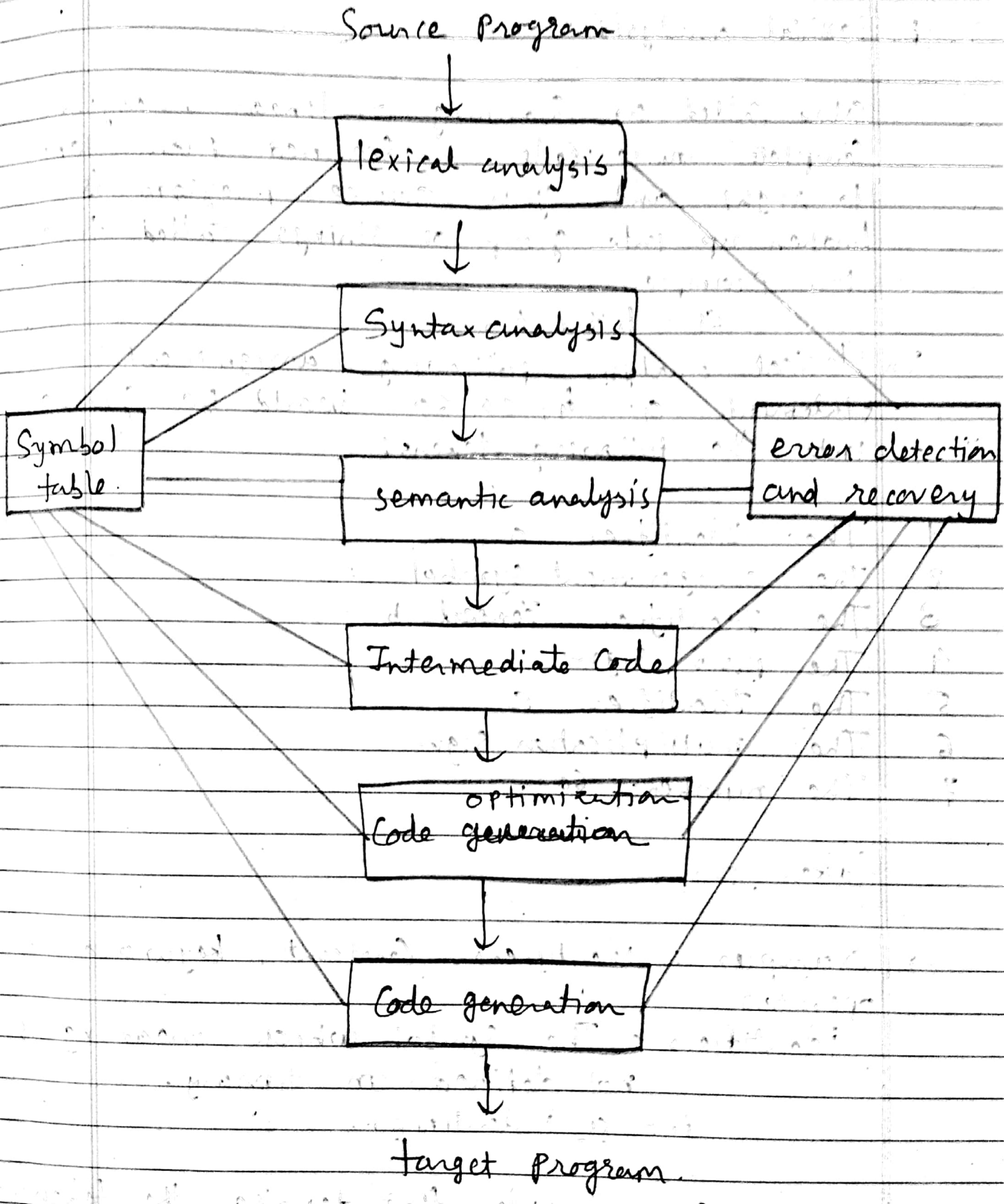
Analysis phase  Synthesis phase

Analysis phase : The main objective of the analysis phase is to break the source code into parts and then arranges these pieces into a meaningful structure.

Analysis Phase is divided into three subpart
1 lexical analysis
2 Syntax analysis
3 Semantic analysis.

Synthesis phase : Synthesis phase is concerned with generation of target language statement which has the same meaning as the source statement

4 Intermediate Code generation
5 Code optimization
6 Code generation.



1 lexical analysis: remove the whitespace or comment in the source code.
Also called as Scanning or linear Analysis.
Complete Source Code is Scanned from left to right and your source program is broken up into group of strings called token.
for example.

$a = b + c * 60$

In lexical analysis following assign ment statement $a := b + c * 60$ would be grouped into the following tokens:

1 The Identifier a
2 The assign ment symbol =
3 The Identifier b
4 The plus sign
5 The Identifier c
6 The multiplication sign
7 The number 60.

token : Sequence of characters that can be treated as single logical entity.

→ examples : identifiers, Constant, keywords, operator.

identifiers : The Symbol which meaning is not defined in library.

int a → identifiers.

pattern : set of rules that describe the token

lexeme : Sequence of characters in the source program that are matched with pattern

lexical errors:
keywords: predefined reserved words used in programming that have special meaning to the Compiler.

of token.

int a

lexeme identifier token.

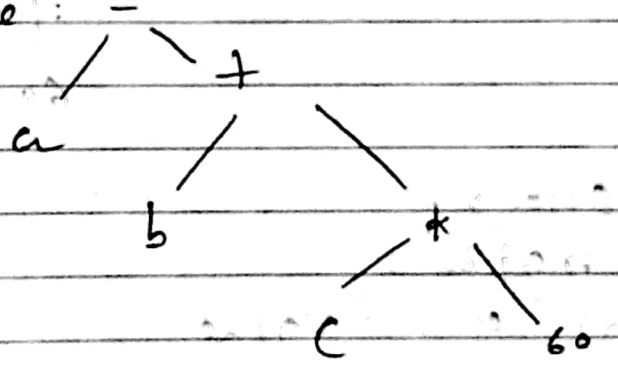
| lexeme | token |
|---|---|
| int | keyword |
| a | identifier |

pattern: letter followed by letters and digits.

2 Syntax Analysis:
also called as Parsing or Hierarchical analysis
It involves grouping of tokens of the Source program into grammatical phrases that are used by the Compiler to synthesize output.
Or generate Syntax errors.

for the expression: a = b + c * 60

parse tree:

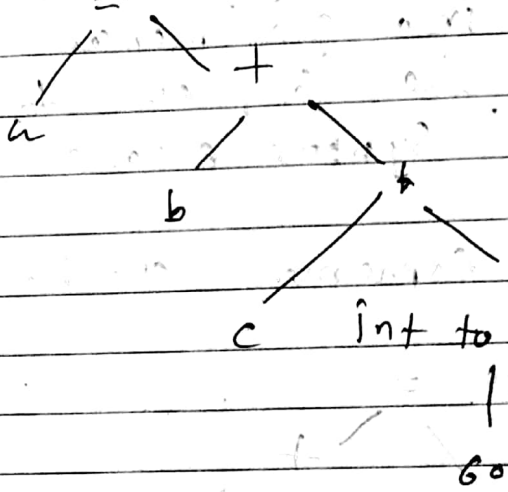



→ parsing can be done in two ways: top down and bottom-up.
Here, the expression are:
b + c and c * 60
So, we will evaluate the c * 60 because * has higher priority than +.

↑, *, / +, -

3 Semantic analysis :
The next phase is Semantic analysis. Which determines the meaning of Source string means matching of parenthesis, matching of if .... else statements

→ main part of Semantic analysis is type checking, type Conversion.




Semantic errors :
1 type mismatch :
int a = "value"
2 undeclared variable

4 Intermediate Code generation :
It should have two properties :
1 It should be easy to produce
2 easy to translate into the target program

This Code is in variety of forms such as three address Code, triple, quadruples, posix.

Here, we will Consider an Intermediate Code in three address code

Three address Code Consists of instructions each of which has atmost three operands

```
t1 := int to real (60)
t2 := c * t1
t3 := b + t2
a  := t3
```

1→ Each instruction has at most one operator in addition to assignment.

2→ Compiler must generate a temporary name to hold the value computed by each instruction.

3→ Some instruction may have fewer than three operands.

5 Code optimization :

→ used to improve the intermediate Code for faster execution and less ~~~~ Consumption of memory.

→ It eliminate the redundant instructions.

```
t1 := c * int to real (60)
a  := b + t1
```

6 Code generation:

→ In Code generation phase the target Code gets generated.

→ The Intermediate instruction are translated into sequence of machine instructions.

ex:
```
MOV  C, R1      The # signifies that
MUL  #60.0, R1  60.0 is to be treated
MOV  b, R2      as a Constant.
ADD  R2, R1
MOV  R1, a
```

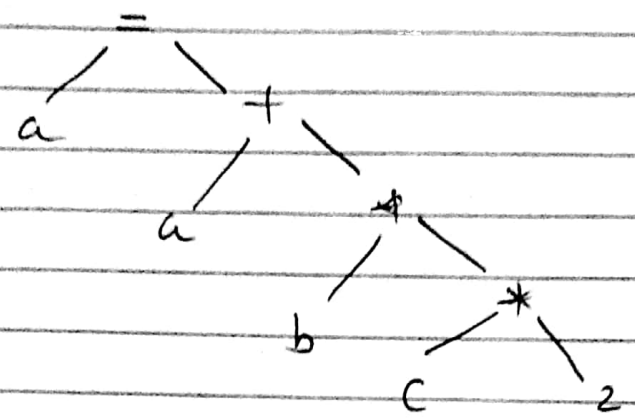
3B-5 for a statement given below, write output of all phases (except of an optimization phase) of a Compiler. Assume a, b, and c of type float.

```
a = a + b * c * 2;
```

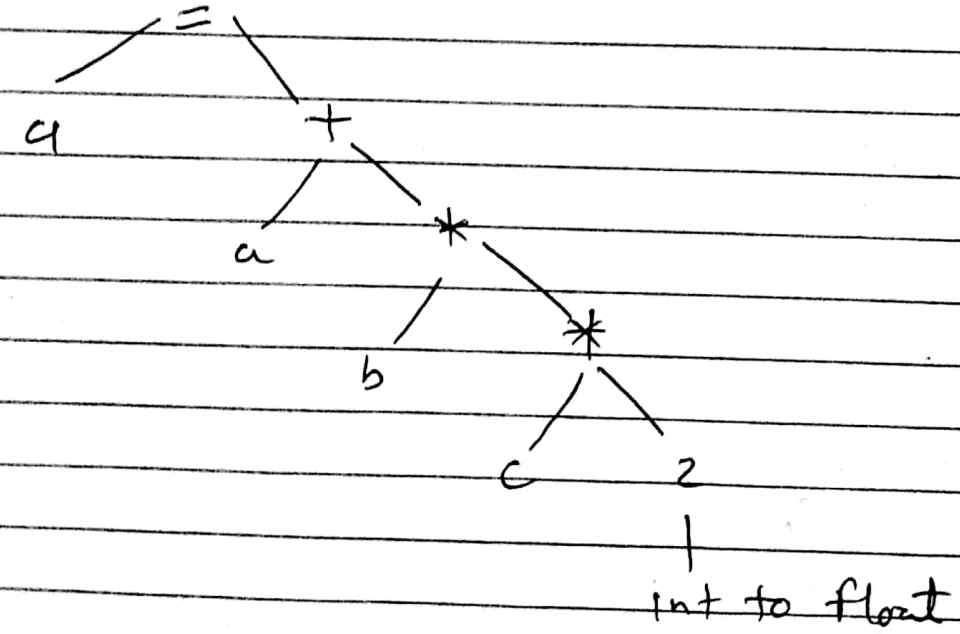
(1) lexical analysis:

→ 1) The identifier a
2) The assignment symbol =
3) The identifier b
4) The multiplication sign
5) The identifier c
6) The multiplication sign
7) The number 2.

② Syntax analysis :




③ Semantic analysis :




④ Intermediate code generation :

$t_1 := \text{int to float}(2)$

$t_2 := c * t_1$

$t_3 := b * t_2$

$t_4 := a + t_3$

$a := t_4$

⑥ Code generation:

```
MOV   c,   R1
MUL   #2.0, R1
MOV   b,   R2
MUL   R2,  R1
MOV   a,   R2
ADD   R2,  R1
MOV   R1,  a
```